

C++ Tutorial



C/C++ Training Certification

javaTpoint

C++ tutorial provides basic and advanced concepts of C++. Our C++ tutorial is designed for beginners and professionals.

C++ is an object-oriented programming language. It is an extension to [C programming](#).

Our C++ tutorial includes all topics of C++ such as first example, control statements, objects and classes, [inheritance](#), [constructor](#), destructor, this, static, polymorphism, abstraction, abstract class, interface, namespace, encapsulation, arrays, strings, exception handling, File IO, etc.

What is C++

C++ is a general purpose, case-sensitive, free-form programming language that supports object-oriented, procedural and generic programming.

C++ is a middle-level language, as it encapsulates both high and low level language features.

Object-Oriented Programming (OOPs)

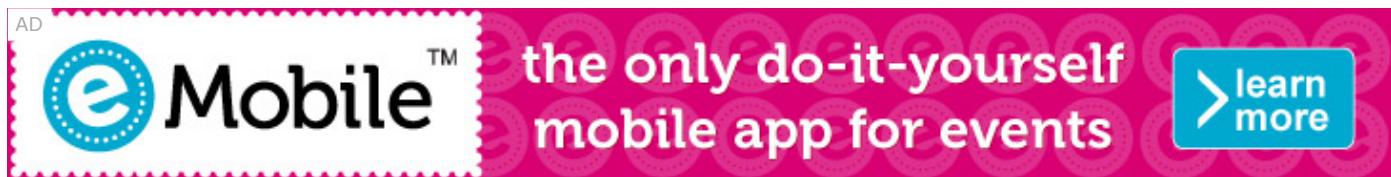
C++ supports the object-oriented programming, the four major pillar of object-oriented programming ([OOPs](#)) used in C++ are:

1. Inheritance
2. Polymorphism
3. Encapsulation
4. Abstraction

C++ Standard Libraries

Standard C++ programming is divided into three important parts:

- The core library includes the data types, variables and literals, etc.
- The standard library includes the set of functions manipulating strings, files, etc.
- The Standard Template Library (STL) includes the set of methods manipulating a data structure.



Usage of C++

By the help of C++ programming language, we can develop different types of secured and robust applications:

- Window application
- Client-Server application
- Device drivers
- Embedded firmware etc



C++ Program

In this tutorial, all C++ programs are given with C++ compiler so that you can easily change the C++ program code.

File: main.cpp

```
#include <iostream>
using namespace std;
int main() {
    cout << "Hello C++ Programming";
    return 0;
}
```

}

A detailed explanation of first C++ program is given in next chapters.

C++ Index

C++ Tutorial	C++ STL Deque	C++ STL Multiset
o C++ Tutorial	o C++ Deque	o C++ multiset
o What is C++	o Deque assign() function	o multiset constructor
o C vs C++	o Deque emplace() function	o multiset destructor
o C++ History	o Deque emplace_back() function	o multiset begin() function
o C++ Features	o Deque emplace_front() function	o multiset end() function
o C++ Installation	o Deque insert() function	o multiset operator=
o C++ Program	o Deque push_back() function	o multiset rbegin() function
o C++ cout, cin, endl	o Deque push_front() function	o multiset rend() function
o C++ Variable	o Deque function	o multiset cbegin() function
o C++ Data types	o Deque function	o multiset cend() function
o C++ Keywords	o Deque function	o multiset max_size()
o C++ Operators	o Deque function	o multiset size() function
C++ Control Statement		
o C++ if-else	o Deque function	o multiset crbegin() function
o C++ switch	o Deque swap() function	o multiset crend()
o C++ For Loop	o Deque clear() function	
o C++ While Loop	o Deque empty() function	
o C++ Do-While Loop	o Deque erase() function	
o C++ Break Statement	o Deque function	
o C++ Continue Statement	o Deque max_size() function	
o C++ Goto Statement	o Deque resize() function	
o C++ Comments	o Deque shrink_to_fit() function	

C++ Functions	<ul style="list-style-type: none"> ○ C++ Functions ○ Call by value & reference ○ C++ Recursion ○ C++ Storage Classes 	<ul style="list-style-type: none"> ○ Deque size() function ○ Deque at() function ○ Deque operator[]() function ○ Deque operator=() function ○ Deque back() function ○ Deque begin() function ○ Deque cbegin() function ○ Deque end() function ○ Deque cend() function ○ Deque rbegin() function ○ Deque crbegin() function ○ Deque rend() function ○ Deque crend() function 	<ul style="list-style-type: none"> ○ multiset erase() function ○ multiset swap() function ○ multiset emplace_hint() function ○ multiset find() function ○ multiset key_comp() function ○ multiset value_comp() function ○ multiset count() ○ multiset equal_range() ○ multiset lower_bound() ○ multiset upper_bound() ○ multiset get_allocator() ○ multiset operator== ○ multiset operator!= ○ multiset operator< ○ multiset operator<= ○ multiset operator> ○ multiset operator>= ○ multiset swap() function
C++ Arrays	<ul style="list-style-type: none"> ○ C++ Arrays ○ C++ Array to Function ○ Multidimensional Arrays 	<ul style="list-style-type: none"> ○ Deque size() function ○ Deque at() function ○ Deque operator[]() function ○ Deque back() function ○ Deque begin() function ○ Deque cbegin() function ○ Deque end() function ○ Deque cend() function ○ Deque rbegin() function ○ Deque crbegin() function ○ Deque rend() function ○ Deque crend() function 	<ul style="list-style-type: none"> ○ multiset erase() function ○ multiset swap() function ○ multiset emplace_hint() function ○ multiset find() function ○ multiset key_comp() function ○ multiset value_comp() function ○ multiset count() ○ multiset equal_range() ○ multiset lower_bound() ○ multiset upper_bound() ○ multiset get_allocator() ○ multiset operator== ○ multiset operator!= ○ multiset operator< ○ multiset operator<= ○ multiset operator> ○ multiset operator>= ○ multiset swap() function
C++ Pointers	<ul style="list-style-type: none"> ○ C++ Pointers 	<ul style="list-style-type: none"> ○ Deque size() function ○ Deque at() function ○ Deque operator[]() function ○ Deque back() function ○ Deque begin() function ○ Deque cbegin() function ○ Deque end() function ○ Deque cend() function ○ Deque rbegin() function ○ Deque crbegin() function ○ Deque rend() function ○ Deque crend() function 	<ul style="list-style-type: none"> ○ multiset erase() function ○ multiset swap() function ○ multiset emplace_hint() function ○ multiset find() function ○ multiset key_comp() function ○ multiset value_comp() function ○ multiset count() ○ multiset equal_range() ○ multiset lower_bound() ○ multiset upper_bound() ○ multiset get_allocator() ○ multiset operator== ○ multiset operator!= ○ multiset operator< ○ multiset operator<= ○ multiset operator> ○ multiset operator>= ○ multiset swap() function
C++ Object Class	<ul style="list-style-type: none"> ○ C++ OOPs Concepts ○ C++ Object Class ○ C++ Constructor ○ C++ Destructor ○ C++ this Pointer ○ C++ static ○ C++ Structs ○ C++ Enumeration ○ C++ Friend Function 	<ul style="list-style-type: none"> ○ C++ List ○ List insert() function ○ List push_back() function ○ List push_front() function ○ List pop_back() function ○ List pop_front() function ○ List empty() function ○ List size() function ○ List max_size() function ○ List front() function ○ List back() function ○ List swap() function ○ List reverse() function 	<ul style="list-style-type: none"> ○ multiset erase() function ○ multiset swap() function ○ multiset emplace_hint() function ○ multiset find() function ○ multiset key_comp() function ○ multiset value_comp() function ○ multiset count() ○ multiset equal_range() ○ multiset lower_bound() ○ multiset upper_bound() ○ multiset get_allocator() ○ multiset operator== ○ multiset operator!= ○ multiset operator< ○ multiset operator<= ○ multiset operator> ○ multiset operator>= ○ multiset swap() function
C++ Inheritance	<ul style="list-style-type: none"> ○ C++ Inheritance ○ C++ Aggregation 	<ul style="list-style-type: none"> ○ C++ Multimap ○ multimap crbegin() ○ multimap crend() 	<ul style="list-style-type: none"> ○ C++ Multimap ○ multimap crbegin() ○ multimap crend()
C++ Polymorphism	<ul style="list-style-type: none"> ○ C++ Polymorphism ○ C++ Overloading 	<ul style="list-style-type: none"> ○ C++ Multimap ○ multimap crbegin() ○ multimap crend() 	<ul style="list-style-type: none"> ○ C++ Multimap ○ multimap crbegin() ○ multimap crend()

- o C++ Overriding
 - o C++ Virtual Function

C++ Abstraction

 - o C++ Interfaces
 - o C++ Data Abstraction

C++ Namespaces

 - o C++ Namespaces

C++ Templates

 - o C++ Templates

C++ Strings

 - o C++ Strings

C++ Exceptions

 - o C++ Exception Handling
 - o C++ try/catch
 - o C++ User-Defined

C++ File & Stream

 - o C++ File & Stream

C++ Programs

 - o C++ Programs
 - o Fibonacci Series
 - o Prime Number
 - o Palindrome Number
 - o Factorial
 - o Armstrong Number
 - o Sum of digits
 - o Reverse Number

- List sort() function
 - List merge() function
 - List splice() function
 - List unique() function
 - List resize() function
 - List assign() function
 - List emplace() function
 - List emplace_back() function
 - List emplace_front() function

C++ STL Map

 - C++ Map
 - map at() function
 - map begin() function
 - map cbegin() function
 - map cend() function
 - map crbegin() function
 - map crend() function
 - map empty() function
 - map end() function
 - map max_size() function
 - map operator[]
 - map rbegin() function
 - map rend() function
 - map size() function
 - map clear() function
 - map emplace() function
 - map emplace_hint() function
 - map erase() function

- multimap function begin()
 - multimap function cbegin()
 - multimap function cend()
 - multimap end() function
 - multimap function rbegin()
 - multimap function rend()
 - multimap function clear()
 - multimap function emplace()
 - multimap function empty()
 - multimap function erase()
 - multimap function insert()
 - multimap function swap()
 - multimap equal_range() function
 - multimap operator==
 - multimap operator!=
 - multimap operator<
 - multimap operator<=
 - multimap operator>
 - multimap operator>=
 - multimap function swap()

- Swap Number
- Matrix Multiplication
- Decimal to Binary
- Number in Characters
- Alphabet Triangle
- Number Triangle
- Fibonacci Triangle

C++ STL Tutorial

- C++ STL Algorithm
- algorithm adjacent_find() function
- algorithm any_of() function
- algorithm copy() function
- algorithm copy_if() function
- algorithm count() function
- algorithm count_if() function
- algorithm equal() function
- algorithm find() function
- algorithm find_end() function
- algorithm find_first_of() function
- algorithm find_if() function
- algorithm find_if_not() function
- algorithm for_each() function

- map find() function
- map insert() function
- map operator=()
- map swap() function

C++ STL Math

- math cos() function
- math sin() function
- math tan() function
- math acos() function
- math asin() function
- math atan() function
- math atan2() function
- math cosh() function
- math sinh() function
- math tanh() function
- math acosh() function
- math asinh() function
- math atanh() function
- math exp() function
- math frexp() function
- math ldexp() function
- math log() function
- math log10() function
- math modf() function
- math exp2() function
- math expm1() function
- math log1p() function
- math log2() function
- math logb() function
- math scalbn() function

- C++ Set
- set constructor
- set destructor
- set operator=() function
- set begin() function
- set cbegin() function
- set end() function
- set cend() function
- set rbegin() function
- set rend() function
- set crbegin() function
- set crend() function
- set empty() function
- set Size() function
- set max_size() function
- set insert() function
- set erase() function
- set Swap() function
- set clear() function
- set emplace() function
- set emplace_hint() function
- set key_comp() function
- set value_comp() function
- set find() function
- set count() function
- set lower_bound() function
- set upper_bound() function

o algorithm function	move()	o math scalbln() function	o set function
o algorithm function	all_of()	o math ilogb() function	o set get_allocator()
o algorithm copy_backward() function		o math copysign()	o set operator==
o algorithm function	copy_n()	o math nextafter()	o set operator!=
o algorithm function	search()	o math nexttoward()	o set operator<
o algorithm is_permutation() function		o math fdim()	o set operator<=
o algorithm mismatch() function		o math fmax()	o set operator>
o algorithm move_backward() function		o math fmin()	o set operator>=
o algorithm function	none_of()	o math pow()	o set swap() function
o algorithm function	search_n()	o math sqrt()	C++ STD Strings
o algorithm function	swap()	o math cbrt()	o string compare() function
o algorithm fill() function		o math hypot()	o string length() function
o algorithm function	iter_swap()	o math ceil()	o string swap() function
o algorithm replace_copy_if() function		o math floor()	o string substr() function
o algorithm replace_copy() function		o math round()	o string size() function
o algorithm replace_if() function		o math lround()	o string resize() function
o algorithm replace() function		o math llround()	o string replace() function
		o math fmod()	o string append() function
		o math trunc()	o string at() function
		o math rint()	o string find() function
		o math lrint()	o string find_first_of() function
		o math llrint()	o string find_first_not_of() function
		o math nearbyint()	o string find_last_of() function
		o math remainder()	o string find_last_not_of() function
		o math remquo()	o string insert() function
		o math fabs()	
		o math abs()	

- algorithm swap_ranges() function
- algorithm transform() function
- algorithm fill_n() function
- algorithm generate_n() function
- algorithm generate() function
- algorithm remove() function
- algorithm is_partitioned() function
- algorithm random_shuffle() function
- algorithm remove_copy_if() function
- algorithm remove_copy() function
- algorithm partition_copy() function
- algorithm partition_point() function
- algorithm partition() function
- algorithm remove_if() function
- algorithm reverse_copy() function
- algorithm reverse() function

- math fma() function
 - math fpclassify() function
 - math isfinite() function
 - math isnan() function
 - math isinf() function
 - math isnormal() function
 - math signbit() function
 - math isgreater() function
 - math isgreaterequal() function
 - math less() function
 - math islessequal() function
 - math islessgreater() function
 - math isunordered() function
 - math erf() function
 - math erfc() function
 - math tgamma() function
 - math lgamma() function
- C++ STL priority_queue**
- C++ priority_queue
 - priority_queue emplace() function
 - priority_queue empty() function
 - priority_queue pop() function

- string max_size() function
- string push_back() function
- string pop_back() function
- sstring assign() function
- string copy() function
- string back() function
- string begin() function
- string capacity() function
- string cbegin() function
- string cend() function
- string clear() function
- string crbegin() function
- string data() function
- string empty() function
- string erase() function
- string front() function
- string operator+=()
- string operator=()
- string operator[]()
- string rfind() function
- string end() function
- string rend() function
- string shrink_to_fit() function
- string c_str() function
- string crend() function
- string rbegin() function
- string reserve() function

- algorithm rotate_copy() function
- algorithm rotate() function
- algorithm shuffle() function
- algorithm stable_partition() function
- algorithm unique_copy() function
- algorithm unique() function
- algorithm is_sorted_until() function
- algorithm is_sorted() function
- algorithm lower_bound() function
- algorithm nth_element() function
- algorithm partial_sort_copy() function
- algorithm partial_sort() function
- algorithm sort() function
- algorithm stable_sort() function
- algorithm binary_search() function
- algorithm equal_range() function
- algorithm includes() function

- priority_queue push() function
 - priority_queue size() function
 - priority_queue swap() function
 - priority_queue top() function
- C++ STL Stack**
- C++ Stack
 - stack emplace() function
 - stack empty() function
 - stack pop() function
 - stack push() function
 - stack size() function
 - stack top() function
- C++ STL Queue**
- C++ Queue
 - queue back() function
 - queue emplace() function
 - queue empty() function
 - queue front() function
 - queue pop() function
 - queue push() function
 - queue size() function

- string get_allocator() function
- C++ STL Vector**
- C++ Vector
 - Vector at() function
 - Vector back() function
 - Vector front() function
 - Vector swap() function
 - Vector push_back() function
 - Vector pop_back() function
 - Vector empty() function
 - Vector insert() function
 - Vector erase() function
 - Vector resize() function
 - Vector clear() function
 - Vector size() function
 - Vector capacity() function
 - Vector assign() function
 - Vector operator=()
 - Vector operator[]()
 - Vector end() function
 - Vector emplace() function
 - Vector emplace_back() function
 - Vector rend() function
 - Vector rbegin() function

- o algorithm
inplace_merge()
function
- o algorithm merge()
function
- o algorithm set_union()
function
- o algorithm upper_bound()
function

C++ STL Bitset

- o C++ Bitset
- o bitset all() function
- o bitset any() function
- o bitset count() function
- o bitset flip() function
- o bitset none() function
- o bitset operator[]
- o bitset reset() function
- o bitset set() function
- o bitset size() function
- o bitset test() function
- o bitset to_string()
function
- o bitset to_ullong()
function
- o bitset to_ulong()
function

- o Vector begin() function
- o Vector max_size()
function
- o Vector cend() function
- o Vector cbegin() function
- o Vector crbegin()
function
- o Vector crend()
function
- o Vector data() function
- o Vector shrink_to_fit()
function

Interview Questions

- o Interview Questions

Prerequisite

Before learning C++, you must have the basic knowledge of C.

Audience

Our C++ tutorial is designed to help beginners and professionals.

Problem

We assure that you will not find any problem in this C++ tutorial. But if there is any mistake, please post the problem in contact form.

Next →

AD

 [For Videos Join Our YouTube Channel: Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share



Learn Latest Tutorials

 [Splunk tutorial](#)

 [SPSS tutorial](#)

Difference between C and C++

What is C?

C is a structural or procedural oriented programming language which is machine-independent and extensively used in various applications.

C is the basic programming language that can be used to develop from the operating systems (like Windows) to complex programs like Oracle database, Git, Python interpreter, and many more. C programming language can be called a god's programming language as it forms the base for other programming languages. If we know the C language, then we can easily learn other programming languages. C language was developed by the great computer scientist Dennis Ritchie at the Bell Laboratories. It contains some additional features that make it unique from other programming languages.

What is C++?

C++ is a special-purpose programming language developed by **Bjarne Stroustrup** at Bell Labs circa 1980. C++ language is very similar to C language, and it is so compatible with C that it can run 99% of C programs without changing any source of code though C++ is an object-oriented programming language, so it is safer and well-structured programming language than C.

Let's understand the differences between C and C++.



The following are the differences between C and C++:

- **Definition**

C is a structural programming language, and it does not support classes and objects, while C++ is an object-oriented programming language that supports the concept of classes and objects.

- **Type of programming language**

C supports the structural programming language where the code is checked line by line, while C++ is an object-oriented programming language that supports the concept of classes and objects.

- **Developer of the language**

Dennis Ritchie developed C language at Bell Laboratories while Bjarne Stroustrup developed the C++ language at Bell Labs circa 1980.

- **Subset**

C++ is a superset of C programming language. C++ can run 99% of C code but C language cannot run C++ code.

- **Type of approach**

C follows the top-down approach, while C++ follows the bottom-up approach. The top-down approach breaks the main modules into tasks; these tasks are broken into sub-tasks, and so on. The bottom-down approach develops the lower level modules first and then the next level modules.

- **Security**

In C, the data can be easily manipulated by the outsiders as it does not support the encapsulation and information hiding while C++ is a very secure language, i.e., no outsiders can manipulate its data as it supports both encapsulation and data hiding. In C language, functions and data are the free entities, and in C++ language, all the functions and data are encapsulated in the form of objects.

- **Function Overloading**

Function overloading is a feature that allows you to have more than one function with the same name but varies in the parameters. C does not support the function overloading, while C++ supports the function overloading.

- **Function Overriding**

Function overriding is a feature that provides the specific implementation to the function, which is already defined in the base class. C does not support the function overriding, while C++ supports the function overriding.

- **Reference variables**

C does not support the reference variables, while C++ supports the reference variables.

- **Keywords**

C contains 32 keywords, and C++ supports 52 keywords.

- **Namespace feature**

A namespace is a feature that groups the entities like classes, objects, and functions under some specific name. C does not contain the namespace feature, while C++ supports the namespace feature that avoids the name collisions.

- **Exception handling**

C does not provide direct support to the exception handling; it needs to use functions that

support exception handling. C++ provides direct support to exception handling by using a try-catch block.

- **Input/Output functions**

In C, scanf and printf functions are used for input and output operations, respectively, while in C++, cin and cout are used for input and output operations, respectively.

- **Memory allocation and de-allocation**

C supports calloc() and malloc() functions for the memory allocation, and free() function for the memory de-allocation. C++ supports a new operator for the memory allocation and delete operator for the memory de-allocation.

- **Inheritance**

Inheritance is a feature that allows the child class to reuse the properties of the parent class. C language does not support the inheritance while C++ supports the inheritance.

- **Header file**

C program uses **<stdio.h>** header file while C++ program uses **<iostream.h>** header file.

Let's summarize the above differences in a tabular form.

No.	C	C++
1)	C follows the procedural style programming.	C++ is multi-paradigm. It supports both procedural and object oriented.
2)	Data is less secured in C.	In C++, you can use modifiers for class members to make it inaccessible for outside users.
3)	C follows the top-down approach.	C++ follows the bottom-up approach.
4)	C does not support function overloading.	C++ supports function overloading.
5)	In C, you can't use functions in structure.	In C++, you can use functions in structure.
6)	C does not support reference variables.	C++ supports reference variables.
7)	In C, scanf() and printf() are mainly used for input/output.	C++ mainly uses stream cin and cout to perform input and output operations.
8)	Operator overloading is not possible in C.	Operator overloading is possible in C++.
9)	C programs are divided into procedures and modules	C++ programs are divided into functions and classes.

10)	C does not provide the feature of namespace.	C++ supports the feature of namespace.
11)	Exception handling is not easy in C. It has to perform using other functions.	C++ provides exception handling using Try and Catch block.
12)	C does not support the inheritance.	C++ supports inheritance.

[← Prev](#)[Next →](#)

AD

 [For Videos Join Our YouTube Channel: Join Now](#)

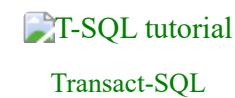
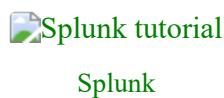
Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share



Learn Latest Tutorials



C++ history

History of C++ language is interesting to know. Here we are going to discuss brief history of C++ language.

C++ programming language was developed in 1980 by Bjarne Stroustrup at bell laboratories of AT&T (American Telephone & Telegraph), located in U.S.A.

Bjarne Stroustrup is known as the **founder of C++ language**.

It was developed for adding a feature of **OOP (Object Oriented Programming)** in C without significantly changing the C component.

C++ programming is "relative" (called a superset) of C, it means any valid C program is also a valid C++ program.

Let's see the programming languages that were developed before C++ language.



Bjarne Stroustrup

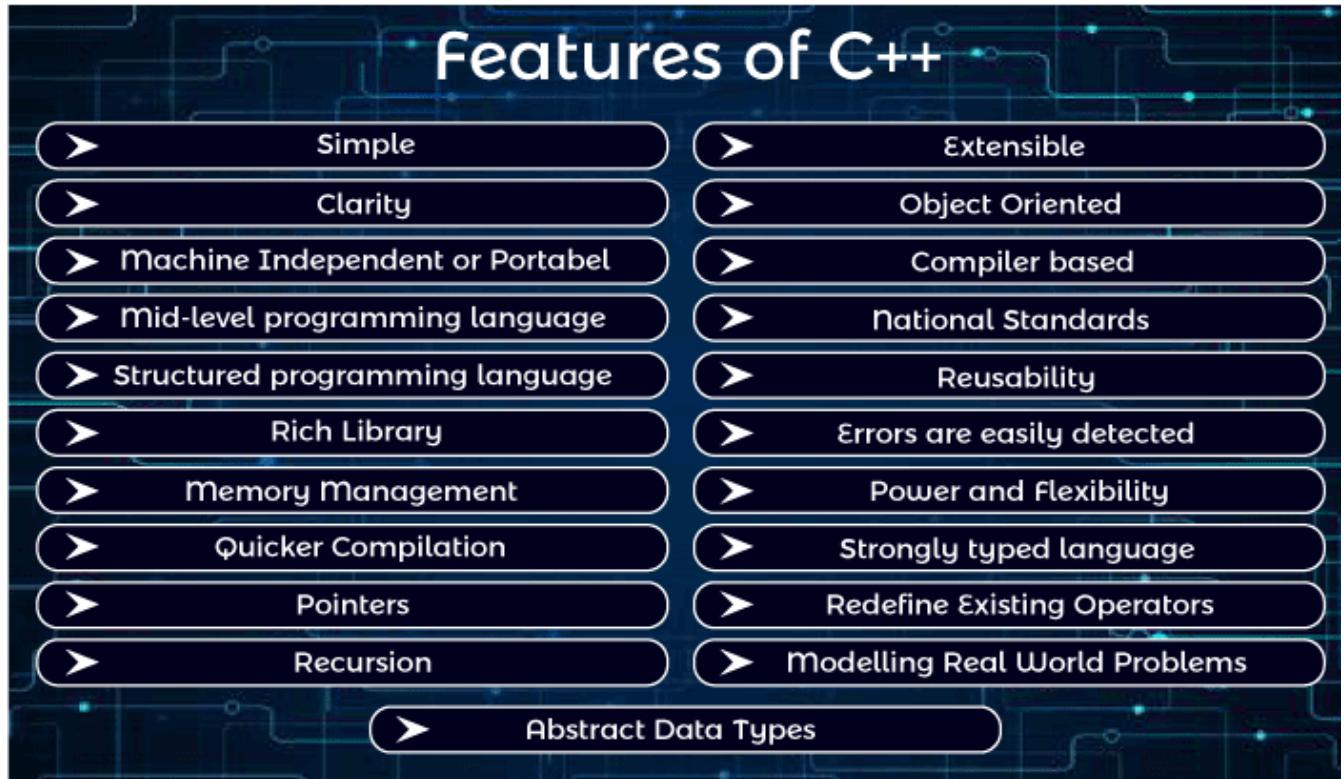
Language	Year	Developed By
Algol	1960	International Group
BCPL	1967	Martin Richard
B	1970	Ken Thompson
Traditional C	1972	Dennis Ritchie
K & R C	1978	Kernighan & Dennis Ritchie
C++	1980	Bjarne Stroustrup

← Prev

Next →

C++ Features

C++ is a widely used programming language.



It provides a lot of features that are given below.

1. Simple
2. Abstract Data types
3. Machine Independent or Portable
4. Mid-level programming language
5. Structured programming language
6. Rich Library
7. Memory Management
8. Quicker Compilation
9. Pointers
10. Recursion
11. Extensible
12. Object-Oriented
13. Compiler based
14. Reusability

15. National Standards
16. Errors are easily detected
17. Power and Flexibility
18. Strongly typed language
19. Redefine Existing Operators
20. Modeling Real-World Problems
21. Clarity

1) Simple

C++ is a simple language because it provides a structured approach (to break the problem into parts), a rich set of library functions, data types, etc.

2) Abstract Data types

In C++, complex data types called Abstract Data Types (ADT) can be created using classes.

3) Portable

C++ is a portable language and programs made in it can be run on different machines.

4) Mid-level / Intermediate programming language

C++ includes both low-level programming and high-level language so it is known as a mid-level and intermediate programming language. It is used to develop system applications such as kernel, driver, etc.

5) Structured programming language

C++ is a structured programming language. In this we can divide the program into several parts using functions.

6) Rich Library

C++ provides a lot of inbuilt functions that make the development fast. Following are the libraries used in C++ programming are:

- o <iostream>
- o <cmath>
- o <cstdlib>

- o <fstream>

7) Memory Management

C++ provides very efficient management techniques. The various memory management operators help save the memory and improve the program's efficiency. These operators allocate and deallocate memory at run time. Some common memory management operators available C++ are new, delete etc.

8) Quicker Compilation

C++ programs tend to be compact and run quickly. Hence the compilation and execution time of the C++ language is fast.

9) Pointer

C++ provides the feature of pointers. We can use pointers for memory, structures, functions, array, etc. We can directly interact with the memory by using the pointers.

10) Recursion

In C++, we can call the function within the function. It provides code reusability for every function.

11) Extensible

C++ programs can easily be extended as it is very easy to add new features into the existing program.

12) Object-Oriented

In C++, object-oriented concepts like data hiding, encapsulation, and data abstraction can easily be implemented using keyword class, private, public, and protected access specifiers. Object-oriented makes development and maintenance easier.

13) Compiler based

C++ is a compiler-based programming language, which means no C++ program can be executed without compilation. C++ compiler is easily available, and it requires very little space for storage. First, we need to compile our program using a compiler, and then we can execute our program.

14) Reusability

With the use of inheritance of functions programs written in C++ can be reused in any other program of C++. You can save program parts into library files and invoke them in your next programming projects simply by including the library files. New programs can be developed in lesser time as the existing code can be reused. It is also possible to define several functions with same name that perform different task. For Example: abs () is used to calculate the absolute value of integer, float and long integer.

15) National Standards

C++ has national standards such as ANSI.



16) Errors are easily detected

It is easier to maintain a C++ programs as errors can be easily located and rectified. It also provides a feature called exception handling to support error handling in your program.

17) Power and Flexibility

C++ is a powerful and flexible language because of most of the powerful flexible and modern UNIX operating system is written in C++. Many compilers and interpreters for other languages such as FORTRAN, PERL, Python, PASCAL, BASIC, LISP, etc., have been written in C++. C++ programs have been used for solving physics and engineering problems and even for animated special effects for movies.

18) Strongly typed language

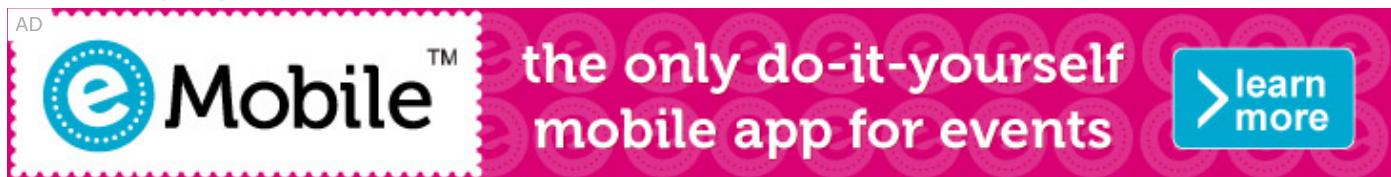
The list of arguments of every function call is typed checked during compilation. If there is a type mismatch between actual and formal arguments, implicit conversion is applied if possible. A compile-time occurs if an implicit conversion is not possible or if the number of arguments is incorrect.

19) Redefine Existing Operators

C++ allows the programmer to redefine the meaning of existing operators such as +, -. **For Example,** The "+" operator can be used for adding two numbers and concatenating two strings.

20) Modelling real-world problems

The programs written in C++ are well suited for real-world modeling problems as close as possible to the user perspective.



21) Clarity

The keywords and library functions used in C++ resemble common English words.

← Prev

Next →

AD

[For Videos Join Our YouTube Channel: Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share



C++ Program

Before starting the abcd of C++ language, you need to learn how to write, compile and run the first C++ program.

To write the first C++ program, open the C++ console and write the following code:

```
#include <iostream.h>
#include<conio.h>
void main() {
    clrscr();
    cout << "Welcome to C++ Programming.";
    getch();
}
```

#include<iostream.h> includes the **standard input output** library functions. It provides **cin** and **cout** methods for reading from input and writing to output respectively.

#include <conio.h> includes the **console input output** library functions. The **getch()** function is defined in conio.h file.

void main() The **main() function is the entry point of every program** in C++ language. The void keyword specifies that it returns no value.

cout << "Welcome to C++ Programming." is **used to print the data "Welcome to C++ Programming."** on the console.

getch() The **getch()** function **asks for a single character**. Until you press any key, it blocks the screen.

The screenshot shows a C++ IDE interface. The menu bar includes File, Edit, Search, Run, Compile, Debug, Project, Options, Window, and Help. The title bar displays "NONAME00.CPP". The code editor contains the following C++ code:

```
#include <iostream.h>
#include <conio.h>
void main(){
clrscr();
cout<<"Welcome to C++ Programming";
getch();
}
```

The status bar at the bottom shows "7:4" and various keyboard shortcuts: F1 Help, Alt-F8 Next Msg, Alt-F7 Prev Msg, Alt-F9 Compile, F9 Make, and F10 Menu.

How to compile and run the C++ program

There are 2 ways to compile and run the C++ program, by menu and by shortcut.

By menu

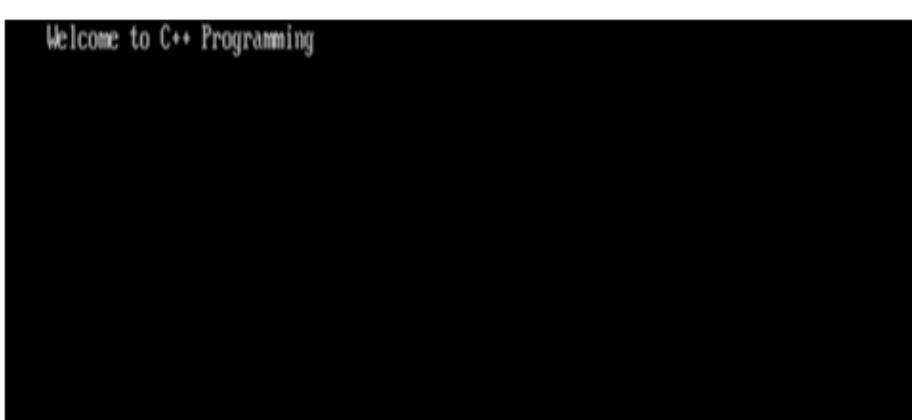
Now **click on the compile menu then compile sub menu** to compile the c++ program.

Then **click on the run menu then run sub menu** to run the c++ program.

By shortcut

Or, press **ctrl+f9** keys compile and run the program directly.

You will see the following output on user screen.



You can view the user screen any time by pressing the **alt+f5** keys.

Now **press Esc** to return to the turbo c++ console.

← Prev

Next →

AD

 [Youtube For Videos Join Our Youtube Channel: Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share



Learn Latest Tutorials

 [Splunk tutorial](#)

Splunk

 [SPSS tutorial](#)

SPSS

 [Swagger tutorial](#)

Swagger

 [T-SQL tutorial](#)

Transact-SQL

 [Tumblr tutorial](#)

Tumblr

 [React tutorial](#)

ReactJS

 [Regex tutorial](#)

Regex

 [Reinforcement learning tutorial](#)

Reinforcement Learning

C++ Basic Input/Output

C++ I/O operation is using the stream concept. Stream is the sequence of bytes or flow of data. It makes the performance fast.

If bytes flow from main memory to device like printer, display screen, or a network connection, etc, this is called as **output operation**.

If bytes flow from device like printer, display screen, or a network connection, etc to main memory, this is called as **input operation**.

I/O Library Header Files

Let us see the common header files used in C++ programming are:

Header File	Function and Description
<iostream>	It is used to define the cout, cin and cerr objects, which correspond to standard output stream, standard input stream and standard error stream, respectively.
<iomanip>	It is used to declare services useful for performing formatted I/O, such as setprecision and setw .
<fstream>	It is used to declare services for user-controlled file processing.

Standard output stream (cout)

The **cout** is a predefined object of **ostream** class. It is connected with the standard output device, which is usually a display screen. The cout is used in conjunction with stream insertion operator (<<) to display the output on a console

Let's see the simple example of standard output stream (cout):

```
#include <iostream>
using namespace std;
int main() {
    char ary[] = "Welcome to C++ tutorial";
    cout << "Value of ary is: " << ary << endl;
```

{

Output:

```
Value of ary is: Welcome to C++ tutorial
```

Standard input stream (cin)

The **cin** is a predefined object of **istream** class. It is connected with the standard input device, which is usually a keyboard. The cin is used in conjunction with stream extraction operator (>>) to read the input from a console.

Let's see the simple example of standard input stream (cin):

```
#include <iostream>
using namespace std;
int main( ) {
    int age;
    cout << "Enter your age: ";
    cin >> age;
    cout << "Your age is: " << age << endl;
}
```

Output:

```
Enter your age: 22
Your age is: 22
```

AD

Standard end line (endl)

The **endl** is a predefined object of **ostream** class. It is used to insert a new line characters and flushes the stream.

Let's see the simple example of standard end line (endl):

```
#include <iostream>
using namespace std;
int main( ) {
    cout << "C++ Tutorial";
    cout << " Javatpoint" << endl;
    cout << "End of line" << endl;
}
```

Output:

```
C++ Tutorial Javatpoint
End of line
```

← Prev

Next →

AD

 [Youtube For Videos Join Our Youtube Channel: Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

C++ Variable

A variable is a name of memory location. It is used to store data. Its value can be changed and it can be reused many times.

It is a way to represent memory location through symbol so that it can be easily identified.

Let's see the syntax to declare a variable:

```
type variable_list;
```

The example of declaring variable is given below:

```
int x;  
float y;  
char z;
```

Here, x, y, z are variables and int, float, char are data types.

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

```
int x=5,b=10; //declaring 2 variable of integer type  
float f=30.8;  
char c='A';
```

Rules for defining variables

A variable can have alphabets, digits and underscore.

A variable name can start with alphabet and underscore only. It can't start with digit.

No white space is allowed within variable name.

A variable name must not be any reserved word or keyword e.g. char, float etc.

Valid variable names:

```
int a;
```

```
int _ab;
```

```
int a30;
```

Invalid variable names:

```
int 4;
```

```
int x y;
```

```
int double;
```

← Prev

Next →

AD

 [For Videos Join Our YouTube Channel: Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share



Learn Latest Tutorials

C++ Data Types

A data type specifies the type of data that a variable can store such as integer, floating, character etc.



Data Types in C++

There are 4 types of data types in C++ language.

Types	Data Types
Basic Data Type	int, char, float, double, etc
Derived Data Type	array, pointer, etc
Enumeration Data Type	enum
User Defined Data Type	structure

Basic Data Types

The basic data types are integer-based and floating-point based. C++ language supports both signed and unsigned literals.

The memory size of basic data types may change according to 32 or 64 bit operating system.

Let's see the basic data types. Its size is given according to 32 bit OS.

Data Types	Memory Size	Range
char	1 byte	-128 to 127
signed char	1 byte	-128 to 127
unsigned char	1 byte	0 to 127
short	2 byte	-32,768 to 32,767
signed short	2 byte	-32,768 to 32,767
unsigned short	2 byte	0 to 32,767
int	2 byte	-32,768 to 32,767
signed int	2 byte	-32,768 to 32,767
unsigned int	2 byte	0 to 32,767
short int	2 byte	-32,768 to 32,767
signed short int	2 byte	-32,768 to 32,767
unsigned short int	2 byte	0 to 32,767
long int	4 byte	
signed long int	4 byte	
unsigned long int	4 byte	
float	4 byte	
double	8 byte	
long double	10 byte	

[← Prev](#)[Next →](#)

C++ Keywords

A keyword is a reserved word. You cannot use it as a variable name, constant name etc. **A list of 32 Keywords in C++ Language which are also available in C language are given below.**

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

A list of 30 Keywords in C++ Language which are not available in C language are given below.

asm	dynamic_cast	namespace	reinterpret_cast	bool
explicit	new	static_cast	false	catch
operator	template	friend	private	class
this	inline	public	throw	const_cast
delete	mutable	protected	true	try
typeid	typename	using	virtual	wchar_t

← Prev

Next →

AD

C++ Operators

An operator is simply a symbol that is used to perform operations. There can be many types of operations like arithmetic, logical, bitwise etc.

There are following types of operators to perform different types of operations in C language.

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operator
- Unary operator
- Ternary or Conditional Operator
- Misc Operator

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

Precedence of Operators in C++

The precedence of operator specifies that which operator will be evaluated first and next. The associativity specifies the operators direction to be evaluated, it may be left to right or right to left.

Let's understand the precedence by the example given below:

```
int data=5+10*10;
```

The "data" variable will contain 105 because * (multiplicative operator) is evaluated before + (additive operator).

The precedence and associativity of C++ operators is given below:

Category	Operator	Associativity
Postfix	() [] -> . ++ --	Left to right
Unary	+ - ! ~ ++ -- (type)* & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Right to left
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== != /td>	Right to left
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR		Right to left
Logical AND	&&	Left to right
Logical OR		Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= >>= <<= &= ^= =	Right to left
Comma	,	Left to right

← Prev

Next →

C++ Identifiers

C++ identifiers in a program are used to refer to the name of the variables, functions, arrays, or other user-defined data types created by the programmer. They are the basic requirement of any language. Every language has its own rules for naming the identifiers.

In short, we can say that the C++ identifiers represent the essential elements in a program which are given below:

- **Constants**
- **Variables**
- **Functions**
- **Labels**
- **Defined data types**

Some naming rules are common in both C and C++. They are as follows:

- Only alphabetic characters, digits, and underscores are allowed.
- The identifier name cannot start with a digit, i.e., the first letter should be alphabetical. After the first letter, we can use letters, digits, or underscores.
- In C++, uppercase and lowercase letters are distinct. Therefore, we can say that C++ identifiers are case-sensitive.
- A declared keyword cannot be used as a variable name.



For example, suppose we have two identifiers, named as 'FirstName', and 'Firstname'. Both the identifiers will be different as the letter 'N' in the first case is uppercase while lowercase in second. Therefore, it proves that identifiers are case-sensitive.

Valid Identifiers

The following are the examples of valid identifiers are:

Result
Test2
_sum

power

Invalid Identifiers

The following are the examples of invalid identifiers:

Sum-1 // containing special character '-'.

2data // the first letter is a digit.

break // use of a keyword.

Note: Identifiers cannot be used as the keywords. It may not conflict with the keywords, but it is highly recommended that the keywords should not be used as the identifier name. You should always use a consistent way to name the identifiers so that your code will be more readable and maintainable.

The major difference between C and C++ is the limit on the length of the name of the variable. ANSI C considers only the first 32 characters in a name while ANSI C++ imposes no limit on the length of the name.

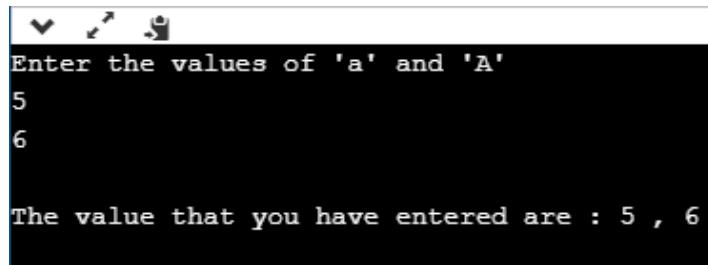
Constants are the identifiers that refer to the fixed value, which do not change during the execution of a program. Both C and C++ support various kinds of literal constants, and they do have any memory location. For example, 123, 12.34, 037, 0X2, etc. are the literal constants.

Let's look at a simple example to understand the concept of identifiers.

```
#include <iostream>
using namespace std;
int main()
{
    int a;
    int A;
    cout<<"Enter the values of 'a' and 'A'";
    cin>>a;
    cin>>A;
    cout<<"\nThe values that you have entered are : "<<a<<" , "<<A;
    return 0;
}
```

In the above code, we declare two variables 'a' and 'A'. Both the letters are same but they will behave as different identifiers. As we know that the identifiers are the case-sensitive so both the identifiers will have different memory locations.

Output



```
Enter the values of 'a' and 'A'
5
6

The value that you have entered are : 5 , 6
```

What are the keywords?

Keywords are the reserved words that have a special meaning to the compiler. They are reserved for a special purpose, which cannot be used as the identifiers. For example, 'for', 'break', 'while', 'if', 'else', etc. are the predefined words where predefined words are those words whose meaning is already known by the compiler. Whereas, the identifiers are the names which are defined by the programmer to the program elements such as variables, functions, arrays, objects, classes.

Differences between Identifiers and Keywords

The following is the list of differences between identifiers and keywords:

Identifiers	Keywords
Identifiers are the names defined by the programmer to the basic elements of a program.	Keywords are the reserved words whose meaning is known by the compiler.
It is used to identify the name of the variable.	It is used to specify the type of entity.
It can consist of letters, digits, and underscore.	It contains only letters.
It can use both lowercase and uppercase letters.	It uses only lowercase letters.
No special character can be used except the underscore.	It cannot contain any special character.
The starting letter of identifiers can be lowercase, uppercase or underscore.	It can be started only with the lowercase letter.
It can be classified as internal and external identifiers.	It cannot be further classified.

Examples are test, result, sum, power, etc.

Examples are 'for', 'if', 'else', 'break', etc.

← Prev

Next →

AD

 [For Videos Join Our Youtube Channel: Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share



Learn Latest Tutorials

 [Splunk tutorial](#)

Splunk

 [SPSS tutorial](#)

SPSS

 [Swagger tutorial](#)

Swagger

 [T-SQL tutorial](#)

Transact-SQL

 [Tumblr tutorial](#)

Tumblr

 [React tutorial](#)

ReactJS

 [Regex tutorial](#)

Regex

 [Reinforcement learning tutorial](#)

C++ Expression

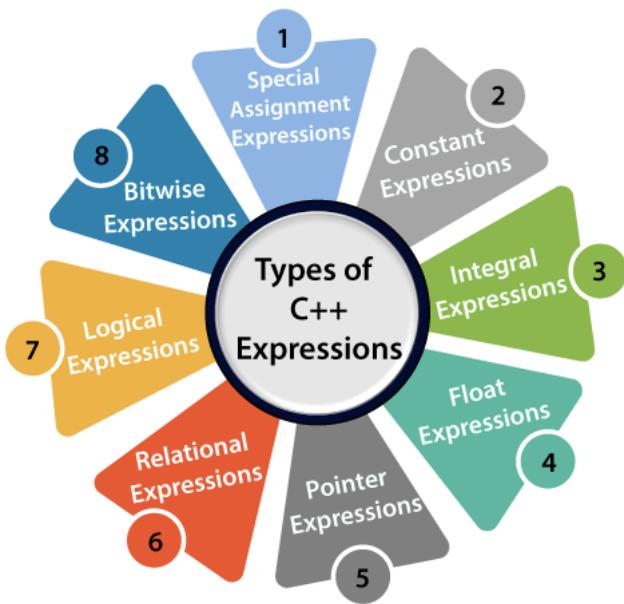
C++ expression consists of operators, constants, and variables which are arranged according to the rules of the language. It can also contain function calls which return values. An expression can consist of one or more operands, zero or more operators to compute a value. Every expression produces some value which is assigned to the variable with the help of an assignment operator.

Examples of C++ expression:

```
(a+b) - c
(x/y) -z
4a2 - 5b +c
(a+b) * (x+y)
```

An expression can be of following types:

- Constant expressions
- Integral expressions
- Float expressions
- Pointer expressions
- Relational expressions
- Logical expressions
- Bitwise expressions
- Special assignment expressions



If the expression is a combination of the above expressions, such expressions are known as compound expressions.

Constant expressions

A constant expression is an expression that consists of only constant values. It is an expression whose value is determined at the compile-time but evaluated at the run-time. It can be composed of integer, character, floating-point, and enumeration constants.

Constants are used in the following situations:

- It is used in the subscript declarator to describe the array bound.
- It is used after the case keyword in the switch statement.
- It is used as a numeric value in an **enum**

- o It specifies a bit-field width.
- o It is used in the pre-processor **#if**



In the above scenarios, the constant expression can have integer, character, and enumeration constants. We can use the static and extern keyword with the constants to define the function-scope.

The following table shows the expression containing constant value:

Expression containing constant	Constant value
<code>x = (2/3) * 4</code>	<code>(2/3) * 4</code>
<code>extern int y = 67</code>	67
<code>int z = 43</code>	43
<code>static int a = 56</code>	56

Let's see a simple program containing constant expression:

```
#include <iostream>
using namespace std;
int main()
{
    int x;      // variable declaration.
    x=(3/2) + 2; // constant expression
    cout<<"Value of x is : "<<x; // displaying the value of x.
    return 0;
}
```

In the above code, we have first declared the 'x' variable of integer type. After declaration, we assign the simple constant expression to the 'x' variable.

Output

```
Value of x is : 3
```

Integral Expressions

An integer expression is an expression that produces the integer value as output after performing all the explicit and implicit conversions.

Following are the examples of integral expression:

```
(x * y) -5
x + int(9.0)
where x and y are the integers.
```

Let's see a simple example of integral expression:

```
#include <iostream>
using namespace std;
int main()
{
```

```

int x; // variable declaration.
int y; // variable declaration
int z; // variable declaration
cout<<"Enter the values of x and y";
cin>>x>>y;
z=x+y;
cout<<"\n"<<"Value of z is :"<<z; // displaying the value of z.
return 0;
}

```

In the above code, we have declared three variables, i.e., x, y, and z. After declaration, we take the user input for the values of 'x' and 'y'. Then, we add the values of 'x' and 'y' and stores their result in 'z' variable.

Output

```

Enter the values of x and y
8
9
Value of z is :17

```

Let's see another example of integral expression.

```

#include <iostream>
using namespace std;
int main()
{
    int x; // variable declaration
    int y=9; // variable initialization
    x=y+int(10.0); // integral expression
    cout<<"Value of x : "<<x; // displaying the value of x.
    return 0;
}

```

In the above code, we declare two variables, i.e., x and y. We store the value of expression ($y+int(10.0)$) in a 'x' variable.



Output

```

Value of x : 19

```

Float Expressions

A float expression is an expression that produces floating-point value as output after performing all the explicit and implicit conversions.

The following are the examples of float expressions:

```

x+y
(x/10) + y
34.5
x+float(10)

```

Let's understand through an example.

```
#include <iostream>
using namespace std;
int main()
{
    float x=8.9; // variable initialization
    float y=5.6; // variable initialization
    float z; // variable declaration
    z=x+y;
    std::cout << "value of z is :" << z << std::endl; // displaying the value of z.

    return 0;
}
```

Output



```
value of z is :14.5
```

Let's see another example of float expression.

```
#include <iostream>
using namespace std;
int main()
{
    float x=6.7; // variable initialization
    float y; // variable declaration
    y=x+float(10); // float expression
    std::cout << "value of y is :" << y << std::endl; // displaying the value of y
    return 0;
}
```

In the above code, we have declared two variables, i.e., x and y. After declaration, we store the value of expression ($x+float(10)$) in variable 'y'.

Output

```
value of y is :16.7
```

Pointer Expressions

A pointer expression is an expression that produces address value as an output.

The following are the examples of pointer expression:

```
&x
ptr
ptr++
```

ptr-

Let's understand through an example.

```
#include <iostream>
using namespace std;
int main()
{
    int a[]={1,2,3,4,5}; // array initialization
    int *ptr; // pointer declaration
    ptr=a; // assigning base address of array to the pointer ptr
    ptr=ptr+1; // incrementing the value of pointer
    std::cout << "value of second element of an array : " << *ptr << std::endl;
    return 0;
}
```

In the above code, we declare the array and a pointer ptr. We assign the base address to the variable 'ptr'. After assigning the address, we increment the value of pointer 'ptr'. When pointer is incremented then 'ptr' will be pointing to the second element of the array.

Output

```
value of second element of an array : 2
```

Relational Expressions

A relational expression is an expression that produces a value of type bool, which can be either true or false. It is also known as a boolean expression. When arithmetic expressions are used on both sides of the relational operator, arithmetic expressions are evaluated first, and then their results are compared.

The following are the examples of the relational expression:

```
a>b
a-b >= x-y
a+b>80
```

Let's understand through an example

```
#include <iostream>
using namespace std;
int main()
{
    int a=45; // variable declaration
    int b=78; // variable declaration
    bool y= a>b; // relational expression
    cout << "Value of y is :" << y; // displaying the value of y.
    return 0;
}
```

In the above code, we have declared two variables, i.e., 'a' and 'b'. After declaration, we have applied the relational operator between the variables to check whether 'a' is greater than 'b' or not.

Output

```
Value of y is :0
```

Let's see another example.

```
#include <iostream>
using namespace std;
int main()
{
    int a=4; // variable declaration
    int b=5; // variable declaration
    int x=3; // variable declaration
    int y=6; // variable declaration
    cout<<((a+b)>=(x+y)); // relational expression
    return 0;
}
```

In the above code, we have declared four variables, i.e., 'a', 'b', 'x' and 'y'. Then, we apply the relational operator (\geq) between these variables.

Output

```
1
```

Logical Expressions

A logical expression is an expression that combines two or more relational expressions and produces a bool type value. The logical operators are ' $\&\&$ ' and ' $\|$ ' that combines two or more relational expressions.

The following are some examples of logical expressions:

```
a>b && x>y
a>10 || b==5
```

Let's see a simple example of logical expression.

```
#include <iostream>
using namespace std;
int main()
{
    int a=2;
    int b=7;
    int c=4;
    cout<<((a>b)|(a>c));
    return 0;
}
```

Output

```
0
```

Bitwise Expressions

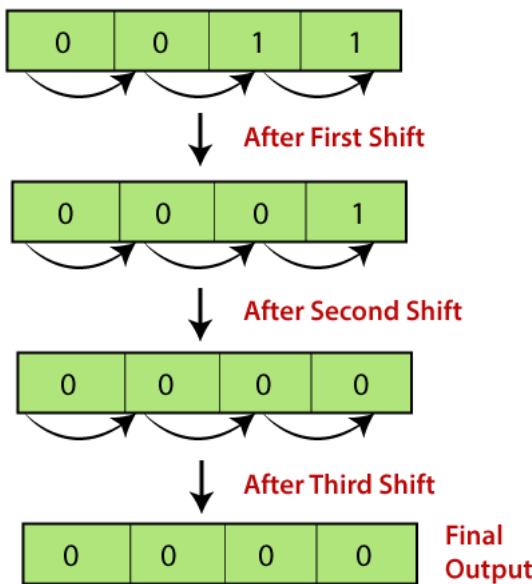
A bitwise expression is an expression which is used to manipulate the data at a bit level. They are basically used to shift the bits.

For example:

x=3

`x>>3` // This statement means that we are shifting the three-bit position to the right.

In the above example, the value of 'x' is 3 and its binary value is 0011. We are shifting the value of 'x' by three-bit position to the right. Let's understand through the diagrammatic representation.



Let's see a simple example.

```
#include <iostream>
using namespace std;
int main()
{
    int x=5; // variable declaration
    std::cout << (x>>1) << std::endl;
    return 0;
}
```

In the above code, we have declared a variable 'x'. After declaration, we applied the bitwise operator, i.e., right shift operator to shift one-bit position to right.

Output

2

Let's look at another example.

```
#include <iostream>
using namespace std;
int main()
{
    int x=7; // variable declaration
    std::cout << (x<<3) << std::endl;
    return 0;
}
```

}

In the above code, we have declared a variable 'x'. After declaration, we applied the left shift operator to variable 'x' to shift the three-bit position to the left.

Output

56

Special Assignment Expressions

Special assignment expressions are the expressions which can be further classified depending upon the value assigned to the variable.

- o **Chained Assignment**

Chained assignment expression is an expression in which the same value is assigned to more than one variable by using single statement.

For example:

```
a=b=20
or
(a=b) = 20
```

Let's understand through an example.

```
#include <iostream>
using namespace std;
int main()

int a; // variable declaration
int b; // variable declaration
a=b=80; // chained assignment
std::cout << "Values of 'a' and 'b' are : " <<a<<","<<b<< std::endl;
return 0;
}
```

In the above code, we have declared two variables, i.e., 'a' and 'b'. Then, we have assigned the same value to both the variables using chained assignment expression.

Output

Values of 'a' and 'b' are : 80,80

Note: Using chained assignment expression, the value cannot be assigned to the variable at the time of declaration. For example, int a=b=c=90 is an invalid statement.

- o **Embedded Assignment Expression**

An embedded assignment expression is an assignment expression in which assignment expression is enclosed within another assignment expression.

Let's understand through an example.

```
#include <iostream>
```

```
using namespace std;
int main()
{
    int a; // variable declaration
    int b; // variable declaration
    a=10+(b=90); // embedded assignment expression
    std::cout << "Values of 'a' is " << a << std::endl;
    return 0;
}
```

In the above code, we have declared two variables, i.e., 'a' and 'b'. Then, we applied embedded assignment expression ($a=10+(b=90)$).

Output

```
Values of 'a' is 100
```

- o **Compound Assignment**

A compound assignment expression is an expression which is a combination of an assignment operator and binary operator.

For example,

```
a+=10;
```

In the above statement, 'a' is a variable and ' $+=$ ' is a compound statement.

Let's understand through an example.

```
#include <iostream>
using namespace std;
int main()
{
    int a=10; // variable declaration
    a+=10; // compound assignment
    std::cout << "Value of a is :" << a << std::endl; // displaying the value of a.
    return 0;
}
```

In the above code, we have declared a variable 'a' and assigns 10 value to this variable. Then, we applied compound assignment operator ($+=$) to 'a' variable, i.e., $a+=10$ which is equal to $(a=a+10)$. This statement increments the value of 'a' by 10.

Output

```
Value of a is :20
```

C++ if-else

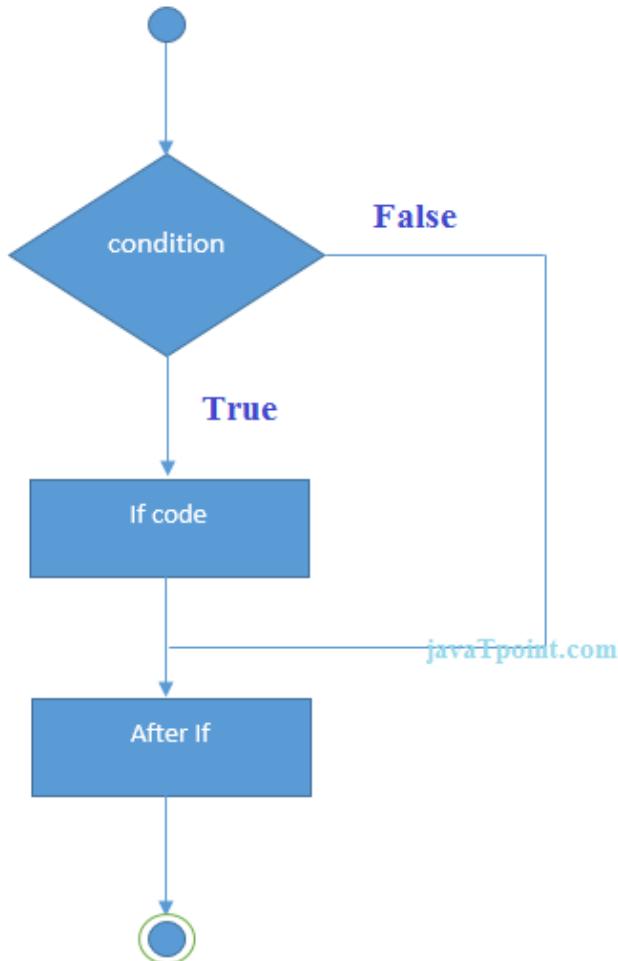
In C++ programming, if statement is used to test the condition. There are various types of if statements in C++.

- if statement
- if-else statement
- nested if statement
- if-else-if ladder

C++ IF Statement

The C++ if statement tests the condition. It is executed if condition is true.

```
if(condition){  
    //code to be executed  
}
```



C++ If Example

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

int main () {
    int num = 10;
    if (num % 2 == 0)
    {
        cout<<"It is even number";
    }
    return 0;
}
```

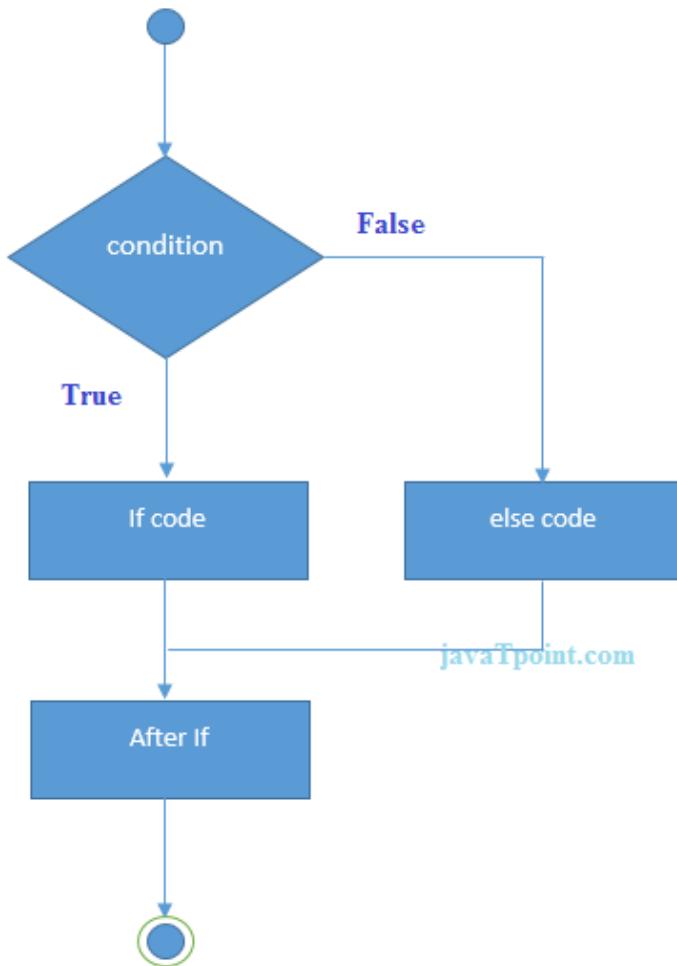
Output:/p>

```
It is even number
```

C++ IF-else Statement

The C++ if-else statement also tests the condition. It executes if block if condition is true otherwise else block is executed.

```
if(condition){
    //code if condition is true
}else{
    //code if condition is false
}
```



AD



VIDEO STREAMING FOR THE AWAKENED MIND

GET STARTED NOW

C++ If-else Example

```
#include <iostream>
using namespace std;
int main () {
    int num = 11;
    if (num % 2 == 0)
    {
        cout<<"It is even number";
    }
    else
    {
        cout<<"It is odd number";
    }
    return 0;
}
```

```
}
```

Output:

```
It is odd number
```

C++ If-else Example: with input from user

```
#include <iostream>
using namespace std;
int main () {
    int num;
    cout<<"Enter a Number: ";
    cin>>num;
    if (num % 2 == 0)
    {
        cout<<"It is even number"<<endl;
    }
    else
    {
        cout<<"It is odd number"<<endl;
    }
    return 0;
}
```

Output:

```
Enter a number:11
It is odd number
```

Output:

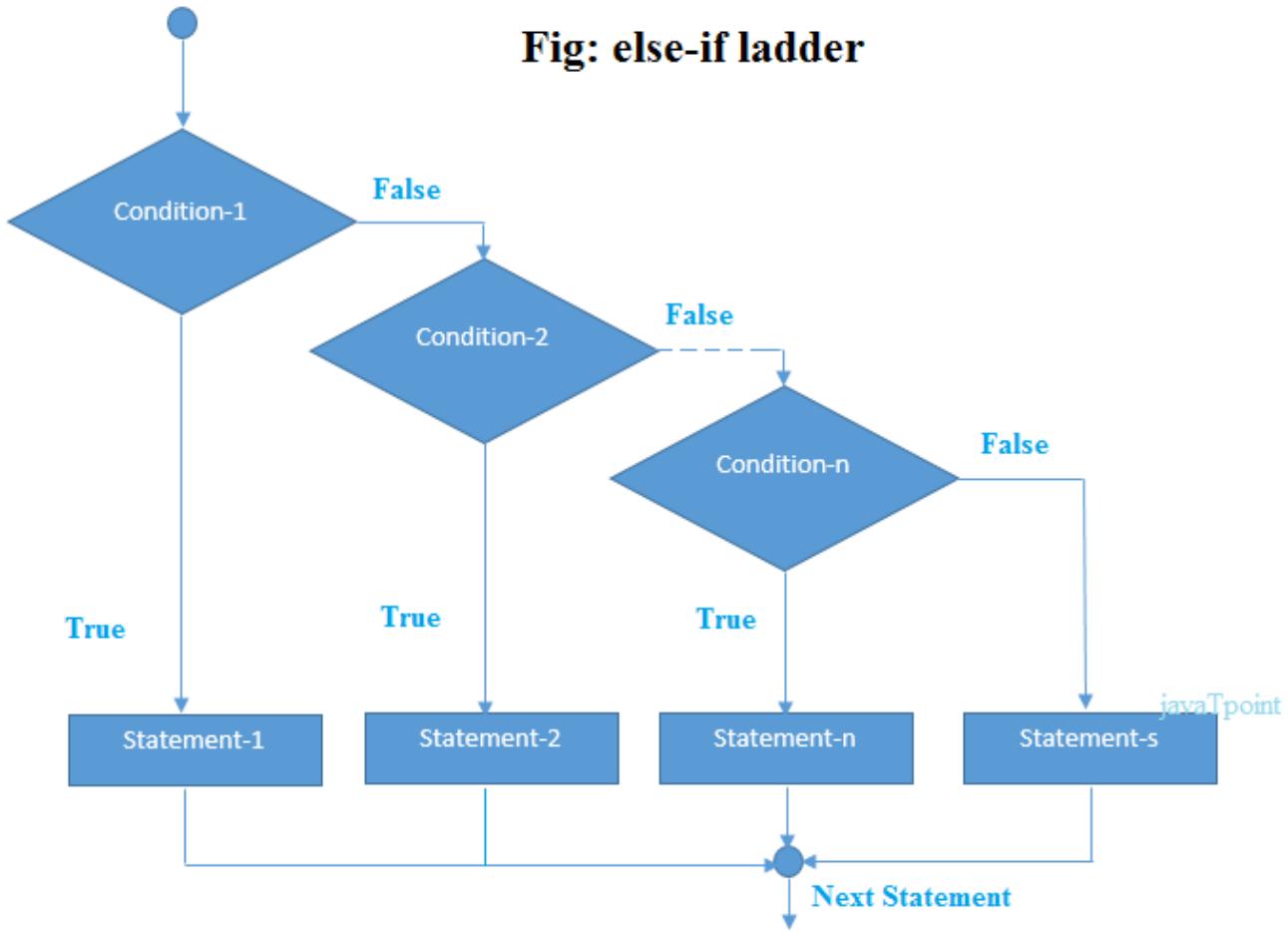
```
Enter a number:12
It is even number
```

C++ IF-else-if ladder Statement

The C++ if-else-if ladder statement executes one condition from multiple statements.

```
if(condition1){  
    //code to be executed if condition1 is true  
}  
else if(condition2){  
    //code to be executed if condition2 is true  
}  
}  
  
else if(condition3){  
    //code to be executed if condition3 is true  
}  
}  
  
...  
  
else{  
    //code to be executed if all the conditions are false  
}
```

Fig: else-if ladder



C++ If else-if Example

```
#include <iostream>
using namespace std;
int main () {
    int num;
    cout<<"Enter a number to check grade:";
    cin>>num;
    if (num <0 || num >100)
    {
        cout<<"wrong number";
    }
    else if(num >= 0 && num < 50){
        cout<<"Fail";
    }
    else if (num >= 50 && num < 60)
    {
        cout<<"D Grade";
    }
    else if (num >= 60 && num < 70)
    {
        cout<<"C Grade";
    }
    else if (num >= 70 && num < 80)
    {
        cout<<"B Grade";
    }
    else if (num >= 80 && num < 90)
    {
        cout<<"A Grade";
    }
    else if (num >= 90 && num <= 100)
    {
        cout<<"A+ Grade";
    }
}
```

Output:

```
Enter a number to check grade:66
```

```
C Grade
```

Output:

```
Enter a number to check grade:-2
```

```
wrong number
```

← Prev

Next →

AD

 [For Videos Join Our Youtube Channel: Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share

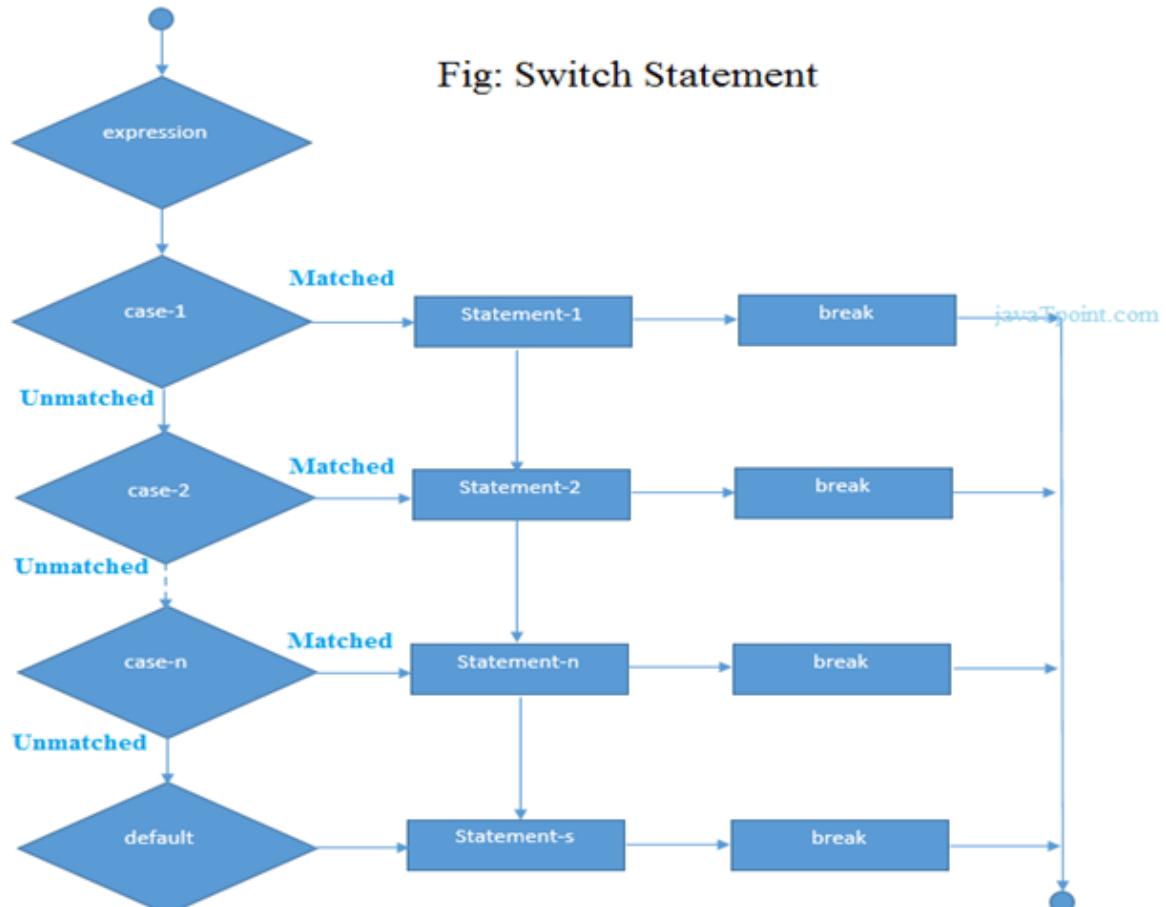


C++ switch

The C++ switch statement executes one statement from multiple conditions. It is like if-else-if ladder statement in C++.

```
switch(expression){
    case value1:
        //code to be executed;
        break;
    case value2:
        //code to be executed;
        break;
    .....
    default:
        //code to be executed if all cases are not matched;
        break;
}
```

Fig: Switch Statement



C++ Switch Example

```
#include <iostream>
using namespace std;
int main () {
    int num;
    cout<<"Enter a number to check grade:";
    cin>>num;
    switch (num)
    {
        case 10: cout<<"It is 10"; break;
        case 20: cout<<"It is 20"; break;
        case 30: cout<<"It is 30"; break;
        default: cout<<"Not 10, 20 or 30"; break;
    }
}
```

Output:

```
Enter a number:
10
It is 10
```

Output:

```
Enter a number:
55
Not 10, 20 or 30
```

← Prev

Next →

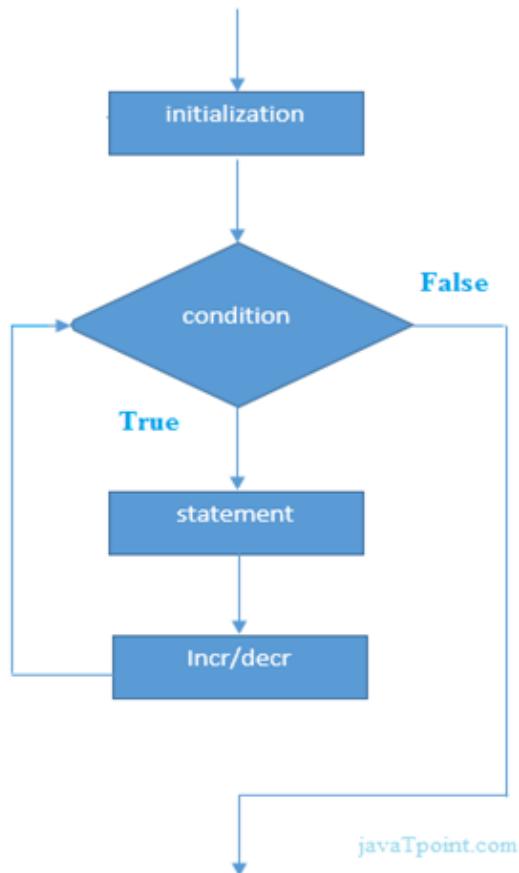
C++ For Loop

The C++ for loop is used to iterate a part of the program several times. If the number of iteration is fixed, it is recommended to use for loop than while or do-while loops.

The C++ for loop is same as C/C#. We can initialize variable, check condition and increment/decrement value.

```
for(initialization; condition; incr/decr){  
    //code to be executed  
}
```

Flowchart:



C++ For Loop Example

```
#include <iostream>  
using namespace std;  
int main() {
```

```
for(int i=1;i<=10;i++){
    cout<<i <<"\n";
}
```

Output:

```
1
2
3
4
5
6
7
8
9
10
```

C++ Nested For Loop

In C++, we can use for loop inside another for loop, it is known as nested for loop. The inner loop is executed fully when outer loop is executed one time. So if outer loop and inner loop are executed 4 times, inner loop will be executed 4 times for each outer loop i.e. total 16 times.

C++ Nested For Loop Example

Let's see a simple example of nested for loop in C++.

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

int main () {
    for(int i=1;i<=3;i++){
        for(int j=1;j<=3;j++){
            cout<<i << " " <<j <<"\n";
        }
    }
}
```

{

Output:

```
1 1
1 2
1 3
2 1
2 2
2 3
3 1
3 2
3 3
```

AD

C++ Infinite For Loop

If we use double semicolon in for loop, it will be executed infinite times. Let's see a simple example of infinite for loop in C++.

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

int main () {
    for (; ;)
    {
        cout<<"Infinitive For Loop";
    }
}
```

Output:

```
Infinitive For Loop
Infinitive For Loop
```

Infinitive For Loop
Infinitive For Loop
Infinitive For Loop
ctrl+c

← Prev

Next →

AD

 [YouTube For Videos Join Our YouTube Channel: Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share



Learn Latest Tutorials

 [Splunk tutorial](#)

Splunk

 [SPSS tutorial](#)

SPSS

 [Swagger tutorial](#)

Swagger

 [T-SQL tutorial](#)

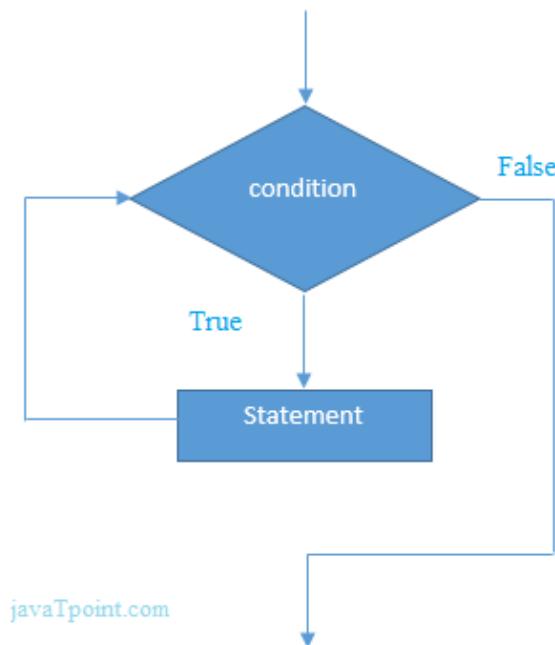
Transact-SQL

C++ While loop

In C++, while loop is used to iterate a part of the program several times. If the number of iteration is not fixed, it is recommended to use while loop than for loop.

```
while(condition){  
    //code to be executed  
}
```

Flowchart:



C++ While Loop Example

Let's see a simple example of while loop to print table of 1.

```
#include <iostream>  
using namespace std;  
int main() {  
    int i=1;  
    while(i<=10)  
    {  
        cout<<i << "\n";  
        i++;  
    }  
}
```

```
    }  
}
```

Output:

```
1  
2  
3  
4  
5  
6  
7  
8  
9  
10
```

C++ Nested While Loop Example

In C++, we can use while loop inside another while loop, it is known as nested while loop. The nested while loop is executed fully when outer loop is executed once.

Let's see a simple example of nested while loop in C++ programming language.

```
#include <iostream>  
using namespace std;  
int main () {  
    int i=1;  
    while(i<=3)  
    {  
        int j = 1;  
        while (j <= 3)  
        {  
            cout<<i<<" "<<j<<"\n";  
            j++;  
        }  
        i++;  
    }
```

```
}
```

Output:

```
1 1
1 2
1 3
2 1
2 2
2 3
3 1
3 2
3 3
```

C++ Infinitive While Loop Example:

We can also create infinite while loop by passing true as the test condition.

```
#include <iostream>
using namespace std;
int main () {
    while(true)
    {
        cout<<"Infinitive While Loop";
    }
}
```

Output:

```
Infinitive While Loop
ctrl+c
```

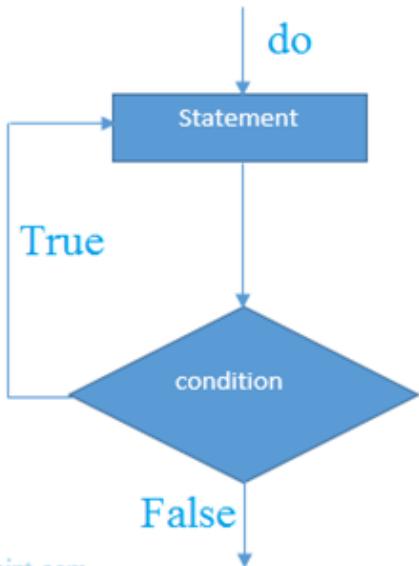
C++ Do-While Loop

The C++ do-while loop is used to iterate a part of the program several times. If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use do-while loop.

The C++ do-while loop is executed at least once because condition is checked after loop body.

```
do{  
    //code to be executed  
}while(condition);
```

Flowchart:



javaTpoint.com

C++ do-while Loop Example

Let's see a simple example of C++ do-while loop to print the table of 1.

```
#include <iostream>  
using namespace std;  
int main() {  
    int i = 1;  
    do{  
        cout<<i<<"\n";  
        i++;  
    }
```

```
    } while (i <= 10);  
}
```

Output:

```
1  
2  
3  
4  
5  
6  
7  
8  
9  
10
```

C++ Nested do-while Loop

In C++, if you use do-while loop inside another do-while loop, it is known as nested do-while loop. The nested do-while loop is executed fully for each outer do-while loop.

Let's see a simple example of nested do-while loop in C++.

```
#include <iostream>  
using namespace std;  
int main() {  
    int i = 1;  
    do{  
        int j = 1;  
        do{  
            cout<<i<<"\n";  
            j++;  
        } while (j <= 3);  
        i++;  
    } while (i <= 3);  
}
```

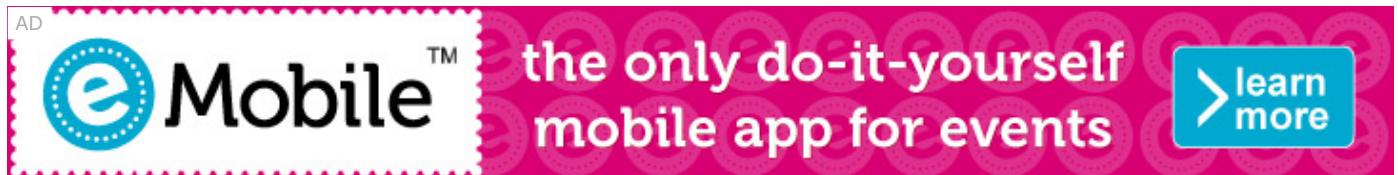
Output:

```
1 1  
1 2  
1 3  
2 1  
2 2  
2 3  
3 1  
3 2  
3 3
```

C++ Infinitive do-while Loop

In C++, if you pass **true** in the do-while loop, it will be infinitive do-while loop.

```
do{  
    //code to be executed  
}while(true);
```



C++ Infinitive do-while Loop Example

```
#include <iostream>  
using namespace std;  
int main() {  
    do{  
        cout<<"Infinitive do-while Loop";  
    } while(true);  
}
```

Output:

```
Infinitive do-while Loop  
Infinitive do-while Loop
```

```
Infinitive do-while Loop
Infinitive do-while Loop
Infinitive do-while Loop
ctrl+c
```

← Prev

Next →

AD

 [For Videos Join Our YouTube Channel: Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share



Learn Latest Tutorials

 [Splunk tutorial](#)

Splunk

 [SPSS tutorial](#)

SPSS

 [Swagger tutorial](#)

Swagger

 [T-SQL tutorial](#)

Transact-SQL

C++ Break Statement

The C++ break is used to break loop or switch statement. It breaks the current flow of the program at the given condition. In case of inner loop, it breaks only inner loop.

```
jump-statement;  
break;
```

Flowchart:

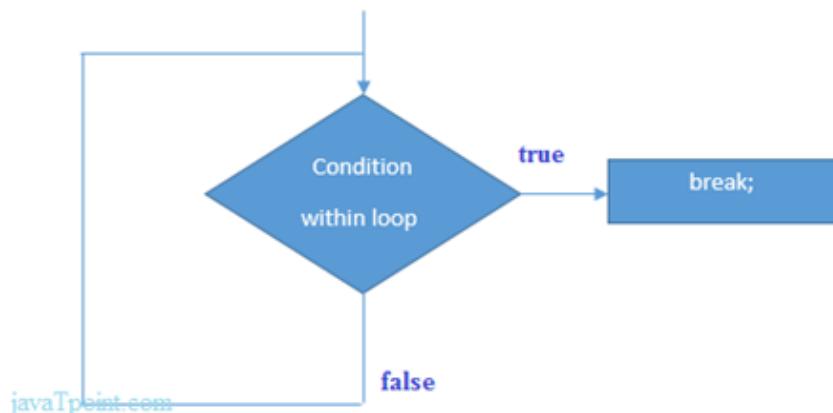


Figure: Flowchart of break statement

C++ Break Statement Example

Let's see a simple example of C++ break statement which is used inside the loop.

```
#include <iostream>  
using namespace std;  
int main() {  
    for (int i = 1; i <= 10; i++)  
    {  
        if (i == 5)  
        {  
            break;  
        }  
        cout<<i<<"\n";  
    }  
}
```

```
}
```

Output:

```
1  
2  
3  
4
```

C++ Break Statement with Inner Loop

The C++ break statement breaks inner loop only if you use break statement inside the inner loop.

Let's see the example code:

```
#include <iostream>  
using namespace std;  
int main()  
{  
    for(int i=1;i<=3;i++){  
        for(int j=1;j<=3;j++){  
            if(i==2&&j==2){  
                break;  
            }  
            cout<<i<<" "<<j<<"\n";  
        }  
    }  
}
```

Output:

```
1 1  
1 2  
1 3  
2 1  
3 1
```

3 2

3 3

[← Prev](#)[Next →](#)

AD

 For Videos Join Our YouTube Channel: [Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share



Learn Latest Tutorials

 Splunk tutorial

Splunk

 SPSS tutorial

SPSS

 Swagger tutorial

Swagger

 T-SQL tutorial

Transact-SQL

 Tumblr tutorial

Tumblr

 React tutorial

ReactJS

 Regex tutorial

Regex

 Reinforcement learning tutorial

C++ Continue Statement

The C++ continue statement is used to continue loop. It continues the current flow of the program and skips the remaining code at specified condition. In case of inner loop, it continues only inner loop.

```
jump-statement;  
continue;
```

C++ Continue Statement Example

```
#include <iostream>  
using namespace std;  
int main()  
{  
    for(int i=1;i<=10;i++){  
        if(i==5){  
            continue;  
        }  
        cout<<i<<"\n";  
    }  
}
```

Output:

```
1  
2  
3  
4  
6  
7  
8  
9  
10
```

C++ Continue Statement with Inner Loop

C++ Continue Statement continues inner loop only if you use continue statement inside the inner loop.

```
#include <iostream>
using namespace std;
int main()
{
    for(int i=1;i<=3;i++){
        for(int j=1;j<=3;j++){
            if(i==2&&j==2){
                continue;
            }
            cout<<i<<" "<<j<<"\n";
        }
    }
}
```

Output:

```
1 1
1 2
1 3
2 1
2 3
3 1
3 2
3 3
```

← Prev

Next →

C++ Goto Statement

The C++ goto statement is also known as jump statement. It is used to transfer control to the other part of the program. It unconditionally jumps to the specified label.

It can be used to transfer control from deeply nested loop or switch case label.

C++ Goto Statement Example

Let's see the simple example of goto statement in C++.

```
#include <iostream>
using namespace std;
int main()
{
ineligible:
    cout<<"You are not eligible to vote!\n";
    cout<<"Enter your age:\n";
    int age;
    cin>>age;
    if (age < 18){
        goto ineligible;
    }
    else
    {
        cout<<"You are eligible to vote!";
    }
}
```

Output:

```
You are not eligible to vote!
```

```
Enter your age:
```

```
16
```

```
You are not eligible to vote!
```

```
Enter your age:
```

```
7
```

```
You are not eligible to vote!
```

```
Enter your age:
```

```
22
```

```
You are eligible to vote!
```

← Prev

Next →

AD

 [For Videos Join Our Youtube Channel: Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share



Learn Latest Tutorials

 [Splunk tutorial](#)

Splunk

 [SPSS tutorial](#)

SPSS

 [Swagger tutorial](#)

Swagger

 [T-SQL tutorial](#)

Transact-SQL

C++ Comments

The C++ comments are statements that are not executed by the compiler. The comments in C++ programming can be used to provide explanation of the code, variable, method or class. By the help of comments, you can hide the program code also.

There are two types of comments in C++.

- Single Line comment
- Multi Line comment

C++ Single Line Comment

The single line comment starts with // (double slash). Let's see an example of single line comment in C++.

```
#include <iostream>
using namespace std;
int main()
{
    int x = 11; // x is a variable
    cout<<x<<"\n";
}
```

Output:

```
11
```

C++ Multi Line Comment

The C++ multi line comment is used to comment multiple lines of code. It is surrounded by slash and asterisk /* */. Let's see an example of multi line comment in C++.

```
#include <iostream>
using namespace std;
int main()
```

```
{  
/* declare and  
print variable in C++. */  
int x = 35;  
cout<<x<<"\n";  
}
```

Output:

```
35
```

← Prev

Next →

AD

 [For Videos Join Our YouTube Channel: Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share



C++ Functions

The function in C++ language is also known as procedure or subroutine in other programming languages.

To perform any task, we can create function. A function can be called many times. It provides modularity and code reusability.

Advantage of functions in C

There are many advantages of functions.

1) Code Reusability

By creating functions in C++, you can call it many times. So we don't need to write the same code again and again.

2) Code optimization

It makes the code optimized, we don't need to write much code.

Suppose, you have to check 3 numbers (531, 883 and 781) whether it is prime number or not. Without using function, you need to write the prime number logic 3 times. So, there is repetition of code.

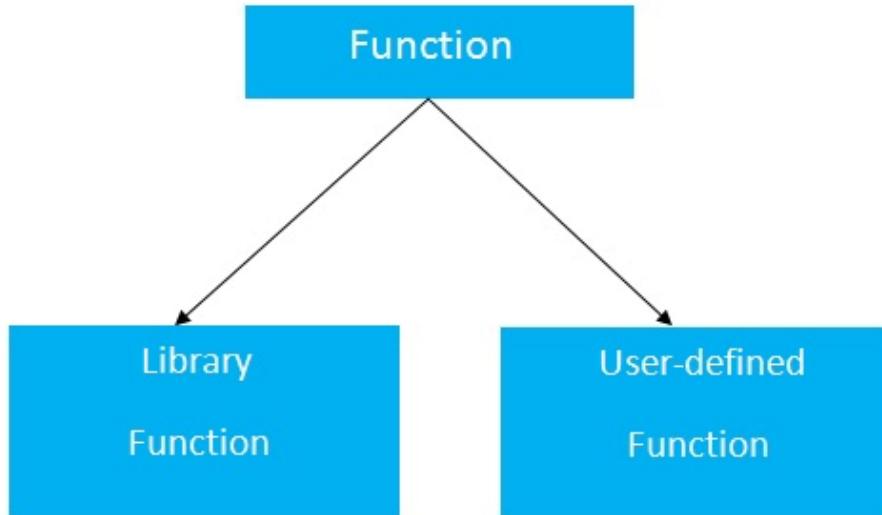
But if you use functions, you need to write the logic only once and you can reuse it several times.

Types of Functions

There are two types of functions in C programming:

1. Library Functions: are the functions which are declared in the C++ header files such as `ceil(x)`, `cos(x)`, `exp(x)`, etc.

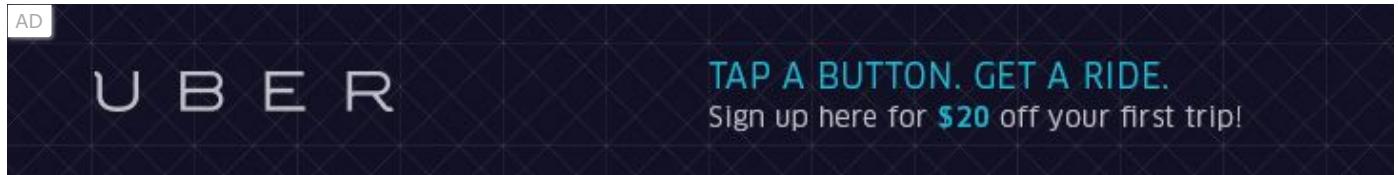
2. User-defined functions: are the functions which are created by the C++ programmer, so that he/she can use it many times. It reduces complexity of a big program and optimizes the code.



Declaration of a function

The syntax of creating function in C++ language is given below:

```
return_type function_name(data_type parameter...)  
{  
    //code to be executed  
}
```



C++ Function Example

Let's see the simple example of C++ function.

```
#include <iostream>  
using namespace std;  
void func() {  
    static int i=0; //static variable  
    int j=0; //local variable  
    i++;  
    j++;  
    cout<<"i=" << i << " and j=" << j << endl;  
}
```

```
int main()
{
    func();
    func();
    func();
}
```

Output:

```
i= 1 and j= 1
i= 2 and j= 1
i= 3 and j= 1
```

← Prev

Next →

AD

 [For Videos Join Our Youtube Channel: Join Now](#)

Feedback

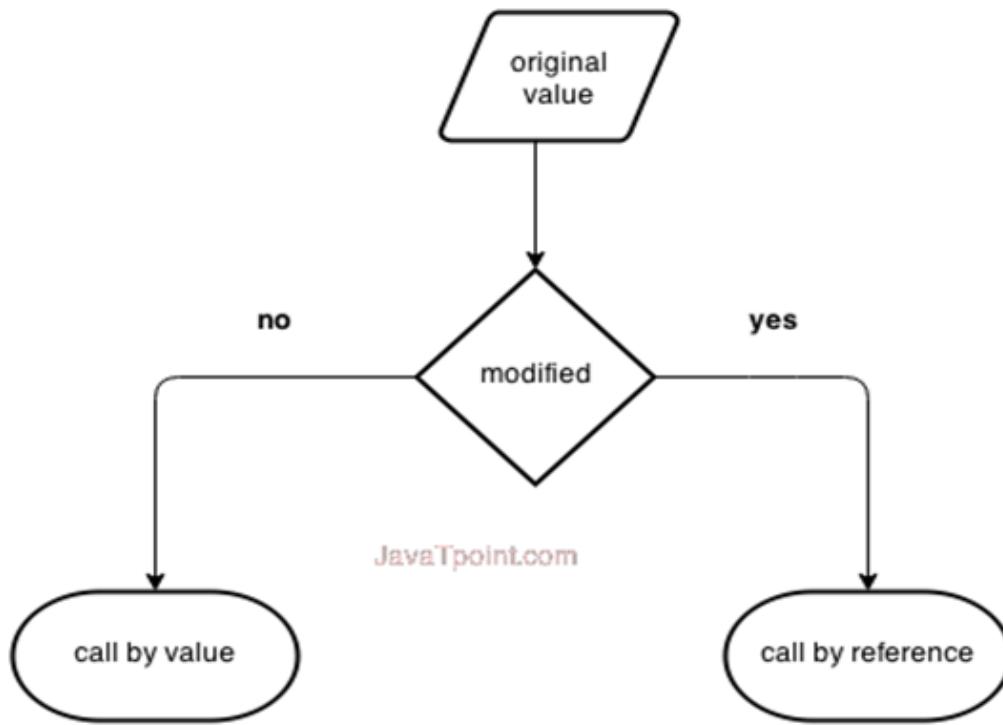
- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share



Call by value and call by reference in C++

There are two ways to pass value or data to function in C language: call by value and call by reference. Original value is not modified in call by value but it is modified in call by reference.



Let's understand call by value and call by reference in C++ language one by one.

Call by value in C++

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

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

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

```

#include <iostream>
using namespace std;
void change(int data);
int main()
{
    int data = 3;
    change(data);
}
    
```

```
cout << "Value of the data is: " << data << endl;  
return 0;  
}  
  
void change(int data)  
{  
data = 5;  
}
```

Output:

```
Value of the data is: 3
```

Call by reference in C++

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

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

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

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

```
#include<iostream>  
using namespace std;  
void swap(int *x, int *y)  
{  
    int swap;  
    swap=*x;  
    *x=*y;  
    *y=swap;  
}  
  
int main()  
{  
    int x=500, y=100;  
    swap(&x, &y); // passing value to function
```

```
cout<<"Value of x is: "<<x<<endl;
cout<<"Value of y is: "<<y<<endl;
return 0;
}
```

Output:

```
Value of x is: 100
Value of y is: 500
```

Difference between call by value and call by reference in C++

No.	Call by value	Call by reference
1	A copy of value is passed to the function	An address of value is passed to the function
2	Changes made inside the function is not reflected on other functions	Changes made inside the function is reflected outside the function also
3	Actual and formal arguments will be created in different memory location	Actual and formal arguments will be created in same memory location

← Prev

Next →

AD

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.

Let's see a simple example of recursion.

```
recursionfunction(){  
recursionfunction(); //calling self function  
}  
}
```

C++ Recursion Example

Let's see an example to print factorial number using recursion in C++ language.

```
#include<iostream>  
using namespace std;  
int main()  
{  
    int factorial(int);  
    int fact,value;  
    cout<<"Enter any number: ";  
    cin>>value;  
    fact=factorial(value);  
    cout<<"Factorial of a number is: "<<fact<<endl;  
    return 0;  
}  
int factorial(int n)  
{  
    if(n<0)  
        return(-1); /*Wrong value*/  
    if(n==0)  
        return(1); /*Terminating condition*/  
}
```

```
else
{
    return(n*factorial(n-1));
}
```

Output:

```
Enter any number: 5
Factorial of a number is: 120
```

We can understand the above program of recursive method call by the figure given below:

```
return 5 * factorial(4) = 120
    ↘ return 4 * factorial(3) = 24
        ↘ return 3 * factorial(2) = 6
            ↘ return 2 * factorial(1) = 2
                ↘ return 1 * factorial(0) = 1
```

[javaTpoint.com](http://javatpoint.com)

$1 * 2 * 3 * 4 * 5 = 120$

Fig: Recursion

← Prev

Next →

AD

C++ Storage Classes

Storage class is used to define the lifetime and visibility of a variable and/or function within a C++ program.

Lifetime refers to the period during which the variable remains active and visibility refers to the module of a program in which the variable is accessible.

There are five types of storage classes, which can be used in a C++ program

1. Automatic
2. Register
3. Static
4. External
5. Mutable

Storage Class	Keyword	Lifetime	Visibility	Initial Value
Automatic	auto	Function Block	Local	Garbage
Register	register	Function Block	Local	Garbage
Mutable	mutable	Class	Local	Garbage
External	extern	Whole Program	Global	Zero
Static	static	Whole Program	Local	Zero

Automatic Storage Class

It is the default storage class for all local variables. The auto keyword is applied to all local variables automatically.

```
{
auto int y;
float y = 3.45;
}
```

The above example defines two variables with a same storage class, auto can only be used within functions.

Register Storage Class

The register variable allocates memory in register than RAM. Its size is same of register size. It has a faster access than other variables.

It is recommended to use register variable only for quick access such as in counter.

Note: We can't get the address of register variable.

```
register int counter=0;
```

Static Storage Class

The static variable is initialized only once and exists till the end of a program. It retains its value between multiple functions call.

The static variable has the default value 0 which is provided by compiler.

```
#include <iostream>
using namespace std;
void func() {
    static int i=0; //static variable
    int j=0; //local variable
    i++;
    j++;
    cout<<"i=" << i << " and j=" << j << endl;
}
int main()
{
    func();
    func();
    func();
}
```

Output:

```
i= 1 and j= 1  
i= 2 and j= 1  
i= 3 and j= 1
```

AD

U B E R

TAP A BUTTON. GET A RIDE.
Sign up here for \$20 off your first trip!

External Storage Class

The extern variable is visible to all the programs. It is used if two or more files are sharing same variable or function.

```
extern int counter=0;
```

← Prev

Next →

AD

 For Videos Join Our Youtube Channel: [Join Now](#)

Feedback

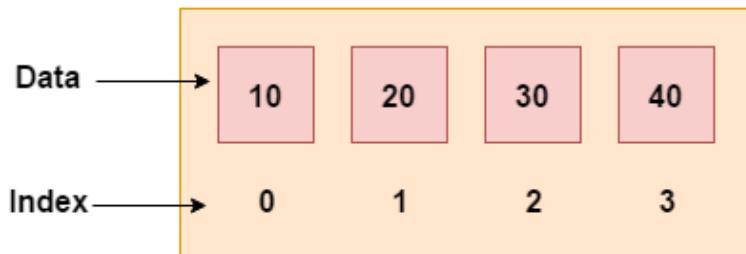
- Send your Feedback to feedback@javatpoint.com

C++ Arrays

Like other programming languages, array in C++ is a group of similar types of elements that have contiguous memory location.

In C++ **std::array** is a container that encapsulates fixed size arrays. In C++, array index starts from 0. We can store only fixed set of elements in C++ array.

A collection of related data items stored in adjacent memory places is referred to as an array in the C/C++ programming language or any other programming language for that matter. Elements of an array can be accessed arbitrarily using its indices. They can be used to store a collection of any type of primitive data type, including int, float, double, char, etc. An array in C/C++ can also store derived data types like structures, pointers, and other data types, which is an addition. The array representation in a picture is provided below.



Advantages of C++ Array

- Code Optimization (less code)
- Random Access
- Easy to traverse data
- Easy to manipulate data
- Easy to sort data etc.

Disadvantages of C++ Array

- Fixed size



C++ Array Types

There are 2 types of arrays in C++ programming:

1. Single Dimensional Array
2. Multidimensional Array



C++ Single Dimensional Array

Let's see a simple example of C++ array, where we are going to create, initialize and traverse array.

```
#include <iostream>
using namespace std;
int main()
{
    int arr[5]={10, 0, 20, 0, 30}; //creating and initializing array
    //traversing array
    for (int i = 0; i < 5; i++)
    {
        cout<<arr[i]<<"\n";
    }
}
```

Output:

```
10
0
20
0
30
```

C++ Array Example: Traversal using foreach loop

We can also traverse the array elements using foreach loop. It returns array element one by one.

```
#include <iostream>
using namespace std;
int main()
{
    int arr[5]={10, 0, 20, 0, 30}; //creating and initializing array
    //traversing array
    for (int i: arr)
    {
        cout<<i<<"\n";
    }
}
```

Output:

```
10
20
30
40
50
```

Why do we need arrays?

With a limited number of objects, we can use regular variables (v1, v2, v3..), but when we need to hold many instances, managing them with normal variables becomes challenging. To represent numerous instances in one variable, we use an array.

What happens if we try to access out of bound array?

The array's elements will be numbered 0 through 9 if we define an array of size 10.

We will have Undefined Behaviour if we attempt to access an element at an index higher than 10, though.

C++ array with empty members

The maximum number of elements that can be stored in an array in C++ is n. What will happen, though, if we store fewer than n elements?

For example,

```
// store only 3 elements in the array  
int x[6] = {19, 10, 8};
```

The array x in this case is 6 elements wide. But we've only given it a three-element initialization.

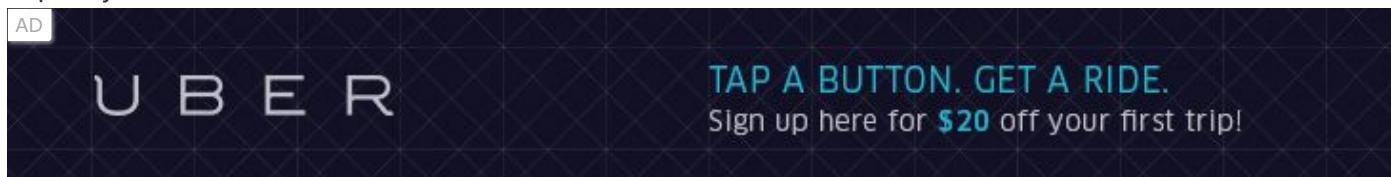
When that happens, the compiler fills in the empty spaces with random values. This random value frequently appears as 0.

Important things to remember while using arrays in C++

1. The array's indexes begin at 0. Meaning that the first item saved at index 0 is x[0].
2. The final element of an array with size n is kept at index (n-1). This example's final element is x[5].
3. An array's elements have sequential addresses. Consider the scenario where x[0] 's address is 2120.

The address of the subsequent element, x[1], will then be 2124, followed by x[2], 2128, and so forth.

Each element in this case has a four-fold increase in size. This is due to the fact that int has a 4 byte capacity.



What is two-dimensional array?

Each element in this kind of array is described by two indexes, the first of which denotes a row and the second of which denotes a column.

As you can see, the components are arranged in a two-dimensional array using rows and columns; there are I number of rows and j number of columns.

What is a multi-dimensional array?

A two-dimensional array is the most basic type of multidimensional array; it also qualifies as a multidimensional array. There are no restrictions on the array's dimensions.

How to insert it in array?

```
int mark[5] = {19, 10, 8, 17, 9}
```

```
// change 4th element to 9
mark[3] = 9;
// take input from the user
// store the value at third position
cin >> mark[2];
// take input from the user
// insert at ith position
cin >> mark[i-1];

// print first element of the array
cout << mark[0];
// print ith element of the array
cout >> mark[i-1];
```

How to display the sum and average of array elements?

```
#include <iostream>
using namespace std;
int main() {
    // initialize an array without specifying the size
    double numbers[] = {7, 5, 6, 12, 35, 27};
    double sum = 0;
    double count = 0;
    double average;
    cout << "The numbers are: ";
    // print array elements
    // use of range-based for loop
    for (const double &n : numbers) {
        cout << n << " ";
    }
    // calculate the sum
    sum += n;
    // count the no. of array elements
    ++count;
}
// print the sum
cout << "\nTheir Sum = " << sum << endl;
// find the average
```

```
average = sum / count;  
cout << "Their Average = " << average << endl;  
  
return 0;  
}
```

Output:

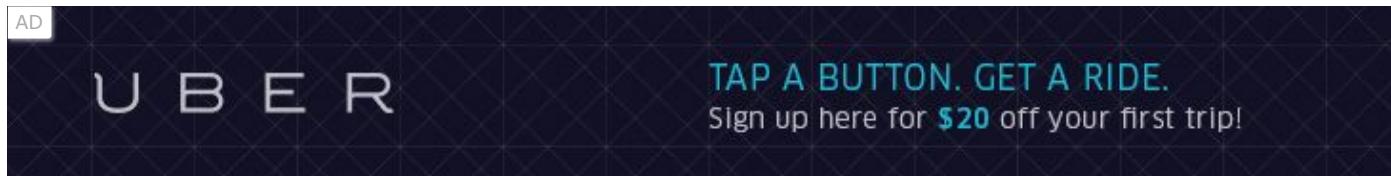
```
The numbers are: 7 5 6 12 35 27  
Their Sum = 92  
Their Average = 15.3333
```

How to display array elements?

```
#include <iostream>  
using namespace std;  
int main() {  
    int numbers[5] = {7, 5, 6, 12, 35};  
    cout << "The numbers are: ";  
    // Printing array elements  
    // using range-based for loop  
    for (const int &n : numbers) {  
        cout << n << " ";  
    }  
    cout << "\nThe numbers are: ";  
    // Printing array elements  
    // using traditional for loop  
    for (int i = 0; i < 5; ++i) {  
        cout << numbers[i] << " ";  
    }  
    return 0;  
}
```

Output:

AD



The banner features the word "UBER" in large white letters. To the right, it says "TAP A BUTTON. GET A RIDE." and "Sign up here for \$20 off your first trip!"

The numbers are: 7 5 6 12 35

The numbers are: 7 5 6 12 35

← Prev

Next →

AD

 [For Videos Join Our YouTube Channel: Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share



Learn Latest Tutorials

 [Splunk tutorial](#)

Splunk

 [SPSS tutorial](#)

SPSS

 [Swagger tutorial](#)

Transact-SQL

 [T-SQL tutorial](#)

C++ Multidimensional Arrays

The multidimensional array is also known as rectangular arrays in C++. It can be two dimensional or three dimensional. The data is stored in tabular form (row * column) which is also known as matrix.

C++ Multidimensional Array Example

Let's see a simple example of multidimensional array in C++ which declares, initializes and traverse two dimensional arrays.

```
#include <iostream>
using namespace std;
int main()
{
    int test[3][3]; //declaration of 2D array
    test[0][0]=5; //initialization
    test[0][1]=10;
    test[1][1]=15;
    test[1][2]=20;
    test[2][0]=30;
    test[2][2]=10;
    //traversal
    for(int i = 0; i < 3; ++i)
    {
        for(int j = 0; j < 3; ++j)
        {
            cout<< test[i][j]<< " ";
        }
        cout<< "\n"; //new line at each row
    }
    return 0;
}
```

Output:

```
5 10 0
0 15 20
30 0 10
```

C++ Multidimensional Array Example: Declaration and initialization at same time

Let's see a simple example of multidimensional array which initializes array at the time of declaration.

```
#include <iostream>
using namespace std;
int main()
{
    int test[3][3] =
    {
        {2, 5, 5},
        {4, 0, 3},
        {9, 1, 8} }; //declaration and initialization
    //traversal
    for(int i = 0; i < 3; ++i)
    {
        for(int j = 0; j < 3; ++j)
        {
            cout<< test[i][j]<< " ";
        }
        cout<< "\n"; //new line at each row
    }
    return 0;
}
```

Output:"

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

9 1 8

[← Prev](#)[Next →](#)

AD

 [For Videos Join Our Youtube Channel: Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share



Learn Latest Tutorials

 [Splunk tutorial](#)

Splunk

 [SPSS tutorial](#)

SPSS

 [Swagger tutorial](#)

Swagger

 [T-SQL tutorial](#)

Transact-SQL

 [Tumblr tutorial](#)

Tumblr

 [React tutorial](#)

ReactJS

 [Regex tutorial](#)

Regex

 [Reinforcement learning tutorial](#)

Reinforcement Learning

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
```

C++ Passing Array to Function Example: print array elements

Let's see an example of C++ function which prints the array elements.

```
#include <iostream>
using namespace std;
void printArray(int arr[5]);
int main()
{
    int arr1[5] = { 10, 20, 30, 40, 50 };
    int arr2[5] = { 5, 15, 25, 35, 45 };
    printArray(arr1); //passing array to function
    printArray(arr2);
}
void printArray(int arr[5])
{
    cout << "Printing array elements:" << endl;
    for (int i = 0; i < 5; i++)
    {
        cout << arr[i] << "\n";
    }
}
```

Output:

```
Printing array elements:
```

```
10
20
30
40
```

```
50
Printing array elements:
5
15
25
35
45
```

C++ Passing Array to Function Example: Print minimum number

Let's see an example of C++ array which prints minimum number in an array using function.

```
#include <iostream>
using namespace std;
void printMin(int arr[5]);
int main()
{
    int arr1[5] = { 30, 10, 20, 40, 50 };
    int arr2[5] = { 5, 15, 25, 35, 45 };
    printMin(arr1); //passing array to function
    printMin(arr2);
}
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";
}
```

Output:

```
Minimum element is: 10
```

Minimum element is: 5

C++ Passing Array to Function Example: Print maximum number

Let's see an example of C++ array which prints maximum number in an array using function.

```
#include <iostream>
using namespace std;
void printMax(int arr[5]);
int main()
{
    int arr1[5] = { 25, 10, 54, 15, 40 };
    int arr2[5] = { 12, 23, 44, 67, 54 };
    printMax(arr1); //Passing array to function
    printMax(arr2);
}
void printMax(int arr[5])
{
    int max = arr[0];
    for (int i = 0; i < 5; i++)
    {
        if (max < arr[i])
        {
            max = arr[i];
        }
    }
    cout<< "Maximum element is: "<< max << "\n";
}
```

Output:

```
Maximum element is: 54
Maximum element is: 67
```

← Prev

Next →

C++ Pointers

The pointer in C++ language is a variable, it is also known as locator or indicator that points to an address of a value.

The symbol of an address is represented by a pointer. In addition to creating and modifying dynamic data structures, they allow programs to emulate call-by-reference. One of the principal applications of pointers is iterating through the components of arrays or other data structures. The pointer variable that refers to the same data type as the variable you're dealing with has the address of that variable set to it (such as an int or string).

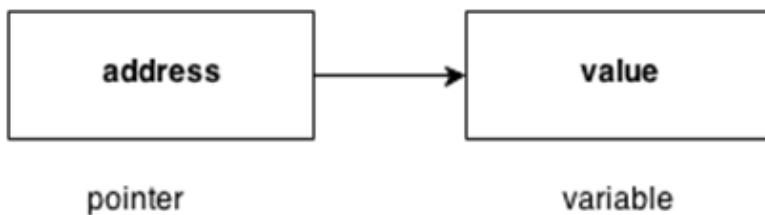
Syntax

```
datatype *var_name;
int *ptr; // ptr can point to an address which holds int data
```

How to use a pointer?

1. Establish a pointer variable.
2. employing the unary operator (&), which yields the address of the variable, to assign a pointer to a variable's address.
3. Using the unary operator (*), which gives the variable's value at the address provided by its argument, one can access the value stored in an address.

Since the data type knows how many bytes the information is held in, we associate it with a reference. The size of the data type to which a pointer points is added when we increment a pointer.



Advantage of pointer

- 1) Pointer reduces the code and improves the performance, it is used to retrieving strings, trees etc. and used with arrays, structures and functions.
- 2) We can return multiple values from function using pointer.

3) It makes you able to access any memory location in the computer's memory.

Usage of pointer

There are many usage of pointers in C++ language.

1) Dynamic memory allocation

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

2) Arrays, Functions and Structures

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

Symbols used in pointer

Symbol	Name	Description
& (ampersand sign)	Address operator	Determine the address of a variable.
* (asterisk sign)	Indirection operator	Access the value of an address.

Declaring a pointer

The pointer in C++ language can be declared using * (asterisk symbol).

```
int * a; //pointer to int  
char * c; //pointer to char
```

AD

Pointer Example

Let's see the simple example of using pointers printing the address and value.

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

```

int main()
{
    int number=30;
    int * p;
    p=&number;//stores the address of number variable
    cout<<"Address of number variable is:" <<&number<<endl;
    cout<<"Address of p variable is:" <<p<<endl;
    cout<<"Value of p variable is:" <<*p<<endl;
    return 0;
}

```

Output:

```

Address of number variable is:0x7ffccc8724c4
Address of p variable is:0x7ffccc8724c4
Value of p variable is:30

```

Pointer Program to swap 2 numbers without using 3rd variable

```

#include <iostream>
using namespace std;
int main()
{
    int a=20,b=10,*p1=&a,*p2=&b;
    cout<<"Before swap: *p1=" <<*p1 << " *p2=" <<*p2<<endl;
    *p1=*p1+*p2;
    *p2=*p1-*p2;
    *p1=*p1-*p2;
    cout<<"After swap: *p1=" <<*p1 << " *p2=" <<*p2<<endl;
    return 0;
}

```

Output:

```

Before swap: *p1=20 *p2=10
After swap: *p1=10 *p2=20

```

What are Pointer and string literals?

String literals are arrays of character sequences with null ends. The elements of a string literal are arrays of type const char (because characters in a string cannot be modified) plus a terminating null-character.

What is a void pointer?

This unique type of pointer, which is available in C++, stands in for the lack of a kind. Pointers that point to a value that has no type are known as void pointers (and thus also an undetermined length and undetermined dereferencing properties). This indicates that void pointers are very flexible because they can point to any data type. This flexibility has benefits. Direct dereference is not possible with these pointers. Before they may be dereferenced, they must be converted into another pointer type that points to a specific data type.

AD

What is a invalid pointer?

A pointer must point to a valid address, not necessarily to useful items (like for arrays). We refer to these as incorrect pointers. Additionally, incorrect pointers are uninitialized pointers.

```
int *ptr1;  
int arr[10];  
int *ptr2 = arr+20;
```

Here, ptr1 is not initialized, making it invalid, and ptr2 is outside of the bounds of arr, making it likewise weak. (Take note that not all build failures are caused by faulty references.)

What is a null pointer?

A null pointer is not merely an incorrect address; it also points nowhere. Here are two ways to mark a pointer as NULL:

```
int *ptr1 = 0;  
int *ptr2 = NULL;
```

What is a pointer to a pointer?

In C++, we have the ability to build a pointer to another pointer, which might then point to data or another pointer. The unary operator (*) is all that is needed in the syntax for declaring the pointer for each level of indirection.

```
char a;  
char *b;  
char ** c;  
a = 'g';  
b = &a;  
c = &b;
```

Here b points to a char that stores 'g', and c points to the pointer b.

AD

What are references and pointers?

1. Call-By-Value
2. Call-By-Reference with a Pointer Argument
3. Call-By-Reference with a Reference Argument

Example

```
#include  
using namespace std;  
  
// Pass-by-Value  
int square1(int n)  
{cout << "address of n1 in square1(): " << &n << "\n";  
n *= n;  
return n;  
}  
  
// Pass-by-Reference with Pointer Arguments  
void square2(int* n)  
{
```

```
cout << "address of n2 in square2(): " << n << "\n";
*n *= *n;
}

// Pass-by-Reference with Reference Arguments

void square3(int& n)
{

cout << "address of n3 in square3(): " << &n << "\n";
n *= n;
}

void example()
{
    // Call-by-Value
    int n1 = 8;
    cout << "address of n1 in main(): " << &n1 << "\n";
    cout << "Square of n1: " << square1(n1) << "\n";
    cout << "No change in n1: " << n1 << "\n";

    // Call-by-Reference with Pointer Arguments
    int n2 = 8;
    cout << "address of n2 in main(): " << &n2 << "\n";
    square2(&n2);
    cout << "Square of n2: " << n2 << "\n";
    cout << "Change reflected in n2: " << n2 << "\n";

    // Call-by-Reference with Reference Arguments
    int n3 = 8;
    cout << "address of n3 in main(): " << &n3 << "\n";
    square3(n3);
    cout << "Square of n3: " << n3 << "\n";
    cout << "Change reflected in n3: " << n3 << "\n";
}

// Driver program
int main() { example(); }
```

Output

```
address of n1 in main(): 0x7fffa7e2de64
address of n1 in square1(): 0x7fffa7e2de4c
Square of n1: 64
No change in n1: 8
address of n2 in main(): 0x7fffa7e2de68
address of n2 in square2(): 0x7fffa7e2de68
Square of n2: 64
Change reflected in n2: 64
address of n3 in main(): 0x7fffa7e2de6c
address of n3 in square3(): 0x7fffa7e2de6c
Square of n3: 64
Change reflected in n3: 64
```

[← Prev](#)[Next →](#)

AD

 For Videos Join Our YouTube Channel: [Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

sizeof() operator in C++

The sizeof() is an operator that evaluates the size of data type, constants, variable. It is a compile-time operator as it returns the size of any variable or a constant at the compilation time.

The size, which is calculated by the sizeof() operator, is the amount of RAM occupied in the computer.

Syntax of the sizeof() operator is given below:

```
sizeof(data_type);
```

In the above syntax, the data_type can be the data type of the data, variables, constants, unions, structures, or any other user-defined data type.

The sizeof () operator can be applied to the following operand types:

- **When an operand is of data type**

If the parameter of a sizeof() operator contains the data type of a variable, then the sizeof() operator will return the size of the data type.

Let's understand this scenario through an example.

```
#include <iostream>
using namespace std;
int main()
{
    // Determining the space in bytes occupied by each data type.
    std::cout << "Size of integer data type : " << sizeof(int) << std::endl;
    std::cout << "Size of float data type : " << sizeof(float) << std::endl;
    std::cout << "Size of double data type : " << sizeof(double) << std::endl;
    std::cout << "Size of char data type : " << sizeof(char) << std::endl;
    return 0;
}
```

In the above program, we have evaluated the size of the in-built data types by using the sizeof() operator. As we know that int occupies 4 bytes, float occupies 4 bytes, double occupies 8 bytes, and char occupies 1 byte, and the same result is shown by the sizeof() operator as we can observe in the

following output.

Output

```
v / s
Size of integer data type : 4
Size of float data type : 4
Size of double data type : 8
Size of char data type : 1

...Program finished with exit code 0
Press ENTER to exit console.□
```

- When an operand is of Class type.



```
#include <iostream>
using namespace std;
class Base
{
int a;
};
int main()
{
Base b;
std::cout << "Size of class base is :" << sizeof(b) << std::endl;
return 0;
}
```

In the above program, we have evaluated the size of the class, which is having a single integer variable. The output would be 4 bytes as int variable occupies 4 bytes.

Output

```
v / s
Size of class base is : 4

...Program finished with exit code 0
Press ENTER to exit console.□
```

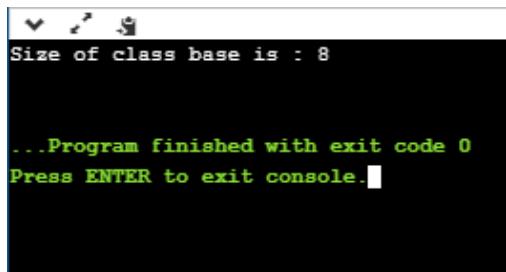
If we add one more integer variable in a class, then the code would look like:

```
#include <iostream>
using namespace std;
class Base
{
    int a;
    int d;
};

int main()
{
    Base b;
    std::cout << "Size of class base is :" << sizeof(b) << std::endl;
    return 0;
}
```

In the above code, we have added one more integer variable. In this case, the size of the class would be 8 bytes as **int** variable occupies 4 bytes, so two integer variables occupy 8 bytes.

Output



```
Size of class base is : 8

...Program finished with exit code 0
Press ENTER to exit console.
```

If we add a char variable in the above code, then the code would look like:

```
#include <iostream>

using namespace std;

class Base
{
    int a;
    int d;
    char ch;
```

```

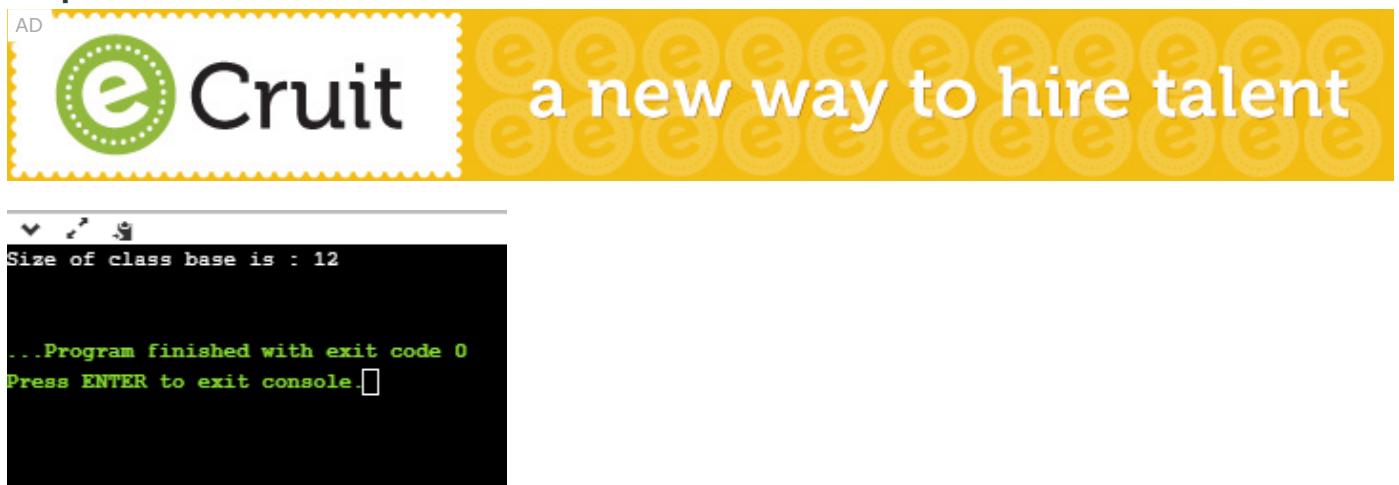
};

int main()
{
    Base b;
    std::cout << "Size of class base is : " << sizeof(b) << std::endl;
    return 0;
}

```

In the above code, the class has two integer variables, and one char variable. According to our calculation, the size of the class would be equal to 9 bytes (int+int+char), but this is wrong due to the concept of structure padding.

Output



- When an operand is of array type.

```

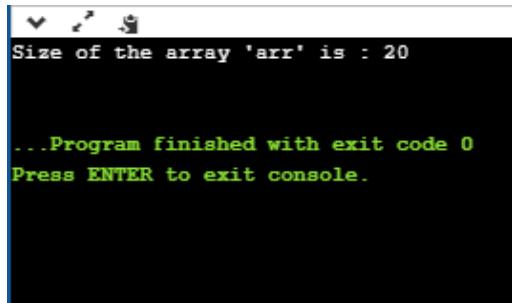
#include <iostream>
using namespace std;
int main()
{
    int arr[]={10,20,30,40,50};
    std::cout << "Size of the array 'arr' is : " << sizeof(arr) << std::endl;
    return 0;
}

```

In the above program, we have declared an array of integer type which contains five elements. We have evaluated the size of the array by using **sizeof()** operator. According to our calculation, the size of the array should be 20 bytes as int data type occupies 4 bytes, and array contains 5 elements,

so total memory space occupied by this array is $5 \times 4 = 20$ bytes. The same result has been shown by the **sizeof()** operator as we can observe in the following output.

Output



```
Size of the array 'arr' is : 20

...Program finished with exit code 0
Press ENTER to exit console.
```

Let's consider another scenario of an array.

```
#include <iostream>
using namespace std;
void fun(int arr[])
{
    std::cout << "Size of array is :" << sizeof(arr) << std::endl;
}
int main()
{
    int arr[] = {10, 20, 30, 40, 50};
    fun(arr);
    return 0;
}
```

In the above program, we have tried to print the size of the array using the function. In this case, we have created an array of type integer, and we pass the 'arr' to the function **fun()**. The **fun()** would return the size of the integer pointer, i.e., **int***, and the size of the **int*** is 8 bytes in the 64-bit operating system.

Output



AD

GAIAM TV
TRANSFORMATION NETWORK

VIDEO STREAMING FOR THE AWAKENED MIND

GET STARTED NOW

```
main.cpp:15:52: warning: 'sizeof' on array function parameter 'arr' will return size of 'int*' [-Wsizeof-array-argument]
main.cpp:13:18: note: declared here
Size of array is : 8

...Program finished with exit code 0
Press ENTER to exit console.
```

- When an operand is of pointer type.

```
#include <iostream>
using namespace std;
int main()
{
    int *ptr1=new int(10);
    std::cout     <<      "size      of      ptr1      :      "      << sizeof(ptr1)
<< std::endl;
    std::cout << "size of *ptr1 : " << sizeof(*ptr1)<< std::endl;
    char *ptr2=new char('a');
    std::cout << "size of ptr2 : " << sizeof(ptr2)<< std::endl;
    std::cout << "size of *ptr2 : " << sizeof(*ptr2)<< std::endl;
    double *ptr3=new double(12.78);
    std::cout << "size of ptr3 : " << sizeof(ptr3)<< std::endl;
    std::cout << "size of *ptr3 : " << sizeof(*ptr3)<< std::endl;
    return 0;
}
```

In the above program, we have determined the size of pointers. The size of pointers would remain same for all the data types. If the computer has 32bit operating system, then the size of the pointer would be 4 bytes. If the computer has 64-bit operating system, then the size of the pointer would be 8 bytes. I am running this program on 64-bit, so the output would be 8 bytes. Now, if we provide the '*' symbol to the pointer, then the output depends on the data type, for example, *ptr1 is of integer type means the sizeof() operator will return 4 bytes as int data type occupies 4 bytes.

Output

The screenshot shows a terminal window with the following text:
size of ptr1 : 8
size of *ptr1 : 4
size of ptr2 : 8
size of *ptr2 : 1
size of ptr3 : 8
size of *ptr3 : 8

...Program finished with exit code 0
Press ENTER to exit console.

- o When an operand is an expression.

```
#include <iostream>  
using namespace std;  
  
int main()  
{  
    int num1;  
    double num2;  
    cout << sizeof(num1+num2);  
    return 0;  
}
```

In the above program, we have declared two variables num1 and num2 of type int and double, respectively. The size of the int is 4 bytes, while the size of double is 8 bytes. The result would be the variable, which is of double type occupying 8 bytes.

Output

The screenshot shows a terminal window with the following text:
8
...Program finished with exit code 0
Press ENTER to exit console.

C++ Array of Pointers

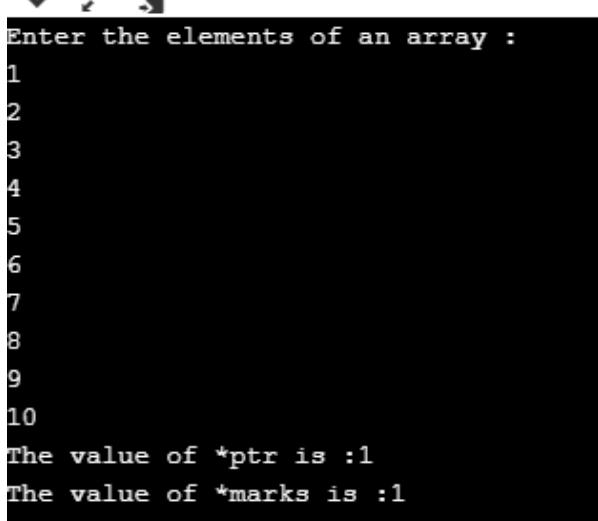
Array and pointers are closely related to each other. In C++, the name of an array is considered as a pointer, i.e., the name of an array contains the address of an element. C++ considers the array name as the address of the first element. For example, if we create an array, i.e., marks which hold the 20 values of integer type, then marks will contain the address of first element, i.e., marks[0]. Therefore, we can say that array name (marks) is a pointer which is holding the address of the first element of an array.

Let's understand this scenario through an example.

```
#include <iostream>
using namespace std;
int main()
{
    int *ptr; // integer pointer declaration
    int marks[10]; // marks array declaration
    std::cout << "Enter the elements of an array :" << std::endl;
    for(int i=0;i<10;i++)
    {
        cin>>marks[i];
    }
    ptr=marks; // both marks and ptr pointing to the same element..
    std::cout << "The value of *ptr is :" << *ptr << std::endl;
    std::cout << "The value of *marks is :" << *marks << std::endl;
}
```

In the above code, we declare an integer pointer and an array of integer type. We assign the address of marks to the ptr by using the statement ptr=marks; it means that both the variables 'marks' and 'ptr' point to the same element, i.e., marks[0]. When we try to print the values of *ptr and *marks, then it comes out to be same. Hence, it is proved that the array name stores the address of the first element of an array.

Output



```
Enter the elements of an array :  
1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
The value of *ptr is :1  
The value of *marks is :1
```

Array of Pointers

An array of pointers is an array that consists of variables of pointer type, which means that the variable is a pointer addressing to some other element. Suppose we create an array of pointer holding 5 integer pointers; then its declaration would look like:

```
int *ptr[5]; // array of 5 integer pointer.
```

In the above declaration, we declare an array of pointer named as ptr, and it allocates 5 integer pointers in memory.

The element of an array of a pointer can also be initialized by assigning the address of some other element. Let's observe this case through an example.

```
int a; // variable declaration.  
ptr[2] = &a;
```

In the above code, we are assigning the address of 'a' variable to the third element of an array 'ptr'.

We can also retrieve the value of 'a' be dereferencing the pointer.

```
*ptr[2];
```

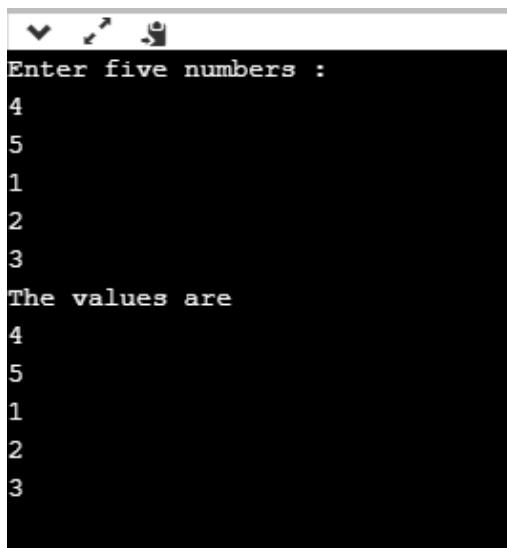
Let's understand through an example.

```
#include <iostream>
```

```
using namespace std;
int main()
{
    int ptr1[5]; // integer array declaration
    int *ptr2[5]; // integer array of pointer declaration
    std::cout << "Enter five numbers :" << std::endl;
    for(int i=0;i<5;i++)
    {
        std::cin >> ptr1[i];
    }
    for(int i=0;i<5;i++)
    {
        ptr2[i]=&ptr1[i];
    }
    // printing the values of ptr1 array
    std::cout << "The values are" << std::endl;
    for(int i=0;i<5;i++)
    {
        std::cout << *ptr2[i] << std::endl;
    }
}
```

In the above code, we declare an array of integer type and an array of integer pointers. We have defined the 'for' loop, which iterates through the elements of an array 'ptr1', and on each iteration, the address of element of ptr1 at index 'i' gets stored in the ptr2 at index 'i'.

Output



```
Enter five numbers :
4
5
1
2
3
The values are
4
5
1
2
3
```

Till now, we have learnt the array of pointers to an integer. Now, we will see how to create the array of pointers to strings.

Array of Pointer to Strings

An array of pointer to strings is an array of character pointers that holds the address of the first character of a string or we can say the base address of a string.

The following are the differences between an array of pointers to string and two-dimensional array of characters:

- An array of pointers to string is more efficient than the two-dimensional array of characters in case of memory consumption because an array of pointer to strings consumes less memory than the two-dimensional array of characters to store the strings.
- In an array of pointers, the manipulation of strings is comparatively easier than in the case of 2d array. We can also easily change the position of the strings by using the pointers.

Let's see how to declare the array of pointers to string.

First, we declare the array of pointer to string:



```
char *names[5] = {"john",
                  "Peter",
                  "Marco",
                  "Devin",
                  "Ronan"};
```

In the above code, we declared an array of pointer names as 'names' of size 5. In the above case, we have done the initialization at the time of declaration, so we do not need to mention the size of the array of a pointer. The above code can be re-written as:

```
char *names[ ] = {"john",
                  "Peter",
                  "Marco",
                  "Devin",
                  "Ronan"};
```

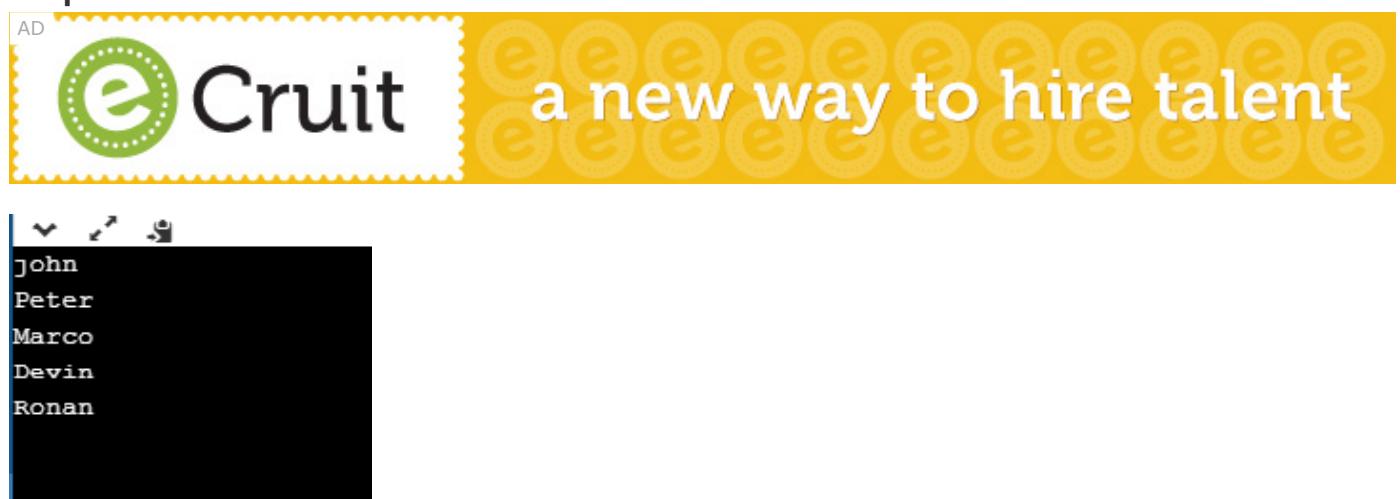
In the above case, each element of the 'names' array is a string literal, and each string literal would hold the base address of the first character of a string. For example, names[0] contains the base address of "john", names[1] contains the base address of "Peter", and so on. It is not guaranteed that all the string literals will be stored in the contiguous memory location, but the characters of a string literal are stored in a contiguous memory location.

Let's create a simple example.

```
#include <iostream>
using namespace std;
int main()
{
    char *names[5] = {"john",
                      "Peter",
                      "Marco",
                      "Devin",
                      "Ronan"};
    for(int i=0;i<5;i++)
    {
        std::cout << names[i] << std::endl;
    }
    return 0;
}
```

In the above code, we have declared an array of char pointer holding 5 string literals, and the first character of each string is holding the base address of the string.

Output



```
AD eCruit a new way to hire talent
john
Peter
Marco
Devin
Ronan
```

C++ Void Pointer

A void pointer is a general-purpose pointer that can hold the address of any data type, but it is not associated with any data type.

Syntax of void pointer

```
void *ptr;
```

In C++, we cannot assign the address of a variable to the variable of a different data type. Consider the following example:

```
int *ptr; // integer pointer declaration
float a=10.2; // floating variable initialization
ptr= &a; // This statement throws an error.
```

In the above example, we declare a pointer of type integer, i.e., ptr and a float variable, i.e., 'a'. After declaration, we try to store the address of 'a' variable in 'ptr', but this is not possible in C++ as the variable cannot hold the address of different data types.

Let's understand through a simple example.

```
#include <iostream.h>
using namespace std;
int main()
{
    int *ptr;
    float f=10.3;
    ptr = &f; // error
    std::cout << "The value of *ptr is : " << *ptr << std::endl;
    return 0;
}
```

In the above program, we declare a pointer of integer type and variable of float type. An integer pointer variable cannot point to the float variable, but it can point to an only integer variable.

Output

```
input
Compilation failed due to following error(s).

main.cpp: In function 'int main()':
main.cpp:17:12: error: cannot convert 'float*' to 'int*' in assignment
ptr = &f;
^
```

C++ has overcome the above problem by using the C++ void pointer as a void pointer can hold the address of any data type.

Let's look at a simple example of void pointer.

```
#include <iostream>
using namespace std;
int main()
{
    void *ptr; // void pointer declaration
    int a=9; // integer variable initialization
    ptr=&a; // storing the address of 'a' variable in a void pointer variable.
    std::cout << &a << std::endl;
    std::cout << ptr << std::endl;
    return 0;
}
```

In the above program, we declare a void pointer variable and an integer variable where the void pointer contains the address of an integer variable.

Output

```
0x7ffcf1da5e04
0x7ffcf1da5e04

...Program finished with exit code 0
Press ENTER to exit console.
```

Difference between void pointer in C and C++

In C, we can assign the void pointer to any other pointer type without any typecasting, whereas in C++, we need to typecast when we assign the void pointer type to any other pointer type.

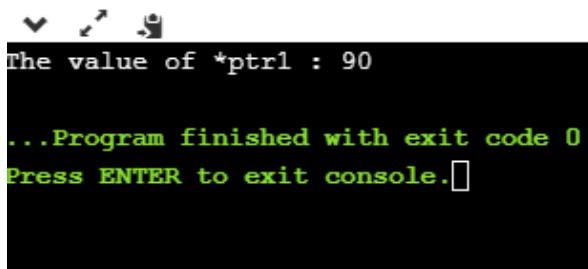
Let's understand through a simple example.

In C,

```
#include <stdio.h>
int main()
{
    void *ptr; // void pointer declaration
    int *ptr1; // integer pointer declaration
    int a = 90; // integer variable initialization
    ptr=&a; // storing the address of 'a' in ptr
    ptr1=ptr; // assigning void pointer to integer pointer type.
    printf("The value of *ptr1 : %d",*ptr1);
    return 0;
}
```

In the above program, we declare two pointers 'ptr' and 'ptr1' of type void and integer, respectively. We also declare the integer type variable, i.e., 'a'. After declaration, we assign the address of 'a' variable to the pointer 'ptr'. Then, we assign the void pointer to the integer pointer, i.e., ptr1 without any typecasting because in C, we do not need to typecast while assigning the void pointer to any other type of pointer.

Output



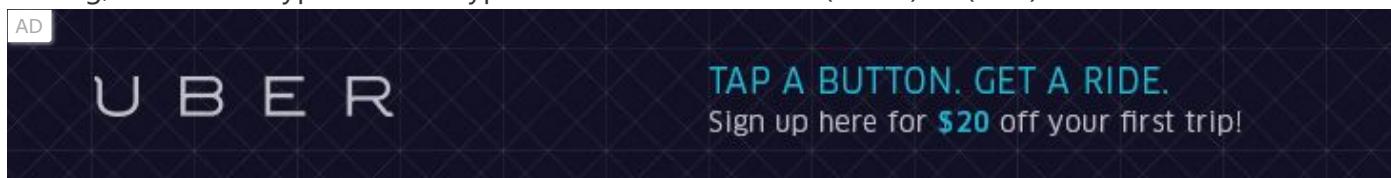
```
The value of *ptr1 : 90
...Program finished with exit code 0
Press ENTER to exit console.█
```

In C++,

```
#include <iostream>
using namespace std;
int main()
```

```
{  
void *ptr; // void pointer declaration  
int *ptr1; // integer pointer declaration  
int data=10; // integer variable initialization  
ptr=&data; // storing the address of data variable in void pointer variable  
ptr1=(int *)ptr; // assigning void pointer to integer pointer  
std::cout << "The value of *ptr1 is : " << *ptr1 << std::endl;  
return 0;  
}
```

In the above program, we declare two pointer variables of type void and int type respectively. We also create another integer type variable, i.e., 'data'. After declaration, we store the address of variable 'data' in a void pointer variable, i.e., ptr. Now, we want to assign the void pointer to integer pointer, in order to do this, we need to apply the cast operator, i.e., (int *) to the void pointer variable. This cast operator tells the compiler which type of value void pointer is holding. For casting, we have to type the data type and * in a bracket like (char *) or (int *).



Output

```
The value of *ptr1 is : 10  
  
...Program finished with exit code 0  
Press ENTER to exit console.□
```

← Prev

Next →

C++ References

Till now, we have read that C++ supports two types of variables:

- An ordinary variable is a variable that contains the value of some type. For example, we create a variable of type int, which means that the variable can hold the value of type integer.
- A pointer is a variable that stores the address of another variable. It can be dereferenced to retrieve the value to which this pointer points to.
- There is another variable that C++ supports, i.e., references. It is a variable that behaves as an alias for another variable.

How to create a reference?

Reference can be created by simply using an ampersand (&) operator. When we create a variable, then it occupies some memory location. We can create a reference of the variable; therefore, we can access the original variable by using either name of the variable or reference. For example,

```
int a=10;
```

Now, we create the reference variable of the above variable.

```
int &ref=a;
```

The above statement means that 'ref' is a reference variable of 'a', i.e., we can use the 'ref' variable in place of 'a' variable.

C++ provides two types of references:

- References to non-const values
- References as aliases

AD



VIDEO STREAMING FOR THE AWAKENED MIND

GET STARTED NOW

References to non-const values

It can be declared by using & operator with the reference type variable.

```
#include <iostream>
using namespace std;
int main()
{
    int a=10;
    int &value=a;
    std::cout << value << std::endl;
    return 0;
}
```

Output

```
10
```

References as aliases

References as aliases is another name of the variable which is being referenced.

For example,

```
int a=10; // 'a' is a variable.
int &b=a; // 'b' reference to a.
int &c=a; // 'c' reference to a.
```

Let's look at a simple example.

```
#include <iostream>
using namespace std;
int main()
{
    int a=70; // variable initialization
    int &b=a;
    int &c=a;
    std::cout << "Value of a is :" << a << std::endl;
    std::cout << "Value of b is :" << b << std::endl;
    std::cout << "Value of c is :" << c << std::endl;
    return 0;
}
```

In the above code, we create a variable 'a' which contains a value '70'. We have declared two reference variables, i.e., b and c, and both are referring to the same variable 'a'. Therefore, we can say that 'a' variable can be accessed by 'b' and 'c' variable.

Output

```
Value of a is :70
Value of b is :70
Value of c is :70
```

Properties of References

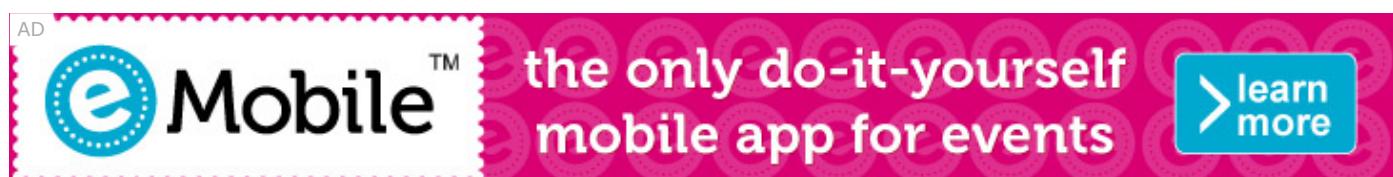
The following are the properties of references:

Initialization

It must be initialized at the time of the declaration.

```
#include <iostream>
using namespace std;
int main()
{
    int a=10; // variable initialization
    int &b=a; // b reference to a
    std::cout << "value of a is " << b << std::endl;
    return 0;
}
```

In the above code, we have created a reference variable, i.e., 'b'. At the time of declaration, 'a' variable is assigned to 'b'. If we do not assign at the time of declaration, then the code would look like:



```
int &b;
&b=a;
```

The above code will throw a compile-time error as 'a' is not assigned at the time of declaration.

Output

```
value of a is 10
```

Reassignment

It cannot be reassigned means that the reference variable cannot be modified.

```
#include <iostream>
using namespace std;
int main()
{
    int x=11; // variable initialization
    int z=67;
    int &y=x; // y reference to x
    int &y=z; // y reference to z, but throws a compile-time error.
    return 0;
}
```

In the above code, 'y' reference variable is referring to 'x' variable, and then 'z' is assigned to 'y'. But this reassignment is not possible with the reference variable, so it throws a compile-time error.



Compile-time error

```
main.cpp: In function 'int main()':
main.cpp:18:9: error: redeclaration of 'int& y'
int &y=z; // y reference to z, but throws a compile-time error.
^
main.cpp:17:9: note: 'int& y' previously declared here
int &y=x; // y reference to x
^
```

Function Parameters

References can also be passed as a function parameter. It does not create a copy of the argument and behaves as an alias for a parameter. It enhances the performance as it does not create a copy of the argument.

Let's understand through a simple example.

```
#include <iostream>
using namespace std;
int main()
{
    int a=9; // variable initialization
    int b=10; // variable initialization
    swap(a, b); // function calling
    std::cout << "value of a is :" << a << std::endl;
    std::cout << "value of b is :" << b << std::endl;
    return 0;
}
void swap(int &p, int &q) // function definition
{
    int temp; // variable declaration
    temp=p;
    p=q;
    q=temp;
}
```

In the above code, we are swapping the values of 'a' and 'b'. We have passed the variables 'a' and 'b' to the swap() function. In swap() function, 'p' is referring to 'a' and 'q' is referring to 'b'. When we swap the values of 'p' and 'q' means that the values of 'a' and 'b' are also swapped.

Output

```
value of a is :10
value of b is :9
```

References as shortcuts

With the help of references, we can easily access the nested data.

```
#include <iostream>
```

```
using namespace std;  
struct profile  
{  
    int id;  
};  
struct employee  
{  
    profile p;  
};  
int main()  
{  
    employee e;  
    int &ref=e.p.id;  
    ref=34;  
    std::cout << e.p.id << std::endl;  
}
```

In the above code, we are trying to access the 'id' of the profile struct of the employee. We generally access this member by using the statement `e.p.id`, but this would be a tedious task if we have multiple access to this member. To avoid this situation, we create a reference variable, i.e., `ref`, which is another name of '`e.p.id`'.

Output

```
34
```

← Prev

Next →

C++ Reference vs Pointer

C++ reference and pointer seem to be similar, but there are some differences that exist between them. A reference is a variable which is another name of the existing variable, while the pointer is variable that stores the address of another variable.

What is Reference?

A reference is a **variable** that is referred to as another name for an already existing variable. The reference of a variable is created by storing the address of another variable.

A reference variable can be considered as a constant pointer with automatic indirection. Here, automatic indirection means that the compiler automatically applies the indirection operator (*) .

Example of reference:

```
int &a = i;
```

In the above declaration, 'a' is an alias name for 'i' variable. We can also refer to the 'i' variable through 'a' variable also.

Let's understand through an example.

```
#include <iostream>
using namespace std;
int main()
{
    int i=8; // variable initialization
    int &a=i; // creating a reference variable
    cout<<"The value of 'i' variable is :"<<a;
    return 0;
}
```

In the above code, we have created a reference variable, i.e., 'a' for 'i' variable. After creating a reference variable, we can access the value of 'i' with the help of 'a' variable.

What is Pointer?

A pointer is a variable that contains the address of another variable. It can be dereferenced with the help of (*) **operator** to access the memory location to which the pointer points.

Differences between Reference and Pointer

The following are the differences between reference and pointer:

- **Definition**

A reference variable is another name for an already existing variable. It is mainly used in '**pass by reference**' where the reference variable is passed as a parameter to the function and the function to which this variable is passed works on the original copy of the variable.

Let's understand through a simple example.

```
#include <iostream>
using namespace std;
void func(int &);
int main()
{
    int a=10;
```

```

std::cout << "Value of 'a' is :" <<a<< std::endl;
func(a);
std::cout << "Now value of 'a' is :" <<a<< std::endl;
return 0;
}

void func(int &m)
{
    m=8;
}

```

Output:

```

Value of 'a' is :10
Now value of 'a' is :8

```

Whereas, **Pointer** is a variable that stores the address of another variable. It makes the programming easier as it holds the memory address of some variable.

- o **Declaration**



We can declare a reference variable by adding a '&' symbol before a variable. If this symbol is used in the expression, then it will be treated as an address operator.

Before using a pointer variable, we should declare a pointer variable, and this variable is created by adding a '*' operator before a variable.

- o **Reassignment**

We cannot reassign the reference variable. Now, we take a simple example as given below:

```

#include <iostream>
using namespace std;
void func(int &);

int main()
{
    int i; // variable declaration
    int k; // variable declaration
    int &a=i;
    int &a=k; // error
    return 0;
}

```

The above code shows the error that multiple declarations of **int &a** are not allowed. Therefore, the above program concludes that reassignment operation is not valid for the reference variable.



Whereas, the pointers can be re-assigned. This reassignment is useful when we are working with the data structures such as linked list, trees, etc.

- o **Memory Address**

In the case of reference, both the reference and actual variable refer to the same address. The new variable will not be assigned to the reference variable until the actual variable is either deleted or goes out of the scope.

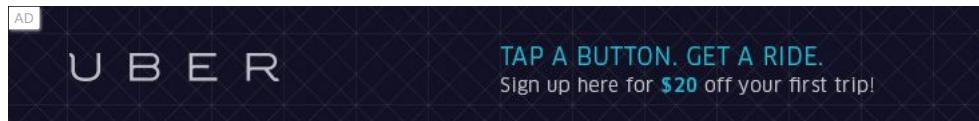
Let's understand this scenario through an example.

```
#include <iostream>
using namespace std;
void func(int &);
int main()
{
    int i;
    int &a=i;
    std::cout << "The address of 'a' variable is : " <<&a<< std::endl;
    std::cout << "The address of 'i' variable is : " <<&i<< std::endl;
    return 0;
}
```

Output:

```
The address of 'a' variable is : 0x7fff078e7e44
```

The above output shows that both the reference variable and the actual variable have the same address.



In the case of pointers, both the pointer variable and the actual variable will have different memory addresses. Let's understand this through an example.

```
#include <iostream>
using namespace std;
int main()
{
    int k;
    int *p;
    p=&k;
    cout<<"The memory address of p variable is :"<<&p;
    cout<<"\nThe memory address of k variable is :"<<&k;
    return 0;
}
```

Output:

```
The memory address of p variable is :0x7ffcc5c164b8
```

The memor

- **NULL value**

We cannot assign the NULL value to the reference variable, but the pointer variable can be assigned with a NULL value.

- **Indirection**

Pointers can have pointer to pointer offering more than one level of indirection.

```
#include <iostream>
```

```
using namespace std;
int main()
{
    int *p;
    int a=8;
    int **q;
    p=&a;
    q=&p;
    std::cout << "The value of q is : " << *q << std::endl;
    return 0;
}
```

In the above code, the pointer 'p' is pointing to variable 'a' while 'q' is a double pointer which is pointing to 'p'. Therefore, we can say that the value of 'p' would be the address of 'a' variable and the value of 'q' variable would be the address of 'p' variable.

Output:

```
The value of q is : 0x7ffd104891dc
```

In the case of References, reference to reference is not possible. If we try to do `C++ program` will throw a compile-time error

Let's understand this scenario through an example.

```
#include <iostream>
using namespace std;
int main()
{
    int a=8; // variable initialization
    int &p=a; // creating a reference variable for ?a? variable.
    int &&q=p; // reference to reference is not valid, it throws an error.
    return 0;
}
```

Output:

```
main.cpp: In function 'int main()':
main.cpp:18:10: error: cannot bind 'int' lvalue to 'int&&'
int &&q=p;
```

- o **Arithmetic Operations**

As we know that arithmetic operations can be applied to the pointers named as "**Pointer Arithmetic**", but arithmetic operations cannot be applied on the references. There is no word, i.e., Reference Arithmetic exists in **C++**.

Let's see a simple example of Pointers.

```
#include <iostream>
using namespace std;
int main()
{
    int a[]={1,2,3,4,5}; // array initialization
    int *ptr; // pointer declaration
    ptr=a; assigning base address to pointer ptr.
    cout<<"The value of *ptr is :" << *ptr;
```

```
ptr=ptr+1; // incrementing the value of ptr by 1.  
std::cout << "\nThe value of *ptr is: " << *ptr << std::endl;  
return 0;  
}
```

Output:

```
The value of *ptr is :1  
The value of *ptr is: 2
```

Let's understand the references through an example.

```
#include <iostream>  
using namespace std;  
int main()  
{  
  
    int value=90; // variable declaration  
    int &a=value; // assigning value to the reference  
    &a=&a+5 // arithmetic operation is not possible with reference variable, it throws an error.  
    return 0;  
}
```

The above code will throw a compile-time error as arithmetic operations are not allowed with references.

← Prev

Next →

AD

 [For Videos Join Our YouTube Channel: Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share

Learn Latest Tutorials

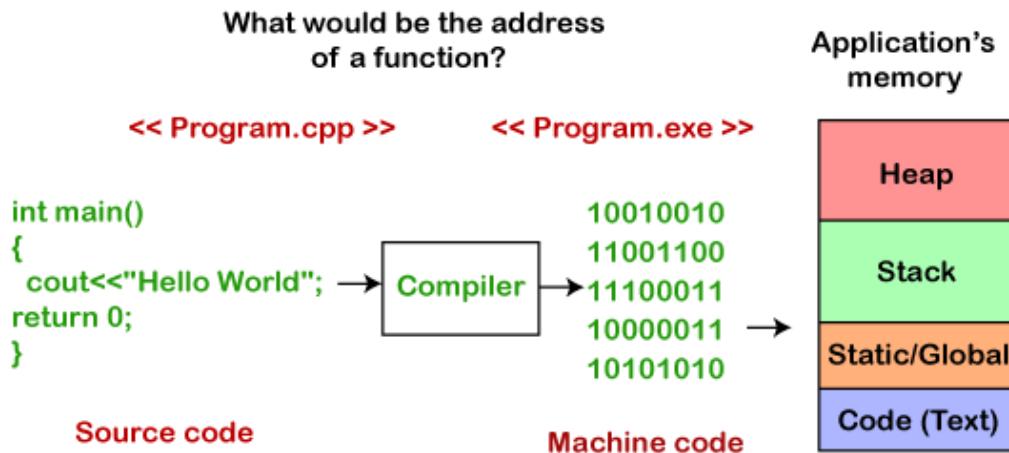
Function Pointer in C++

As we know that pointers are used to point some variables; similarly, the function pointer is a pointer used to point functions. It is basically used to store the address of a function. We can call the function by using the function pointer, or we can also pass the pointer to another function as a parameter.

They are mainly useful for event-driven applications, callbacks, and even for storing the functions in arrays.

What is the address of a function?

Function Pointers



Computer only understands the low-level language, i.e., binary form. The program we write in C++ is always in high-level language, so to convert the program into binary form, we use compiler. Compiler is a program that converts source code into an executable file. This executable file gets stored in RAM. The CPU starts the execution from the main() method, and it reads the copy in RAM but not the original file.

All the functions and machine code instructions are data. This data is a bunch of bytes, and all these bytes have some address in RAM. The function pointer contains RAM address of the first instruction of a function.

Syntax for Declaration

The following is the syntax for the declaration of a function pointer:

```
int (*FuncPtr) (int,int);
```

The above syntax is the function declaration. As functions are not simple as variables, but C++ is a type safe, so function pointers have return type and parameter list. In the above syntax, we first supply the return type, and then the name of the pointer, i.e., FuncPtr which is surrounded by the brackets and preceded by the pointer symbol, i.e., (*). After this, we have supplied the parameter list (int,int). The above function pointer can point to any function which takes two integer parameters and returns integer type value.

Address of a function

We can get the address of a function very easily. We just need to mention the name of the function, we do not need to call the function.

Let's illustrate through an example.

```
#include <iostream>
using namespace std;
int main()
{
    std::cout << "Address of a main() function is : " << &main << std::endl;
    return 0;
}
```

In the above program, we are displaying the address of a main() function. To print the address of a main() function, we have just mentioned the name of the function, there is no bracket nor parameters. Therefore, the name of the function by itself without any brackets or parameters means the address of a function.

We can use the alternate way to print the address of a function, i.e., &main.

Calling a function indirectly

We can call the function with the help of a function pointer by simply using the name of the function pointer. The syntax of calling the function through the function pointer would be similar as we do the calling of the function normally.

Let's understand this scenario through an example.

```
#include <iostream>
using namespace std;
int add(int a , int b)
```

```

{
    return a+b;
}

int main()
{
    int (*funcptr)(int,int); // function pointer declaration
    funcptr=add; // funcptr is pointing to the add function
    int sum=funcptr(5,5);
    std::cout << "value of sum is :" <<sum<< std::endl;
    return 0;
}

```

In the above program, we declare the function pointer, i.e., int (*funcptr)(int,int) and then we store the address of add() function in funcptr. This implies that funcptr contains the address of add() function. Now, we can call the add() function by using funcptr. The statement funcptr(5,5) calls the add() function, and the result of add() function gets stored in sum variable.

Output:

```

value of sum is :10

...Program finished with exit code 0
Press ENTER to exit console. []

```

Let's look at another example of function pointer.

```

#include <iostream>
using namespace std;
void printname(char *name)
{
    std::cout << "Name is :" <<name<< std::endl;
}

int main()
{
    char s[20]; // array declaration
    void (*ptr)(char*); // function pointer declaration
}

```

```

ptr=printname; // storing the address of printname in ptr.
std::cout << "Enter the name of the person: " << std::endl;
cin>>s;
cout<<s;
ptr(s); // calling printname() function
return 0;
}

```

In the above program, we define the function printname() which contains the char pointer as a parameter. We declare the function pointer, i.e., void (*ptr)(char*). The statement ptr=printname means that we are assigning the address of printname() function to ptr. Now, we can call the printname() function by using the statement ptr(s).

Output:

AD

```

Enter the name of the person:
john
john
Name is :john

...Program finished with exit code 0
Press ENTER to exit console.

```

Passing a function pointer as a parameter

The function pointer can be passed as a parameter to another function.

Let's understand through an example.

```

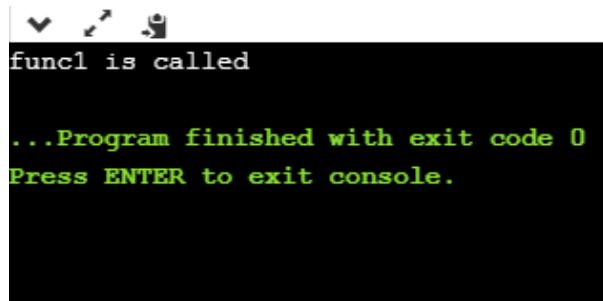
#include <iostream>
using namespace std;
void func1()
{
    cout<<"func1 is called";
}
void func2(void (*funcptr)())

```

```
{  
    funcptr();  
}  
int main()  
{  
    func2(func1);  
    return 0;  
}
```

In the above code, the func2() function takes the function pointer as a parameter. The main() method calls the func2() function in which the address of func1() is passed. In this way, the func2() function is calling the func1() indirectly.

Output:



```
func1 is called  
...Program finished with exit code 0  
Press ENTER to exit console.
```

← Prev

Next →

AD

 [For Videos Join Our Youtube Channel: Join Now](#)

What is Memory Management?

Memory management is a process of managing computer memory, assigning the memory space to the programs to improve the overall system performance.

Why is memory management required?

As we know that arrays store the homogeneous data, so most of the time, memory is allocated to the array at the declaration time. Sometimes the situation arises when the exact memory is not determined until runtime. To avoid such a situation, we declare an array with a maximum size, but some memory will be unused. To avoid the wastage of memory, we use the new operator to allocate the memory dynamically at the run time.

Memory Management Operators

In **C language**, we use the **malloc()** or **calloc()** functions to allocate the memory dynamically at run time, and **free()** function is used to deallocate the dynamically allocated memory. **C++** also supports these functions, but C++ also defines unary operators such as **new** and **delete** to perform the same tasks, i.e., allocating and freeing the memory.

New operator

A **new** operator is used to create the object while a **delete** operator is used to delete the object. When the object is created by using the new operator, then the object will exist until we explicitly use the delete operator to delete the object. Therefore, we can say that the lifetime of the object is not related to the block structure of the program.

Syntax

```
pointer_variable = new data-type
```

The above syntax is used to create the object using the new operator. In the above syntax, '**pointer_variable**' is the name of the pointer variable, '**new**' is the operator, and '**data-type**' defines the type of the data.

Example 1:

```
int *p;
```

```
p = new int;
```

In the above example, 'p' is a pointer of type int.

Example 2:

```
float *q;
q = new float;
```

In the above example, 'q' is a pointer of type float.

In the above case, the declaration of pointers and their assignments are done separately. We can also combine these two statements as follows:

```
int *p = new int;
float *q = new float;
```

Assigning a value to the newly created object

Two ways of assigning values to the newly created object:

- We can assign the value to the newly created object by simply using the assignment operator. In the above case, we have created two pointers 'p' and 'q' of type int and float, respectively. Now, we assign the values as follows:

```
*p = 45;
*q = 9.8;
```

We assign 45 to the newly created int object and 9.8 to the newly created float object.

- We can also assign the values by using new operator which can be done as follows:



```
pointer_variable = new data-type(value);
```

Let's look at some examples.

```
int *p = new int(45);  
float *p = new float(9.8);
```

How to create a single dimensional array

As we know that new operator is used to create memory space for any data-type or even user-defined data type such as an array, structures, unions, etc., so the syntax for creating a one-dimensional array is given below:

```
pointer-variable = new data-type[size];
```

Examples:

```
int *a1 = new int[8];
```

In the above statement, we have created an array of type int having a size equal to 8 where p[0] refers first element, p[1] refers the first element, and so on.

Delete operator

When memory is no longer required, then it needs to be deallocated so that the memory can be used for another purpose. This can be achieved by using the delete operator, as shown below:

AD



VIDEO STREAMING FOR THE AWAKENED MIND

GET STARTED NOW

```
delete pointer_variable;
```

In the above statement, '**delete**' is the operator used to delete the existing object, and '**pointer_variable**' is the name of the pointer variable.

In the previous case, we have created two pointers 'p' and 'q' by using the new operator, and can be deleted by using the following statements:

```
delete p;  
delete q;
```

The dynamically allocated array can also be removed from the memory space by using the following syntax:

```
delete [size] pointer_variable;
```

In the above statement, we need to specify the size that defines the number of elements that are required to be freed. The drawback of this syntax is that we need to remember the size of the array. But, in recent versions of C++, we do not need to mention the size as follows:

```
delete [] pointer_variable;
```

Let's understand through a simple example:



```
#include <iostream>  
using namespace std  
int main()  
{  
    int size; // variable declaration  
    int *arr = new int[size]; // creating an array  
    cout<<"Enter the size of the array : ";  
    std::cin >> size; //  
    cout<<"\nEnter the element : ";  
    for(int i=0;i<size;i++) // for loop  
    {  
        cin>>arr[i];  
    }  
    cout<<"\nThe elements that you have entered are :";  
    for(int i=0;i<size;i++) // for loop  
    {  
        cout<<arr[i]<<",";
```

```
}

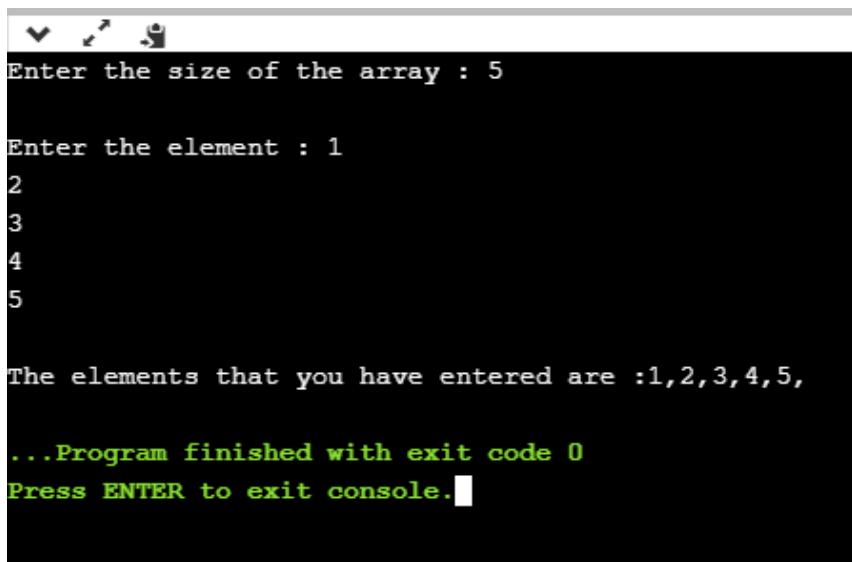
delete arr; // deleting an existing array.

return 0;

}
```

In the above code, we have created an array using the new operator. The above program will take the user input for the size of an array at the run time. When the program completes all the operations, then it deletes the object by using the statement **delete arr**.

Output



```
Enter the size of the array : 5

Enter the element : 1
2
3
4
5

The elements that you have entered are :1,2,3,4,5,
...Program finished with exit code 0
Press ENTER to exit console.
```

Advantages of the new operator

The following are the advantages of the new operator over malloc() function:

- It does not use the sizeof() operator as it automatically computes the size of the data object.
- It automatically returns the correct data type pointer, so it does not need to use the typecasting.
- Like other operators, the new and delete operator can also be overloaded.
- It also allows you to initialize the data object while creating the memory space for the object.

malloc() vs new in C++

Both the **malloc()** and new in C++ are used for the same purpose. They are used for allocating memory at the runtime. But, malloc() and new have different syntax. The main difference between the malloc() and new is that the new is an operator while malloc() is a standard library function that is predefined in a **stdlib** header file.

What is new?

The new is a memory allocation operator, which is used to allocate the memory at the runtime. The memory initialized by the new operator is allocated in a heap. It returns the starting address of the memory, which gets assigned to the variable. The functionality of the new **operator in C++** is similar to the malloc() function, which was used in the **C programming language**. C++ is compatible with the malloc() function also, but the new operator is mostly used because of its advantages.

Syntax of new operator

```
type variable = new type(parameter_list);
```

In the above syntax

type: It defines the datatype of the variable for which the memory is allocated by the new operator.

variable: It is the name of the variable that points to the memory.

parameter_list: It is the list of values that are initialized to a variable.

The new operator does not use the sizeof() operator to allocate the memory. It also does not use the resize as the new operator allocates sufficient memory for an object. It is a construct that calls the constructor at the time of declaration to initialize an object.

As we know that the new operator allocates the memory in a heap; if the memory is not available in a heap and the new operator tries to allocate the memory, then the exception is thrown. If our code is not able to handle the exception, then the program will be terminated abnormally.

Let's understand the new operator through an example.

```
#include <iostream>
using namespace std;
int main()
{
```

```

int *ptr; // integer pointer variable declaration
ptr=new int; // allocating memory to the pointer variable ptr.
std::cout << "Enter the number : " << std::endl;
std::cin >> *ptr;
std::cout << "Entered number is " << *ptr << std::endl;
return 0;
}

```

Output:

```

Enter the number :
15
Entered number is 15

...Program finished with exit code 0
Press ENTER to exit console.□

```

What is malloc()?

A malloc() is a function that allocates memory at the runtime. This function returns the void pointer, which means that it can be assigned to any pointer type. This void pointer can be further typecast to get the pointer that points to the memory of a specified type.

The syntax of the malloc() function is given below:

```
type variable_name = (type *)malloc(sizeof(type));
```

where,

type: it is the datatype of the variable for which the memory has to be allocated.

variable_name: It defines the name of the variable that points to the memory.

(type*): It is used for typecasting so that we can get the pointer of a specified type that points to the memory.



sizeof(): The sizeof() operator is used in the malloc() function to obtain the memory size required for the allocation.

Note: The malloc() function returns the void pointer, so typecasting is required to assign a different type to the pointer. The sizeof() operator is required in the malloc() function as the malloc() function returns the raw memory, so the sizeof() operator will tell the malloc() function how much memory is required for the allocation.

If the sufficient memory is not available, then the memory can be resized using realloc() function. As we know that all the dynamic memory requirements are fulfilled using heap memory, so malloc() function also allocates the memory in a heap and returns the pointer to it. The heap memory is very limited, so when our code starts execution, it marks the memory in use, and when our code completes its task, then it frees the memory by using the free() function. If the sufficient memory is not available, and our code tries to access the memory, then the malloc() function returns the NULL pointer. The memory which is allocated by the malloc() function can be deallocated by using the free() function.

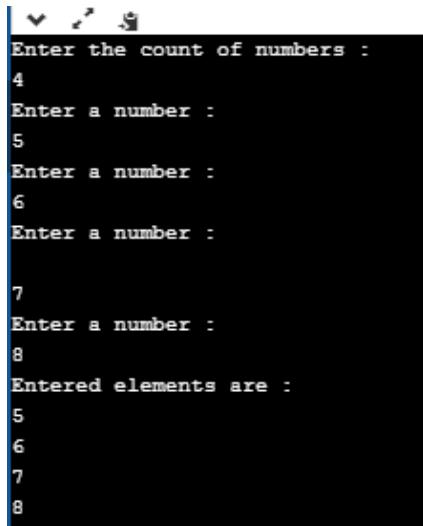
Let's understand through an example.

```
#include <iostream>
#include<stdlib.h>
using namespace std;

int main()
{
    int len; // variable declaration
    std::cout << "Enter the count of numbers :" << std::endl;
    std::cin >> len;
    int *ptr; // pointer variable declaration
    ptr=(int*) malloc(sizeof(int)*len); // allocating memory to the pointer variable
    for(int i=0;i<len;i++)
    {
        std::cout << "Enter a number :" << std::endl;
        std::cin >> *(ptr+i);
    }
    std::cout << "Entered elements are :" << std::endl;
    for(int i=0;i<len;i++)
}
```

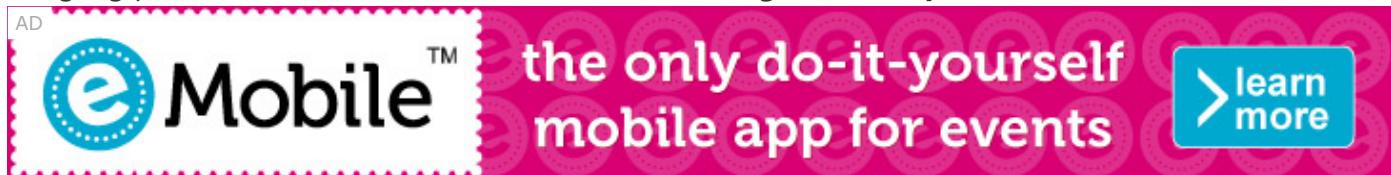
```
{  
    std::cout << *(ptr+i) << std::endl;  
}  
free(ptr);  
return 0;  
}
```

Output:



```
Enter the count of numbers :  
4  
Enter a number :  
5  
Enter a number :  
6  
Enter a number :  
7  
Enter a number :  
8  
Entered elements are :  
5  
6  
7  
8
```

If we do not use the **free()** function at the correct place, then it can lead to the cause of the dangling pointer. **Let's understand this scenario through an example.**



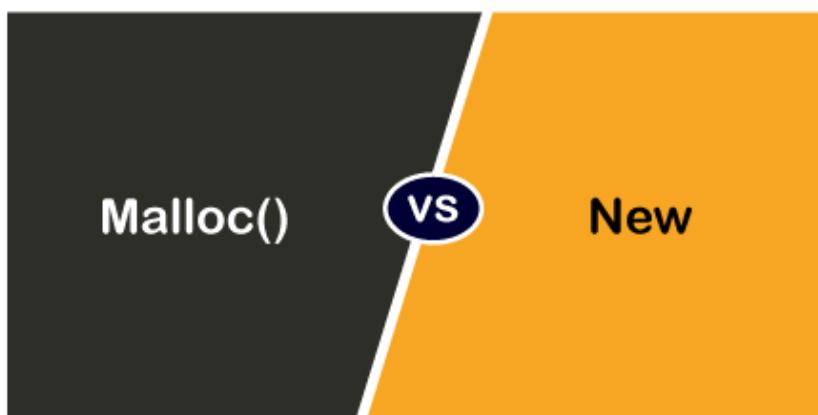
```
#include <iostream>  
#include<stdlib.h>  
using namespace std;  
int *func()  
{  
    int *p;  
    p=(int*) malloc(sizeof(int));  
    free(p);  
    return p;  
}  
int main()  
{
```

```
int *ptr;  
ptr=func();  
free(ptr);  
return 0;  
}
```

In the above code, we are calling the func() function. The func() function returns the integer pointer. Inside the func() function, we have declared a *p pointer, and the memory is allocated to this pointer variable using malloc() function. In this case, we are returning the pointer whose memory is already released. The ptr is a dangling pointer as it is pointing to the released memory location. Or we can say ptr is referring to that memory which is not pointed by the pointer.

Till now, we get to know about the new operator and the malloc() function. Now, we will see the differences between the new operator and the malloc() function.

Differences between the malloc() and new



- The new operator constructs an object, i.e., it calls the constructor to initialize an object while **malloc()** function does not call the constructor. The new operator invokes the constructor, and the delete operator invokes the destructor to destroy the object. This is the biggest difference between the malloc() and new.
- The new is an operator, while malloc() is a predefined function in the stdlib header file.
- The operator new can be overloaded while the malloc() function cannot be overloaded.
- If the sufficient memory is not available in a heap, then the new operator will throw an exception while the malloc() function returns a NULL pointer.
- In the new operator, we need to specify the number of objects to be allocated while in malloc() function, we need to specify the number of bytes to be allocated.
- In the case of a new operator, we have to use the delete operator to deallocate the memory. But in the case of malloc() function, we have to use the free() function to deallocate the

memory.

Syntax of new operator

```
type reference_variable = new type name;
```

where,

type: It defines the data type of the reference variable.

reference_variable: It is the name of the pointer variable.

new: It is an operator used for allocating the memory.

type name: It can be any basic data type.

For example,

```
int *p;  
p = new int;
```

In the above statements, we are declaring an integer pointer variable. The statement **p = new int;** allocates the memory space for an integer variable.

Syntax of malloc() is given below:

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

ptr: It is a pointer variable.

data_type: It can be any basic data type.

For example,

```
int *p;  
p = (int *) malloc(sizeof(int))
```

The above statement will allocate the memory for an integer variable in a heap, and then stores the address of the reserved memory in 'p' variable.

- On the other hand, the memory allocated using malloc() function can be deallocated using the free() function.
- Once the memory is allocated using the new operator, then it cannot be resized. On the other hand, the memory is allocated using malloc() function; then, it can be reallocated using realloc() function.
- The execution time of new is less than the malloc() function as new is a construct, and malloc is a function.
- The new operator does not return the separate pointer variable; it returns the address of the newly created object. On the other hand, the malloc() function returns the void pointer which can be further typecast in a specified type.



← Prev

Next →

AD

[For Videos Join Our Youtube Channel: Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

free vs delete in C++

In this topic, we are going to learn about the **free()** function and **delete** operator in C++.

free() function

The **free()** function is used in C++ to de-allocate the memory dynamically. It is basically a library function used in C++, and it is defined in **stdlib.h** header file. This library function is used when the pointers either pointing to the memory allocated using **malloc()** function or Null pointer.

Syntax of free() function

Suppose we have declared a pointer 'ptr', and now, we want to de-allocate its memory:

```
free(ptr);
```

The above syntax would de-allocate the memory of the pointer variable 'ptr'.

free() parameters

In the above syntax, ptr is a parameter inside the **free()** function. The ptr is a pointer pointing to the memory block allocated using **malloc()**, **calloc()** or **realloc** function. This pointer can also be null or a pointer allocated using **malloc** but not pointing to any other memory block.

- If the pointer is null, then the **free()** function will not do anything.
- If the pointer is allocated using **malloc**, **calloc**, or **realloc**, but not pointing to any memory block then this function will cause undefined behavior.

free() Return Value

The **free()** function does not return any value. Its main function is to free the memory.

Let's understand through an example.

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

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

```
ptr = (int*) malloc(5*sizeof(int));
cout << "Enter 5 integer" << endl;

for (int i=0; i<5; i++)
{
    // *(ptr+i) can be replaced by ptr[i]
    cin >>ptr[i];
}

cout << endl << "User entered value" << endl;

for (int i=0; i<5; i++)
{
    cout <<*(ptr+i) << " ";
}
free(ptr);

/* prints a garbage value after ptr is free */
cout << "Garbage Value" << endl;

for (int i=0; i<5; i++)
{
    cout << *(ptr+i)<< " ";
}
return 0;
}
```

The above code shows how free() function works with malloc(). First, we declare integer pointer *ptr, and then we allocate the memory to this pointer variable by using malloc() function. Now, ptr is pointing to the uninitialized memory block of 5 integers. After allocating the memory, we use the free() function to destroy this allocated memory. When we try to print the value, which is pointed by the ptr, we get a garbage value, which means that memory is de-allocated.

Output

```

1
2
3
4
5

User entered value
1 2 3 4 5 Garbage Value
0 0 3 4 5

```

Let's see how free() function works with a calloc.

```

#include <iostream>
#include <cstdlib>
using namespace std;
int main()
{
    float *ptr; // float pointer declaration
    ptr=(float*)calloc(1,sizeof(float));
    *ptr=6.7;
    std::cout << "The value of *ptr before applying the free() function : " << *ptr << std::endl;
    free(ptr);
    std::cout << "The value of *ptr after applying the free() function : " << *ptr << std::endl;
    return 0;
}

```

In the above example, we can observe that free() function works with a calloc(). We use the calloc() function to allocate the memory block to the float pointer ptr. We have assigned a memory block to the ptr that can have a single float type value.

Output:

```

The value of *ptr before applying the free() function : 6.7
The value of *ptr after applying the free() function : 0

...Program finished with exit code 0
Press ENTER to exit console.

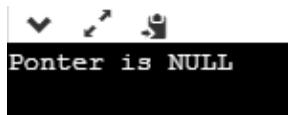
```

Let's look at another example.

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main()
{
    int *ptr1=NULL;
    int *ptr2;
    int x=9;
    ptr2=&x;
    if(ptr1)
    {
        std::cout << "Pointer is not Null" << std::endl;
    }
    else
    {
        cout<<"Ponter is NULL";
    }
    free(ptr1);
    //free(ptr2); // If this statement is executed, then it gives a runtime error.
    return 0;
}
```

The above code shows how free() function works with a NULL pointer. We have declared two pointers, i.e., ptr1 and ptr2. We assign a NULL value to the pointer ptr1 and the address of x variable to pointer ptr2. When we apply the free(ptr1) function to the ptr1, then the memory block assigned to the ptr is successfully freed. The statement free(ptr2) shows a runtime error as the memory block assigned to the ptr2 is not allocated using malloc or calloc function.

Output



Delete operator

It is an operator used in C++ programming language, and it is used to de-allocate the memory dynamically. This operator is mainly used either for those pointers which are allocated using a new operator or NULL pointer.

Syntax

```
delete pointer_name
```

For example, if we allocate the memory to the pointer using the new operator, and now we want to delete it. To delete the pointer, we use the following statement:



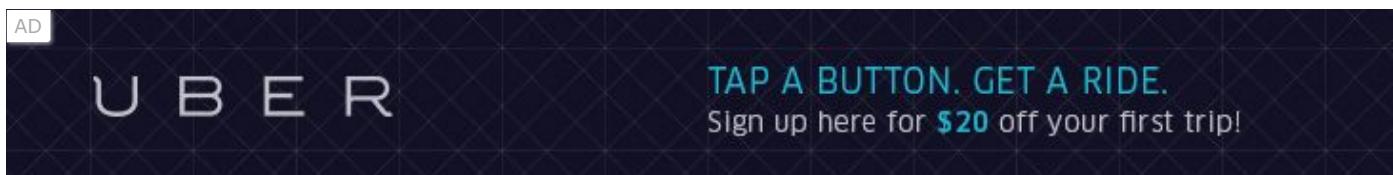
```
delete p;
```

To delete the array, we use the statement as given below:

```
delete [] p;
```

Some important points related to delete operator are:

- It is either used to delete the array or non-array objects which are allocated by using the new keyword.
- To delete the array or non-array object, we use `delete[]` and `delete` operator, respectively.
- The new keyword allocated the memory in a heap; therefore, we can say that the delete operator always de-allocates the memory from the heap
- It does not destroy the pointer, but the value or the memory block, which is pointed by the pointer is destroyed.



Let's look at the simple example of a delete operator.

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

int main()
```

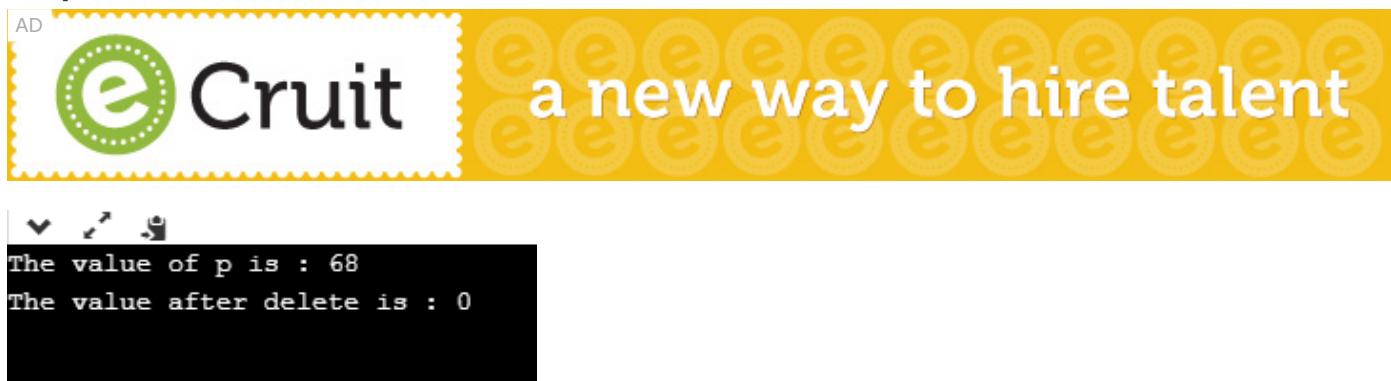
```

{
int *ptr;
ptr=new int;
*ptr=68;
std::cout << "The value of p is : " <<*ptr<< std::endl;
delete ptr;
std::cout << "The value after delete is : " <<*ptr<< std::endl;
return 0;
}

```

In the above code, we use the new operator to allocate the memory, so we use the delete operator to destroy the memory block, which is pointed by the pointer ptr.

Output



Let's see how delete works with an array of objects.

```

#include <iostream>
using namespace std;
int main()
{
    int *ptr=new int[5]; // memory allocation using new operator.
    std::cout << "Enter 5 integers:" << std::endl;
    for(int i=1;i<=5;i++)
    {
        cin>>ptr[i];
    }
    std::cout << "Entered values are:" << std::endl;
    for(int i=1;i<=5;i++)
    {
        cout<<*(ptr+i)<<endl;
    }
}

```

```
}

delete[] ptr; // deleting the memory block pointed by the ptr.

std::cout << "After delete, the garbage value:" << std::endl;

for(int i=1;i<=5;i++)

{

    cout<<*(ptr+i)<<endl;

}

return 0;

}
```

Output

```
Enter 5 integers:
4
5
3
1
2
Entered values are:
4
5
3
1
2
After delete, the garbage value:
0
5
3
1
2
```

Differences between delete and free()

The following are the differences between delete and free() in C++ are:

- The delete is an operator that de-allocates the memory dynamically while the free() is a function that destroys the memory at the runtime.
- The delete operator is used to delete the pointer, which is either allocated using new operator or a NULL pointer, whereas the free() function is used to delete the pointer that is either allocated using malloc(), calloc() or realloc() function or NULL pointer.
- When the delete operator destroys the allocated memory, then it calls the destructor of the class in C++, whereas the free() function does not call the destructor; it only frees the memory from the heap.

- o The delete() operator is faster than the free() function.

[← Prev](#)[Next →](#)

AD

 [For Videos Join Our Youtube Channel: Join Now](#)

Feedback

- o Send your Feedback to feedback@javatpoint.com

Help Others, Please Share



Learn Latest Tutorials

 [Splunk tutorial](#)

Splunk

 [SPSS tutorial](#)

SPSS

 [Swagger tutorial](#)

Swagger

 [T-SQL tutorial](#)

Transact-SQL

 [Tumblr tutorial](#)

Tumblr

 [React tutorial](#)

ReactJS

 [Regex tutorial](#)

Regex

 [Reinforcement learning tutorial](#)

learning

C++ OOPs Concepts

The major purpose of C++ programming is to introduce the concept of object orientation to the C programming language.

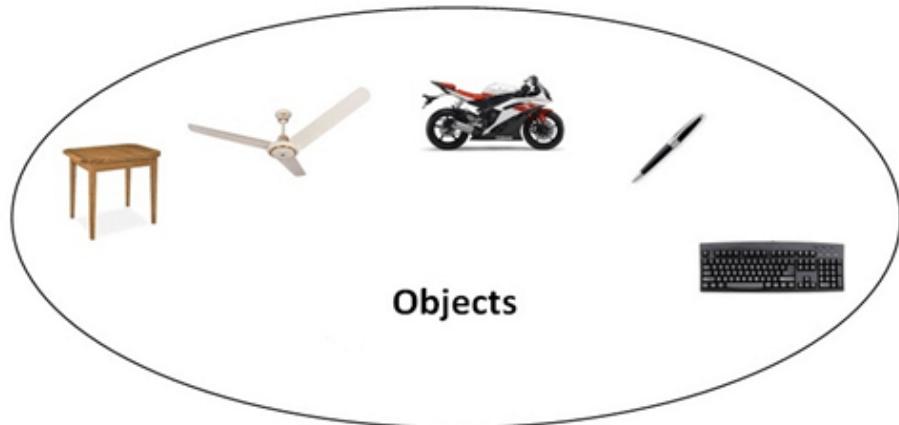
Object Oriented Programming is a paradigm that provides many concepts such as **inheritance, data binding, polymorphism etc.**

The programming paradigm where everything is represented as an object is known as truly object-oriented programming language. **Smalltalk** is considered as the first truly object-oriented programming language.

OOPs (Object Oriented Programming System)

Object means a real word entity such as pen, chair, table etc. **Object-Oriented Programming** is a methodology or paradigm to design a program using classes and objects. It simplifies the software development and maintenance by providing some concepts:

- Object
- Class
- Inheritance
- Polymorphism
- Abstraction
- Encapsulation



Object

Any entity that has state and behavior is known as an object. For example: chair, pen, table, keyboard, bike etc. It can be physical and logical.

Class

Collection of objects is called class. It is a logical entity.

A Class in C++ is the foundational element that leads to Object-Oriented programming. A class instance must be created in order to access and use the user-defined data type's data members and member functions. An object's class acts as its blueprint. Take the class of cars as an example. Even if

different names and brands may be used for different cars, all of them will have some characteristics in common, such as four wheels, a speed limit, a range of miles, etc. In this case, the class of car is represented by the wheels, the speed limitations, and the mileage.

Inheritance

When one object acquires all the properties and behaviours of parent object i.e. known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.

1. Sub class - Subclass or Derived Class refers to a class that receives properties from another class.
2. Super class - The term "Base Class" or "Super Class" refers to the class from which a subclass inherits its properties.
3. Reusability - As a result, when we wish to create a new class, but an existing class already contains some of the code we need, we can generate our new class from the old class thanks to inheritance. This allows us to utilize the fields and methods of the pre-existing class.

Polymorphism

When one task is performed by different ways i.e. known as polymorphism. For example: to convince the customer differently, to draw something e.g. shape or rectangle etc.

Different situations may cause an operation to behave differently. The type of data utilized in the operation determines the behavior.

Abstraction

Hiding internal details and showing functionality is known as abstraction. Data abstraction is the process of exposing to the outside world only the information that is absolutely necessary while concealing implementation or background information. For example: phone call, we don't know the internal processing.

In C++, we use abstract class and interface to achieve abstraction.

Encapsulation

Binding (or wrapping) code and data together into a single unit is known as encapsulation. For example: capsule, it is wrapped with different medicines.

Encapsulation is typically understood as the grouping of related pieces of information and data into a single entity. Encapsulation is the process of tying together data and the functions that work with it in object-oriented programming. Take a look at a practical illustration of encapsulation: at a company, there are various divisions, including the sales division, the finance division, and the

accounts division. All financial transactions are handled by the finance sector, which also maintains records of all financial data. In a similar vein, the sales section is in charge of all tasks relating to sales and maintains a record of each sale. Now, a scenario could occur when, for some reason, a financial official requires all the information on sales for a specific month. Under the umbrella term "sales section," all of the employees who can influence the sales section's data are grouped together. Data abstraction or concealing is another side effect of encapsulation. In the same way that encapsulation hides the data. In the aforementioned example, any other area cannot access any of the data from any of the sections, such as sales, finance, or accounts.

Dynamic Binding - In dynamic binding, a decision is made at runtime regarding the code that will be run in response to a function call. For this, C++ supports virtual functions.

Advantage of OOPs over Procedure-oriented programming language

1. OOPs makes development and maintenance easier where as in Procedure-oriented programming language it is not easy to manage if code grows as project size grows.
2. OOPs provide data hiding whereas in Procedure-oriented programming language a global data can be accessed from anywhere.
3. OOPs provide ability to simulate real-world event much more effectively. We can provide the solution of real word problem if we are using the Object-Oriented Programming language.

Why do we need oops in C++?

There were various drawbacks to the early methods of programming, as well as poor performance. The approach couldn't effectively address real-world issues since, similar to procedural-oriented programming, you couldn't reuse the code within the program again, there was a difficulty with global data access, and so on.

With the use of classes and objects, object-oriented programming makes code maintenance simple. Because inheritance allows for code reuse, the program is simpler because you don't have to write the same code repeatedly. Data hiding is also provided by ideas like encapsulation and abstraction.

AD

AD

Why is C++ a partial oop?

The object-oriented features of the C language were the primary motivation behind the construction of the C++ language.

The C++ programming language is categorized as a partial object-oriented programming language despite the fact that it supports OOP concepts, including classes, objects, inheritance, encapsulation, abstraction, and polymorphism.

- 1) The main function must always be outside the class in C++ and is required. This means that we may do without classes and objects and have a single main function in the application.

It is expressed as an object in this case, which is the first time Pure OOP has been violated.

- 2) Global variables are a feature of the C++ programming language that can be accessed by any other object within the program and are defined outside of it. Encapsulation is broken here. Even though C++ encourages encapsulation for classes and objects, it ignores it for global variables.

AD

Overloading

Polymorphism also has a subset known as overloading. An existing operator or function is said to be overloaded when it is forced to operate on a new data type.

Conclusion

You will have gained an understanding of the need for object-oriented programming, what C++ OOPs are, and the fundamentals of OOPs, such as polymorphism, inheritance, encapsulation, etc., after reading this course on OOPS Concepts in C++. Along with instances of polymorphism and inheritance, you also learned about the benefits of C++ OOPs.

← Prev

Next →

C++ Object and Class

Since C++ is an object-oriented language, program is designed using objects and classes in C++.

C++ Object

In C++, Object is a real world entity, for example, chair, car, pen, mobile, laptop etc.

In other words, object is an entity that has state and behavior. Here, state means data and behavior means functionality.

Object is a runtime entity, it is created at runtime.

Object is an instance of a class. All the members of the class can be accessed through object.

Let's see an example to create object of student class using s1 as the reference variable.

```
Student s1; //creating an object of Student
```

In this example, Student is the type and s1 is the reference variable that refers to the instance of Student class.

C++ Class

In C++, class is a group of similar objects. It is a template from which objects are created. It can have fields, methods, constructors etc.

Let's see an example of C++ class that has three fields only.

```
class Student
{
    public:
        int id; //field or data member
        float salary; //field or data member
        String name;//field or data member
}
```

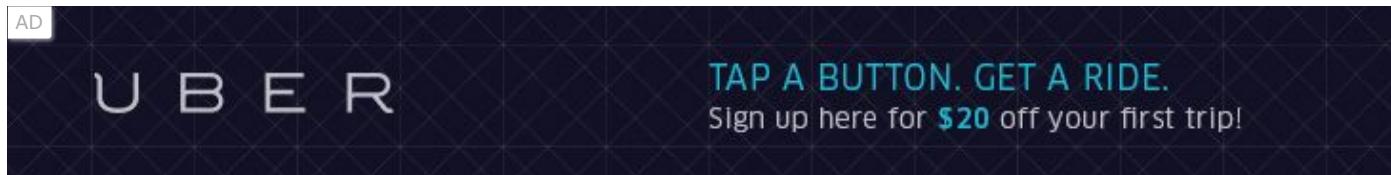
C++ Object and Class Example

Let's see an example of class that has two fields: id and name. It creates instance of the class, initializes the object and prints the object value.

```
#include <iostream>
using namespace std;
class Student {
public:
    int id;//data member (also instance variable)
    string name;//data member(also instance variable)
};
int main() {
    Student s1; //creating an object of Student
    s1.id = 201;
    s1.name = "Sonoo Jaiswal";
    cout<<s1.id<<endl;
    cout<<s1.name<<endl;
    return 0;
}
```

Output:

```
201
Sonoo Jaiswal
```



C++ Class Example: Initialize and Display data through method

Let's see another example of C++ class where we are initializing and displaying object through method.

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

```

class Student {
    public:
        int id;//data member (also instance variable)
        string name;//data member(also instance variable)
        void insert(int i, string n)
        {
            id = i;
            name = n;
        }
        void display()
        {
            cout<<id<< " " <<name<<endl;
        }
};

int main(void) {
    Student s1; //creating an object of Student
    Student s2; //creating an object of Student
    s1.insert(201, "Sonoo");
    s2.insert(202, "Nakul");
    s1.display();
    s2.display();
    return 0;
}

```

Output:

```

201 Sonoo
202 Nakul

```

C++ Class Example: Store and Display Employee Information

Let's see another example of C++ class where we are storing and displaying employee information using method.

```

#include <iostream>
using namespace std;
class Employee {

```

public:

```
int id;//data member (also instance variable)
string name;//data member(also instance variable)
float salary;
void insert(int i, string n, float s)
{
    id = i;
    name = n;
    salary = s;
}
void display()
{
    cout<<id<< " "<<name<< " "<<salary<<endl;
}
};

int main(void) {
    Employee e1; //creating an object of Employee
    Employee e2; //creating an object of Employee
    e1.insert(201, "Sonoo",990000);
    e2.insert(202, "Nakul", 29000);
    e1.display();
    e2.display();
    return 0;
}
```

Output:

```
201 Sonoo 990000
202 Nakul 29000
```

← Prev

Next →

C++ Constructor

In C++, constructor is a special method which is invoked automatically at the time of object creation. It is used to initialize the data members of new object generally. The constructor in C++ has the same name as class or structure.

In brief, A particular procedure called a constructor is called automatically when an object is created in C++. In general, it is employed to create the data members of new things. In C++, the class or structure name also serves as the constructor name. When an object is completed, the constructor is called. Because it creates the values or gives data for the thing, it is known as a constructor.

The Constructors prototype looks like this:

```
<class-name> (list-of-parameters);
```

The following syntax is used to define the class's constructor:

```
<class-name> (list-of-parameters) { // constructor definition }
```

The following syntax is used to define a constructor outside of a class:

```
<class-name>::<class-name> (list-of-parameters){ // constructor definition}
```

Constructors lack a return type since they don't have a return value.

There can be two types of constructors in C++.

- Default constructor
- Parameterized constructor

C++ Default Constructor

A constructor which has no argument is known as default constructor. It is invoked at the time of creating object.

Let's see the simple example of C++ default Constructor.

```
#include <iostream>
```

```
using namespace std;
class Employee
{
public:
    Employee()
    {
        cout<<"Default Constructor Invoked"<<endl;
    }
};

int main(void)
{
    Employee e1; //creating an object of Employee
    Employee e2;
    return 0;
}
```

Output:

```
Default Constructor Invoked
Default Constructor Invoked
```

C++ Parameterized Constructor

A constructor which has parameters is called parameterized constructor. It is used to provide different values to distinct objects.

Let's see the simple example of C++ Parameterized Constructor.

```
#include <iostream>
using namespace std;
class Employee {
public:
    int id;//data member (also instance variable)
    string name;//data member(also instance variable)
    float salary;
    Employee(int i, string n, float s)
    {
```

```

    id = i;
    name = n;
    salary = s;
}
void display()
{
    cout<<id<<" "<<name<<" "<<salary<<endl;
}
};

int main(void)
{
    Employee e1 =Employee(101, "Sonoo", 890000); //creating an object of Employee
    Employee e2=Employee(102, "Nakul", 59000);
    e1.display();
    e2.display();
    return 0;
}

```

Output:

```

101 Sonoo 890000
102 Nakul 59000

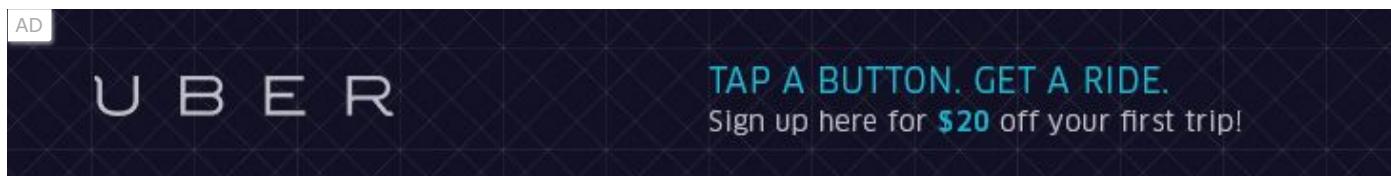
```

What distinguishes constructors from a typical member function?

1. Constructor's name is the same as the class's
2. Default There isn't an input argument for constructors. However, input arguments are available for copy and parameterized constructors.
3. There is no return type for constructors.
4. An object's constructor is invoked automatically upon creation.
5. It must be shown in the classroom's open area.
6. The C++ compiler creates a default constructor for the object if a constructor is not specified (expects any parameters and has an empty body).

By using a practical example, let's learn about the various constructor types in C++. Imagine you visited a store to purchase a marker. What are your alternatives if you want to buy a marker? For the first one, you ask a store to give you a marker, given that you didn't specify the brand name or

colour of the marker you wanted, simply asking for one amount to a request. So, when we just said, "I just need a marker," he would hand us whatever the most popular marker was in the market or his store. The default constructor is exactly what it sounds like! The second approach is to go into a store and specify that you want a red marker of the XYZ brand. He will give you that marker since you have brought up the subject. The parameters have been set in this instance thus. And a parameterized constructor is exactly what it sounds like! The third one requires you to visit a store and declare that you want a marker that looks like this (a physical marker on your hand). The shopkeeper will thus notice that marker. He will provide you with a new marker when you say all right. Therefore, make a copy of that marker. And that is what a copy constructor does!



What are the characteristics of a constructor?

1. The constructor has the same name as the class it belongs to.
2. Although it is possible, constructors are typically declared in the class's public section. However, this is not a must.
3. Because constructors don't return values, they lack a return type.
4. When we create a class object, the constructor is immediately invoked.
5. Overloaded constructors are possible.
6. Declaring a constructor virtual is not permitted.
7. One cannot inherit a constructor.
8. Constructor addresses cannot be referenced to.
9. When allocating memory, the constructor makes implicit calls to the new and delete operators.

What is a copy constructor?

A member function known as a copy constructor initializes an item using another object from the same class—an in-depth discussion on Copy Constructors.

Every time we specify one or more non-default constructors (with parameters) for a class, we also need to include a default constructor (without parameters), as the compiler won't supply one in this circumstance. The best practice is to always declare a default constructor, even though it is not required.

A reference to an object belonging to the same class is required by the copy constructor.



```
Sample(Sample &t)
```

```
{  
id=t.id;  
}
```

What is a destructor in C++?

An equivalent special member function to a constructor is a destructor. The constructor creates class objects, which are destroyed by the destructor. The word "destructor," followed by the tilde () symbol, is the same as the class name. You can only define one destructor at a time. One method of destroying an object made by a constructor is to use a destructor. Destructors cannot be overloaded as a result. Destructors don't take any arguments and don't give anything back. As soon as the item leaves the scope, it is immediately called. Destructors free up the memory used by the objects the constructor generated. Destructor reverses the process of creating things by destroying them.

The language used to define the class's destructor

```
~<class-name>()  
{  
}
```

The language used to define the class's destructor outside of it

```
<class-name>:: ~<class-name>(){}  
{  
}
```

← Prev

Next →

C++ Destructor

A destructor works opposite to constructor; it destructs the objects of classes. It can be defined only once in a class. Like constructors, it is invoked automatically.

A destructor is defined like constructor. It must have same name as class. But it is prefixed with a tilde sign (~).

Note: C++ destructor cannot have parameters. Moreover, modifiers can't be applied on destructors.

C++ Constructor and Destructor Example

Let's see an example of constructor and destructor in C++ which is called automatically.

```
#include <iostream>
using namespace std;
class Employee
{
public:
    Employee()
    {
        cout<<"Constructor Invoked"<<endl;
    }
    ~Employee()
    {
        cout<<"Destructor Invoked"<<endl;
    }
};
int main(void)
{
    Employee e1; //creating an object of Employee
    Employee e2; //creating an object of Employee
    return 0;
}
```

Output:

```
Constructor Invoked  
Constructor Invoked  
Destructor Invoked  
Destructor Invoked
```

← Prev

Next →

AD



For Videos Join Our Youtube Channel: [Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share



Learn Latest Tutorials



Splunk



SPSS



Swagger tutorial

Swagger



Transact-SQL

C++ this Pointer

In C++ programming, **this** is a keyword that refers to the current instance of the class. There can be 3 main usage of this keyword in C++.

- It can be used **to pass current object as a parameter to another method.**
- It can be used **to refer current class instance variable.**
- It can be used **to declare indexers.**

C++ this Pointer Example

Let's see the example of this keyword in C++ that refers to the fields of current class.

```
#include <iostream>
using namespace std;
class Employee {
public:
    int id; //data member (also instance variable)
    string name; //data member(also instance variable)
    float salary;
    Employee(int id, string name, float salary)
    {
        this->id = id;
        this->name = name;
        this->salary = salary;
    }
    void display()
    {
        cout<<id<< " "<<name<< " "<<salary<<endl;
    }
};

int main(void) {
    Employee e1 =Employee(101, "Sonoo", 890000); //creating an object of Employee
    Employee e2=Employee(102, "Nakul", 59000); //creating an object of Employee
    e1.display();
    e2.display();
}
```

```
return 0;  
}
```

Output:

```
101 Sonoo 890000  
102 Nakul 59000
```

← Prev

Next →

AD

 YouTube For Videos Join Our YouTube Channel: [Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share



Learn Latest Tutorials



C++ static

In C++, static is a keyword or modifier that belongs to the type not instance. So instance is not required to access the static members. In C++, static can be field, method, constructor, class, properties, operator and event.

Advantage of C++ static keyword

Memory efficient: Now we don't need to create instance for accessing the static members, so it saves memory. Moreover, it belongs to the type, so it will not get memory each time when instance is created.

C++ Static Field

A field which is declared as static is called static field. Unlike instance field which gets memory each time whenever you create object, there is only one copy of static field created in the memory. It is shared to all the objects.

It is used to refer the common property of all objects such as rateOfInterest in case of Account, companyName in case of Employee etc.

C++ static field example

Let's see the simple example of static field in C++.

```
#include <iostream>
using namespace std;
class Account {
public:
    int accno; //data member (also instance variable)
    string name; //data member(also instance variable)
    static float rateOfInterest;
    Account(int accno, string name)
    {
        this->accno = accno;
        this->name = name;
    }
}
```

```

void display()
{
    cout<<accno<< "<<name<< " "<<rateOfInterest<<endl;
}
};

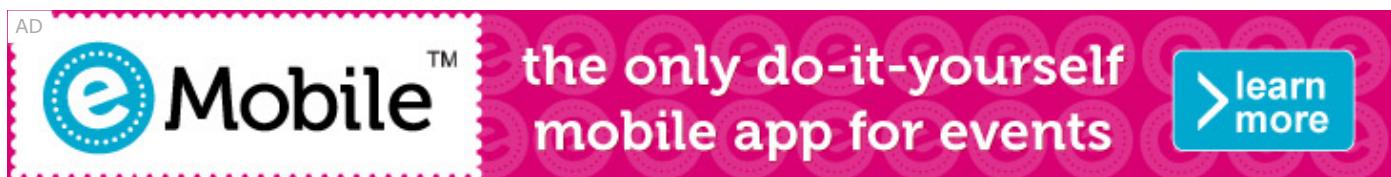
float Account::rateOfInterest=6.5;

int main(void) {
    Account a1 =Account(201, "Sanjay"); //creating an object of Employee
    Account a2=Account(202, "Nakul"); //creating an object of Employee
    a1.display();
    a2.display();
    return 0;
}

```

Output:

201 Sanjay 6.5
202 Nakul 6.5



C++ static field example: Counting Objects

Let's see another example of static keyword in C++ which counts the objects.

```

#include <iostream>
using namespace std;
class Account {
public:
    int accno; //data member (also instance variable)
    string name;
    static int count;
    Account(int accno, string name)
    {
        this->accno = accno;
        this->name = name;
    }
};

```

```
        count++;
    }

void display()
{
    cout<<accno<< " " <<name<<endl;
}

};

int Account::count=0;

int main(void) {
    Account a1 =Account(201, "Sanjay"); //creating an object of Account
    Account a2=Account(202, "Nakul");
    Account a3=Account(203, "Ranjana");
    a1.display();
    a2.display();
    a3.display();
    cout<<"Total Objects are: "<<Account::count;
    return 0;
}
```

Output:

```
201 Sanjay
202 Nakul
203 Ranjana
Total Objects are: 3
```

← Prev

Next →

C++ Structs

In C++, classes and structs are blueprints that are used to create the instance of a class. Structs are used for lightweight objects such as Rectangle, color, Point, etc.

Unlike class, structs in C++ are value type than reference type. It is useful if you have data that is not intended to be modified after creation of struct.

C++ Structure is a collection of different data types. It is similar to the class that holds different types of data.

The Syntax Of Structure

```
struct structure_name
{
    // member declarations.
}
```

In the above declaration, a structure is declared by preceding the **struct keyword** followed by the identifier(structure name). Inside the curly braces, we can declare the member variables of different types. **Consider the following situation:**

```
struct Student
{
    char name[20];
    int id;
    int age;
}
```

In the above case, Student is a structure contains three variables name, id, and age. When the structure is declared, no memory is allocated. When the variable of a structure is created, then the memory is allocated. Let's understand this scenario.

How to create the instance of Structure?

Structure variable can be defined as:

```
Student s;
```

Here, s is a structure variable of type **Student**. When the structure variable is created, the memory will be allocated. Student structure contains one char variable and two integer variable. Therefore, the memory for one char variable is 1 byte and two ints will be $2 \times 4 = 8$. The total memory occupied by the s variable is 9 byte.

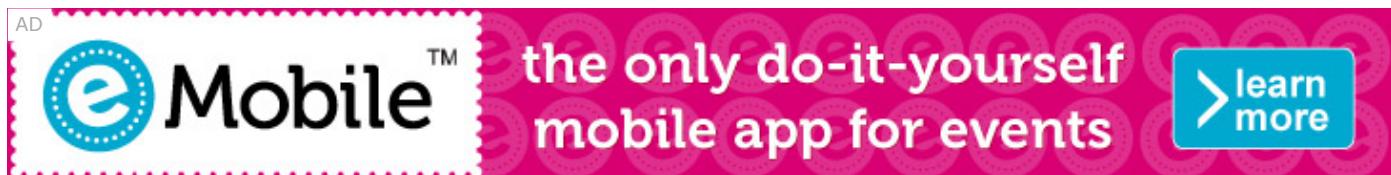
How to access the variable of Structure:

The variable of the structure can be accessed by simply using the instance of the structure followed by the dot (.) operator and then the field of the structure.

For example:

```
s.id = 4;
```

In the above statement, we are accessing the id field of the structure Student by using the **dot(.)** operator and assigns the value 4 to the id field.



C++ Struct Example

Let's see a simple example of struct Rectangle which has two data members width and height.

```
#include <iostream>
using namespace std;
struct Rectangle
{
    int width, height;

};

int main(void) {
    struct Rectangle rec;
    rec.width=8;
    rec.height=5;
    cout<<"Area of Rectangle is: "<<(rec.width * rec.height)<<endl;
    return 0;
}
```

{

Output:

Area of Rectangle is: 40

C++ Struct Example: Using Constructor and Method

Let's see another example of struct where we are using the constructor to initialize data and method to calculate the area of rectangle.

```
#include <iostream>
using namespace std;
struct Rectangle {
    int width, height;
    Rectangle(int w, int h)
    {
        width = w;
        height = h;
    }
    void areaOfRectangle() {
        cout<<"Area of Rectangle is: "<<(width*height); }
};

int main(void) {
    struct Rectangle rec=Rectangle(4,6);
    rec.areaOfRectangle();
    return 0;
}
```

Output:

Area of Rectangle is: 24

Structure v/s Class

Structure**Class**

If access specifier is not declared explicitly, then by default access specifier will be public.	If access specifier is not declared explicitly, then by default access specifier will be private.
Syntax of Structure: struct structure_name { // body of the structure. }	Syntax of Class: class class_name { // body of the class. }
The instance of the structure is known as "Structure variable".	The instance of the class is known as "Object of the class".

[← Prev](#)[Next →](#)

AD

 For Videos Join Our YouTube Channel: [Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share



C++ Enumeration

Enum in C++ is a data type that contains fixed set of constants.

It can be used for days of the week (SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY and SATURDAY), directions (NORTH, SOUTH, EAST and WEST) etc. The C++ enum constants are static and final implicitly.

C++ Enums can be thought of as classes that have fixed set of constants.

Points to remember for C++ Enum

- enum improves type safety
- enum can be easily used in switch
- enum can be traversed
- enum can have fields, constructors and methods
- enum may implement many interfaces but cannot extend any class because it internally extends `Enum` class

C++ Enumeration Example

Let's see the simple example of enum data type used in C++ program.

```
#include <iostream>
using namespace std;
enum week { Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday };
int main()
{
    week day;
    day = Friday;
    cout << "Day: " << day+1 << endl;
    return 0;
}
```

Output:

Day: 5

← Prev

Next →

AD

 [Youtube For Videos Join Our Youtube Channel: Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share



Learn Latest Tutorials

 [Splunk tutorial](#)

Splunk

 [SPSS tutorial](#)

SPSS

 [Swagger tutorial](#)

Swagger

 [T-SQL tutorial](#)

Transact-SQL

 [Tumblr tutorial](#)

Tumblr

 [React tutorial](#)

ReactJS

 [Regex tutorial](#)

Regex

 [Reinforcement learning tutorial](#)

learning

C++ Friend function

If a function is defined as a friend function in C++, then the protected and private data of a class can be accessed using the function.

By using the keyword `friend` compiler knows the given function is a friend function.

For accessing the data, the declaration of a friend function should be done inside the body of a class starting with the keyword `friend`.

Declaration of friend function in C++

```
class class_name
{
    friend data_type function_name(argument/s);      // syntax of friend function.
};
```

In the above declaration, the friend function is preceded by the keyword `friend`. The function can be defined anywhere in the program like a normal C++ function. The function definition does not use either the keyword **friend or scope resolution operator**.

Characteristics of a Friend function:

- The function is not in the scope of the class to which it has been declared as a friend.
- It cannot be called using the object as it is not in the scope of that class.
- It can be invoked like a normal function without using the object.
- It cannot access the member names directly and has to use an object name and dot membership operator with the member name.
- It can be declared either in the private or the public part.

C++ friend function Example

Let's see the simple example of C++ friend function used to print the length of a box.

```
#include <iostream>
using namespace std;
class Box
{
```

private:**int** length;**public:**

Box(): length(0) { }

friend int printLength(Box); //friend function

};

int printLength(Box b)

{

b.length += 10;

return b.length;

}

int main()

{

Box b;

cout << "Length of box: " << printLength(b) << endl;

return 0;

}

Output:

Length of box: 10

Let's see a simple example when the function is friendly to two classes.

```
#include <iostream>
using namespace std;
class B;      // forward declarartion.
class A
{
    int x;
public:
    void setdata(int i)
    {
        x=i;
    }
    friend void min(A,B);      // friend function.
};
```

```
class B
{
    int y;
public:
    void setdata(int i)
    {
        y=i;
    }
    friend void min(A,B);           // friend function
};

void min(A a,B b)
{
    if(a.x<=b.y)
        std::cout << a.x << std::endl;
    else
        std::cout << b.y << std::endl;
}

int main()
{
    A a;
    B b;
    a.setdata(10);
    b.setdata(20);
    min(a,b);
    return 0;
}
```

Output:

```
10
```

In the above example, min() function is friendly to two classes, i.e., the min() function can access the private members of both the classes A and B.

C++ Friend class

A friend class can access both private and protected members of the class in which it has been declared as friend.

Let's see a simple example of a friend class.

```
#include <iostream>

using namespace std;

class A
{
    int x =5;
    friend class B;      // friend class.

};

class B
{
public:
    void display(A &a)
    {
        cout<<"value of x is :"<<a.x;
    }
};

int main()
{
    A a;
    B b;
    b.display(a);
    return 0;
}
```

Output:

```
value of x is : 5
```

In the above example, class B is declared as a friend inside the class A. Therefore, B is a friend of class A. Class B can access the private members of class A.

C++ Math Functions

C++ offers some basic math functions and the required header file to use these functions is `<math.h>`

Trigonometric functions

Method	Description
<code>cos(x)</code>	It computes the cosine of x.
<code>sin(x)</code>	It computes the sine of x.
<code>tan(x)</code>	It computes the tangent of x.
<code>acos(x)</code>	It finds the inverse cosine of x.
<code>asin(x)</code>	It finds the inverse sine of x.
<code>atan(x)</code>	It finds the inverse tangent of x.
<code>atan2(x,y)</code>	It finds the inverse tangent of a coordinate x and y.

Hyperbolic functions

Method	Description
<code>cosh(x)</code>	It computes the hyperbolic cosine of x.
<code>sinh(x)</code>	It computes the hyperbolic sine of x.
<code>tanh(x)</code>	It computes the hyperbolic tangent of x.
<code>acosh(x)</code>	It finds the arc hyperbolic cosine of x.
<code>asinh(x)</code>	It finds the arc hyperbolic sine of x.
<code>atanh(x)</code>	It finds the arc hyperbolic tangent of x.

Exponential functions

Method	Description
<code>exp(x)</code>	It computes the exponential e raised to the power x.
<code>frexp(value_type x,int* exp)</code>	It breaks a number into significand and 2 raised to the power exponent.
<code>ldexp(float x, int e)</code>	It computes the product of x and 2 raised to the power e.
<code>log(x)</code>	It computes the natural logarithm of x.
<code>log10(x)</code>	It computes the common logarithm of x.
<code>modf()</code>	It breaks a number into an integer and fractional part.
<code>exp2(x)</code>	It computes the base 2 exponential of x.
<code>expm1(x)</code>	It computes the exponential raised to the power x minus one.
<code>log1p(x)</code>	It computes the natural logarithm of x plus one.
<code>log2(x)</code>	It computes the base 2 logarithm of x.
<code>logb(x)</code>	It computes the logarithm of x.
<code>scalbn(x, n)</code>	It computes the product of x and FLT_RADX raised to the power n.
<code>scalbln(x, n)</code>	It computes the product of x and FLT_RADX raised to the power n.
<code>ilogb(x)</code>	It returns the exponent part of x.

AD

Floating point manipulation functions

Method	Description
<code>copysign(x,y)</code>	It returns the magnitude of x with the sign of y.
<code>nextafter(x,y)</code>	It represents the next representable value of x in the direction of y.

<code>nexttoward(x,y)</code>	It represents the next representable value of x in the direction of y.
------------------------------	--

Maximum, Minimum and Difference functions

Method	Description
<code>fdim(x,y)</code>	It calculates the positive difference between x and y.
<code>fmax(x,y)</code>	It returns the larger number among two numbers x and y.
<code>fmin()</code>	It returns the smaller number among two numbers x and y .

Power functions

Method	Description
<code>pow(x,y)</code>	It computes x raised to the power y.
<code>sqrt(x)</code>	It computes the square root of x.
<code>cbrt(x)</code>	It computes the cube root of x.
<code>hypot(x,y)</code>	It finds the hypotenuse of a right angled triangle.

Nearest integer operations

Method	Description
<code>ceil(x)</code>	It rounds up the value of x.
<code>floor(x)</code>	It rounds down the value of x.
<code>round(x)</code>	It rounds off the value of x.
<code>lround(x)</code>	It rounds off the value of x and cast to long integer.
<code>llround(x)</code>	It rounds off the value of x and cast to long long integer.
<code>fmod(n,d)</code>	It computes the remainder of division n/d.
<code>trunc(x)</code>	It rounds off the value x towards zero.
<code>rint(x)</code>	It rounds off the value of x using rounding mode.

<code>lrint(x)</code>	It rounds off the value of x using rounding mode and cast to long integer.
<code>llrint(x)</code>	It rounds off the value x and cast to long long integer.
<code>nearbyint(x)</code>	It rounds off the value x to a nearby integral value.
<code>remainder(n,d)</code>	It computes the remainder of n/d.
<code>remquo()</code>	It computes remainder and quotient both.

Other functions

Method	Description
<code>fabs(x)</code>	It computes the absolute value of x.
<code>abs(x)</code>	It computes the absolute value of x.
<code>fma(x,y,z)</code>	It computes the expression $x*y+z$.

Macro functions

Method	Description
<code>fpclassify(x)</code>	It returns the value of type that matches one of the macro constants.
<code>isfinite(x)</code>	It checks whether x is finite or not.
<code>isinf()</code>	It checks whether x is infinite or not.
<code>isnan()</code>	It checks whether x is nan or not.
<code>isnormal(x)</code>	It checks whether x is normal or not.
<code>signbit(x)</code>	It checks whether the sign of x is negative or not.

Comparison macro functions

Method	Description
<code>isgreater(x,y)</code>	It determines whether x is greater than y or not.
<code>isgreaterequal(x,y)</code>	It determines whether x is greater than or equal to y or not.

<code>less(x,y)</code>	It determines whether x is less than y or not.
<code>islessequal(x,y)</code>	It determines whether x is less than or equal to y.
<code>islessgreater(x,y)</code>	It determines whether x is less or greater than y or not.
<code>isunordered(x,y)</code>	It checks whether x can be meaningfully compared or not.

Error and gamma functions

Method	Description
<code>erf(x)</code>	It computes the error function value of x.
<code>erfc(x)</code>	It computes the complementary error function value of x.
<code>tgamma(x)</code>	It computes the gamma function value of x.
<code>lgamma(x)</code>	It computes the logarithm of a gamma function of x.

← Prev

Next →

AD

 [For Videos Join Our Youtube Channel: Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

C++ Inheritance

In C++, inheritance is a process in which one object acquires all the properties and behaviors of its parent object automatically. In such way, you can reuse, extend or modify the attributes and behaviors which are defined in other class.

In C++, the class which inherits the members of another class is called derived class and the class whose members are inherited is called base class. The derived class is the specialized class for the base class.

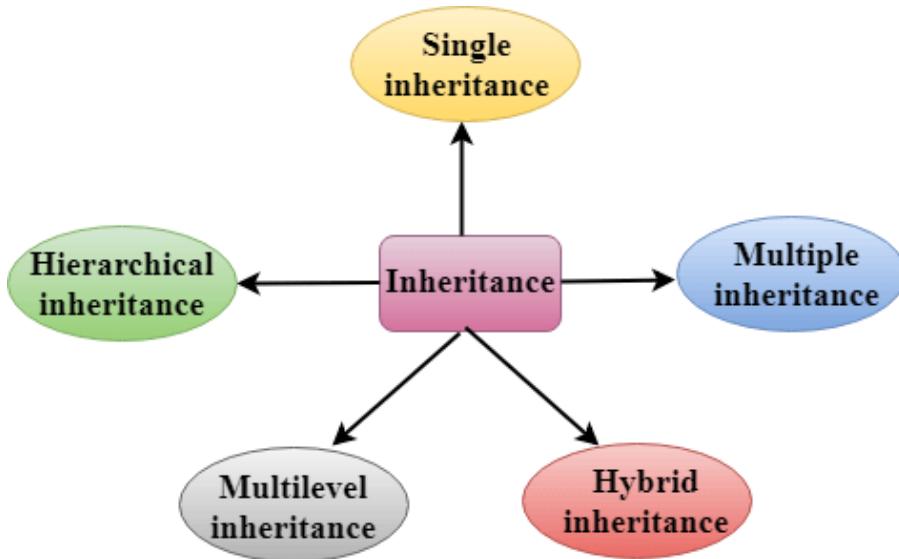
Advantage of C++ Inheritance

Code reusability: Now you can reuse the members of your parent class. So, there is no need to define the member again. So less code is required in the class.

Types Of Inheritance

C++ supports five types of inheritance:

- Single inheritance
- Multiple inheritance
- Hierarchical inheritance
- Multilevel inheritance
- Hybrid inheritance



Derived Classes

A Derived class is defined as the class derived from the base class.

The Syntax of Derived class:

```
class derived_class_name :: visibility-mode base_class_name
{
    // body of the derived class.
}
```

Where,

derived_class_name: It is the name of the derived class.

visibility mode: The visibility mode specifies whether the features of the base class are publicly inherited or privately inherited. It can be public or private.

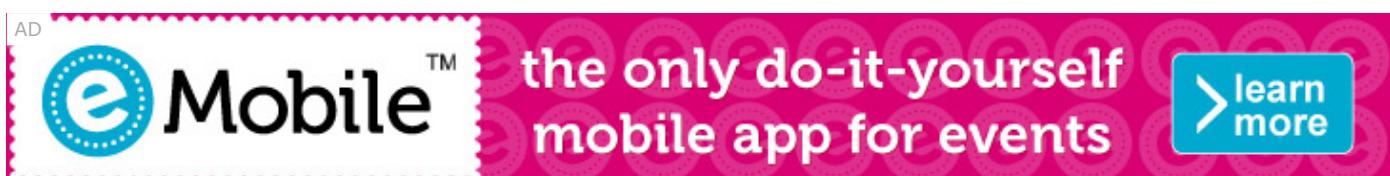
base_class_name: It is the name of the base class.

- When the base class is privately inherited by the derived class, public members of the base class becomes the private members of the derived class. Therefore, the public members of the base class are not accessible by the objects of the derived class only by the member functions of the derived class.
- 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 base class.



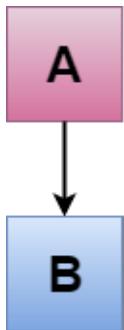
Note:

- In C++, the default mode of visibility is private.
- The private members of the base class are never inherited.



C++ Single Inheritance

Single inheritance is defined as the inheritance in which a derived class is inherited from the only one base class.



Where 'A' is the base class, and 'B' is the derived class.

C++ Single Level Inheritance Example: Inheriting Fields

When one class inherits another class, it is known as single level inheritance. Let's see the example of single level inheritance which inherits the fields only.

```
#include <iostream>
using namespace std;
class Account {
public:
    float salary = 60000;
};

class Programmer: public Account {
public:
    float bonus = 5000;
};

int main(void) {
    Programmer p1;
    cout<<"Salary: "<<p1.salary<<endl;
    cout<<"Bonus: "<<p1.bonus<<endl;
    return 0;
}
```

Output:

```
Salary: 60000
```

```
Bonus: 5000
```

In the above example, Employee is the **base** class and Programmer is the **derived** class.

C++ Single Level Inheritance Example: Inheriting Methods

Let's see another example of inheritance in C++ which inherits methods only.

```
#include <iostream>
using namespace std;
class Animal {
public:
void eat() {
    cout<<"Eating..."<<endl;
}
;
class Dog: public Animal
{
public:
void bark(){
cout<<"Barking...";
}
;
int main(void) {
    Dog d1;
    d1.eat();
    d1.bark();
    return 0;
}
```

Output:



```
Eating...  
Barking...
```

Let's see a simple example.

```
#include <iostream>  
using namespace std;  
class A  
{  
    int a = 4;  
    int b = 5;  
    public:  
    int mul()  
    {  
        int c = a*b;  
        return c;  
    }  
};  
  
class B : private A  
{  
    public:  
    void display()  
    {  
        int result = mul();  
        std::cout << "Multiplication of a and b is : " << result << std::endl;  
    }  
};  
int main()  
{  
    B b;  
    b.display();  
  
    return 0;  
}
```

Output:

```
Multiplication of a and b is : 20
```

In the above example, class A is privately inherited. Therefore, the mul() function of class 'A' cannot be accessed by the object of class B. It can only be accessed by the member function of class B.

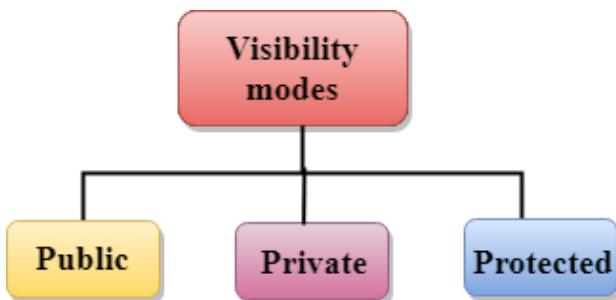
How to make a Private Member Inheritable

The private member is not inheritable. If we modify the visibility mode by making it public, but this takes away the advantage of data hiding.

C++ introduces a third visibility modifier, i.e., **protected**. The member which is declared as protected will be accessible to all the member functions within the class as well as the class immediately derived from it.



Visibility modes can be classified into three categories:



- **Public:** When the member is declared as public, it is accessible to all the functions of the program.
- **Private:** When the member is declared as private, it is accessible within the class only.
- **Protected:** When the member is declared as protected, it is accessible within its own class as well as the class immediately derived from it.

Visibility of Inherited Members

Base class visibility	Derived class visibility	
Public	Private	Protected

Private	Not Inherited	Not Inherited	Not Inherited
Protected	Protected	Private	Protected
Public	Public	Private	Protected

C++ Multilevel Inheritance

Multilevel inheritance is a process of deriving a class from another derived class.



C++ Multi Level Inheritance Example

When one class inherits another class which is further inherited by another class, it is known as multi level inheritance in C++. Inheritance is transitive so the last derived class acquires all the members of all its base classes.

Let's see the example of multi level inheritance in C++.

```

#include <iostream>
using namespace std;
class Animal {
public:
void eat() {
    cout<<"Eating..."<<endl;
}
class Dog: public Animal
{
  
```

```

public:
void bark(){
    cout<<"Barking..."<<endl;
}
};

class BabyDog: public Dog
{
    public:
    void weep() {
        cout<<"Weeping...";
    }
};

int main(void) {
    BabyDog d1;
    d1.eat();
    d1.bark();
    d1.weep();
    return 0;
}

```

Output:

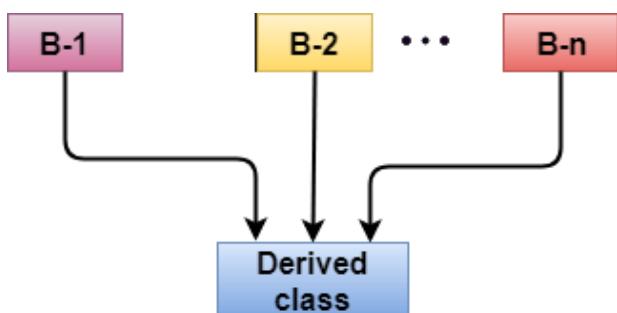
```

Eating...
Barking...
Weeping...

```

C++ Multiple Inheritance

Multiple inheritance is the process of deriving a new class that inherits the attributes from two or more classes.



Syntax of the Derived class:

```
class D : visibility B-1, visibility B-2, ?  
{  
    // Body of the class;  
}
```

Let's see a simple example of multiple inheritance.

```
#include <iostream>  
using namespace std;  
class A  
{  
    protected:  
        int a;  
    public:  
        void get_a(int n)  
        {  
            a = n;  
        }  
};  
  
class B  
{  
    protected:  
        int b;  
    public:  
        void get_b(int n)  
        {  
            b = n;  
        }  
};  
  
class C : public A,public B  
{  
    public:  
        void display()  
        {
```

```
    std::cout << "The value of a is : " <<a<< std::endl;
    std::cout << "The value of b is : " <<b<< std::endl;
    cout<<"Addition of a and b is :"<<a+b;
}

};

int main()
{
    C c;
    c.get_a(10);
    c.get_b(20);
    c.display();

    return 0;
}
```

Output:

```
The value of a is : 10
The value of b is : 20
Addition of a and b is : 30
```

In the above example, class 'C' inherits two base classes 'A' and 'B' in a public mode.

Ambiguity Resolution in Inheritance

Ambiguity can be occurred in using the multiple inheritance when a function with the same name occurs in more than one base class.

Let's understand this through an example:

```
#include <iostream>
using namespace std;
class A
{
    public:
    void display()
    {
        std::cout << "Class A" << std::endl;
```

```
}

};

class B

{

public:

void display()

{

    std::cout << "Class B" << std::endl;

}

};

class C : public A, public B

{

void view()

{

    display();

}

};

int main()

{

    C c;

    c.display();

    return 0;

}
```

Output:

```
error: reference to 'display' is ambiguous
        display();
```

- The above issue can be resolved by using the class resolution operator with the function. In the above example, the derived class code can be rewritten as:

```
class C : public A, public B

{

void view()

{

    A :: display();      // Calling the display() function of class A.
```

```
B :: display();      // Calling the display() function of class B.  
}  
};
```

An ambiguity can also occur in single inheritance.

Consider the following situation:

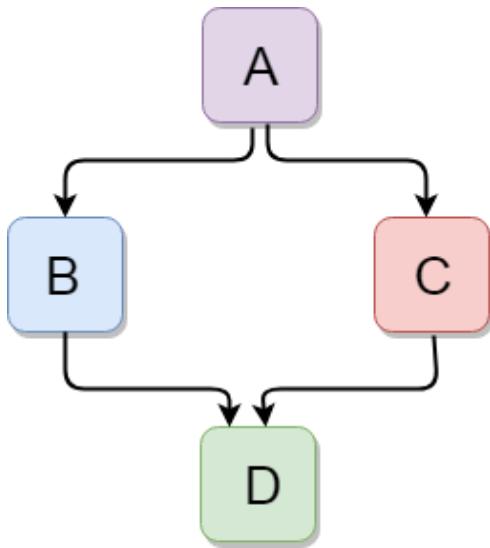
```
class A  
{  
public:  
void display()  
{  
cout<<"Class A";  
}  
};  
  
class B  
{  
public:  
void display()  
{  
cout<<"Class B";  
}  
};
```

In the above case, the function of the derived class overrides the method of the base class. Therefore, call to the display() function will simply call the function defined in the derived class. If we want to invoke the base class function, we can use the class resolution operator.

```
int main()  
{  
B b;  
b.display();      // Calling the display() function of B class.  
b.B :: display(); // Calling the display() function defined in B class.  
}
```

C++ Hybrid Inheritance

Hybrid inheritance is a combination of more than one type of inheritance.



Let's see a simple example:

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

class A
{
protected:
    int a;
public:
    void get_a()
    {
        std::cout << "Enter the value of 'a' : " << std::endl;
        cin>>a;
    }
};

class B : public A
{
protected:
    int b;
public:
    void get_b()
    {
```

```
    std::cout << "Enter the value of 'b' : " << std::endl;
    cin>>b;
}

};

class C
{
    protected:
    int c;
    public:
    void get_c()
    {
        std::cout << "Enter the value of c is : " << std::endl;
        cin>>c;
    }
};

class D : public B, public C
{
    protected:
    int d;
    public:
    void mul()
    {
        get_a();
        get_b();
        get_c();
        std::cout << "Multiplication of a,b,c is : " << a*b*c << std::endl;
    }
};

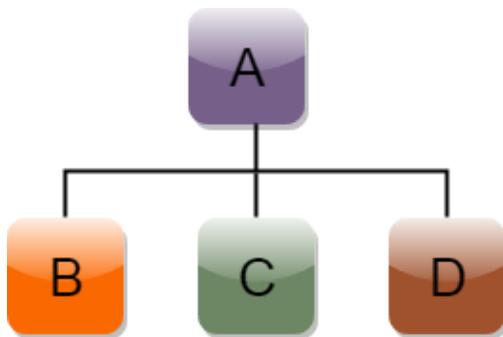
int main()
{
    D d;
    d.mul();
    return 0;
}
```

Output:

```
Enter the value of 'a' :  
10  
Enter the value of 'b' :  
20  
Enter the value of c is :  
30  
Multiplication of a,b,c is : 6000
```

C++ Hierarchical Inheritance

Hierarchical inheritance is defined as the process of deriving more than one class from a base class.



Syntax of Hierarchical inheritance:

```
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.  
}
```

Let's see a simple example:

```
#include <iostream>
using namespace std;
class Shape           // Declaration of base class.
{
public:
    int a;
    int b;
    void get_data(int n,int m)
    {
        a= n;
        b = m;
    }
};

class Rectangle : public Shape // inheriting Shape class
{
public:
    int rect_area()
    {
        int result = a*b;
        return result;
    }
};

class Triangle : public Shape // inheriting Shape class
{
public:
    int triangle_area()
    {
        float result = 0.5*a*b;
        return result;
    }
};

int main()
{
    Rectangle r;
    Triangle t;
```

```
int length,breadth,base,height;  
std::cout << "Enter the length and breadth of a rectangle: " << std::endl;  
cin>>length>>breadth;  
r.get_data(length,breadth);  
int m = r.rect_area();  
std::cout << "Area of the rectangle is : " <<m<< std::endl;  
std::cout << "Enter the base and height of the triangle: " << std::endl;  
cin>>base>>height;  
t.get_data(base,height);  
float n = t.triangle_area();  
std::cout <<"Area of the triangle is : " << n<<std::endl;  
return 0;  
}
```

Output:

```
Enter the length and breadth of a rectangle:  
23  
20  
Area of the rectangle is : 460  
Enter the base and height of the triangle:  
2  
5  
Area of the triangle is : 5
```

← Prev

Next →

C++ Aggregation (HAS-A Relationship)

In C++, aggregation is a process in which one class defines another class as any entity reference. It is another way to reuse the class. It is a form of association that represents HAS-A relationship.

C++ Aggregation Example

Let's see an example of aggregation where Employee class has the reference of Address class as data member. In such way, it can reuse the members of Address class.

```
#include <iostream>
using namespace std;
class Address {
public:
    string addressLine, city, state;
    Address(string addressLine, string city, string state)
    {
        this->addressLine = addressLine;
        this->city = city;
        this->state = state;
    }
};

class Employee
{
private:
    Address* address; //Employee HAS-A Address
public:
    int id;
    string name;
    Employee(int id, string name, Address* address)
    {
        this->id = id;
        this->name = name;
        this->address = address;
    }
    void display()
}
```

```
{  
    cout<<id << " " <<name<< " " <<  
    address->addressLine<< " " << address->city<< " " <<address->state<<endl;  
}  
};  
int main(void) {  
    Address a1= Address("C-146, Sec-15","Noida","UP");  
    Employee e1 = Employee(101,"Nakul",&a1);  
    e1.display();  
    return 0;  
}
```

Output:

```
101 Nakul C-146, Sec-15 Noida UP
```

← Prev

Next →

AD

 [For Videos Join Our YouTube Channel: Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

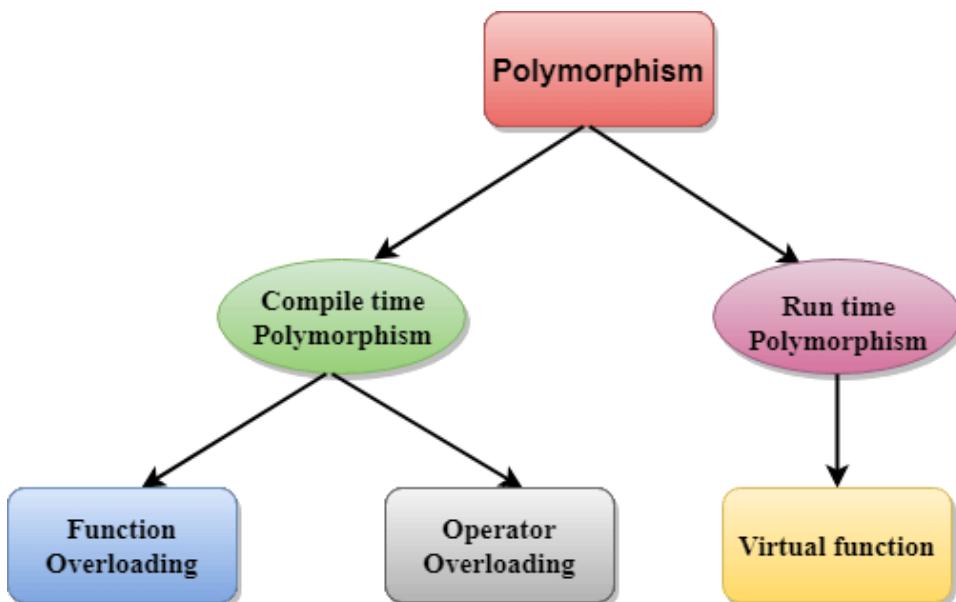
C++ Polymorphism

The term "Polymorphism" is the combination of "poly" + "morphs" which means many forms. It is a greek word. In object-oriented programming, we use 3 main concepts: inheritance, encapsulation, and polymorphism.

Real Life Example Of Polymorphism

Let's consider a real-life example of polymorphism. A lady behaves like a teacher in a classroom, mother or daughter in a home and customer in a market. Here, a single person is behaving differently according to the situations.

There are two types of polymorphism in C++:



- **Compile time polymorphism:** The overloaded functions are invoked by matching the type and number of arguments. This information is available at the compile time and, therefore, compiler selects the appropriate function at the compile time. It is achieved by function overloading and operator overloading which is also known as static binding or early binding. Now, let's consider the case where function name and prototype is same.

```

class A                                // base class declaration.

{
    int a;
    public:
        void display()
    {
        cout<< "Class A ";
    }
}
  
```

```

    }
};

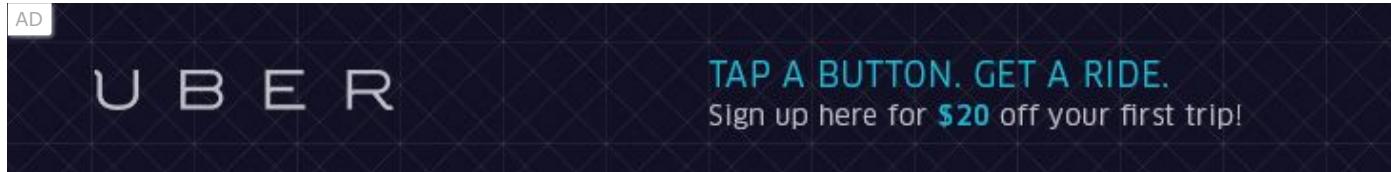
class B : public A          // derived class declaration.

{
    int b;
    public:
        void display()
    {
        cout<<"Class B";
    }
};

```

In the above case, the prototype of display() function is the same in both the **base and derived class**. Therefore, the static binding cannot be applied. It would be great if the appropriate function is selected at the run time. This is known as **run time polymorphism**.

- **Run time polymorphism:** Run time polymorphism is achieved when the object's method is invoked at the run time instead of compile time. It is achieved by method overriding which is also known as dynamic binding or late binding.



Differences b/w compile time and run time polymorphism.

Compile time polymorphism	Run time polymorphism
The function to be invoked is known at the compile time.	The function to be invoked is known at the run time.
It is also known as overloading, early binding and static binding.	It is also known as overriding, Dynamic binding and late binding.
Overloading is a compile time polymorphism where more than one method is having the same name but with the different number of parameters or the type of the parameters.	Overriding is a run time polymorphism where more than one method is having the same name, number of parameters and the type of the parameters.

It is achieved by function overloading and operator overloading.	It is achieved by virtual functions and pointers.
It provides fast execution as it is known at the compile time.	It provides slow execution as it is known at the run time.
It is less flexible as mainly all the things execute at the compile time.	It is more flexible as all the things execute at the run time.

C++ Runtime Polymorphism Example

Let's see a simple example of run time polymorphism in C++.

// an example without the virtual keyword.

```
#include <iostream>
using namespace std;
class Animal {
public:
void eat(){
cout<<"Eating...";
}
};

class Dog: public Animal
{
public:
void eat()
{
    cout<<"Eating bread...";
}
};

int main(void) {
Dog d = Dog();
d.eat();
return 0;
}
```

Output:

Eating bread...

AD



VIDEO STREAMING FOR THE AWAKENED MIND

GET STARTED NOW

C++ Run time Polymorphism Example: By using two derived class

Let's see another example of run time polymorphism in C++ where we are having two derived classes.

// an example with virtual keyword.

```
#include <iostream>
using namespace std;
class Shape { // base class
public:
virtual void draw(){ // virtual function
cout<<"drawing..."<<endl;
}
};

class Rectangle: public Shape // inheriting Shape class.
{
public:
void draw()
{
cout<<"drawing rectangle..."<<endl;
}
};

class Circle: public Shape // inheriting Shape class.

{
public:
void draw()
{
cout<<"drawing circle..."<<endl;
}
};
```

```

};

int main(void) {
    Shape *s;                                // base class pointer.
    Shape sh;                                 // base class object.

    Rectangle rec;
    Circle cir;

    s=&sh;
    s->draw();

    s=&rec;
    s->draw();

    s=?
    s->draw();
}

```

Output:

```

drawing...
drawing rectangle...
drawing circle...

```

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.

```

#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";
};

int main(void) {

```

```
Animal d= Dog();
cout<<d.color;
}
```

Output:

Black

← Prev

Next →

AD

 YouTube For Videos Join Our YouTube Channel: [Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share



Learn Latest Tutorials

 [Splunk tutorial](#)

 [SPSS tutorial](#)

C++ Overloading (Function and Operator)

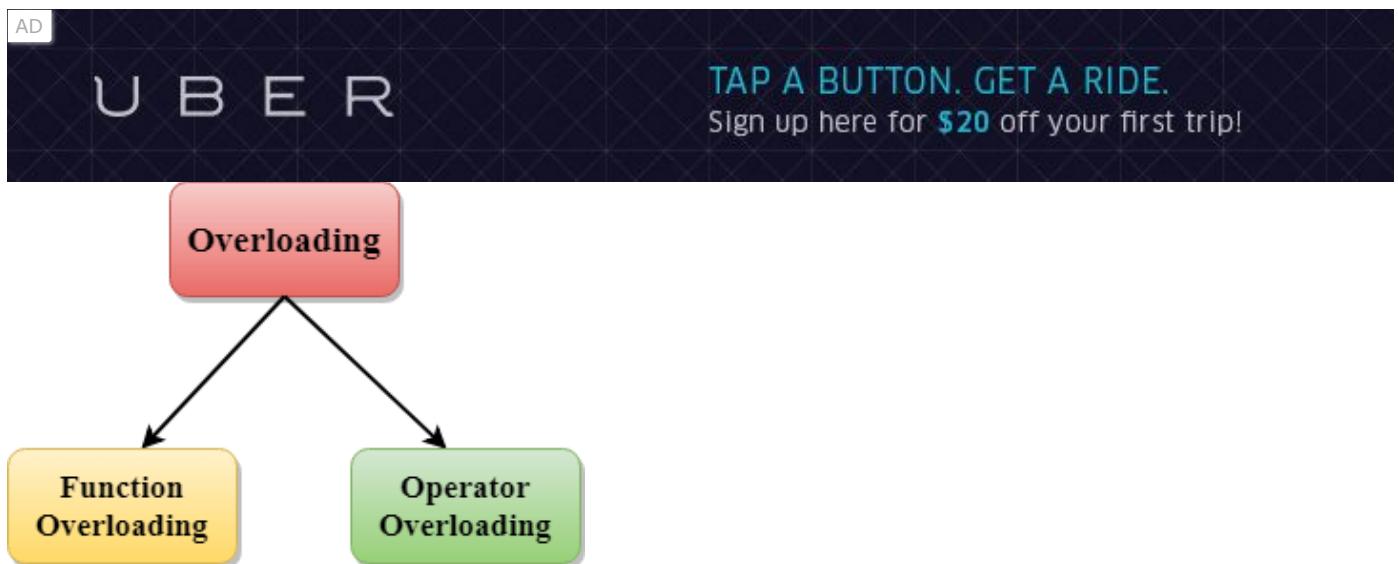
If we create two or more members having the same name but different in number or type of parameter, it is known as C++ overloading. In C++, we can overload:

- 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.

C++ Function Overloading Example

Let's see the simple example of function overloading where we are changing number of arguments of add() method.

// 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;
}
```

Output:

```
30
55
```

Let's see the simple example when the type of the arguments vary.

// 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);
    std::cout << "r1 is :" << r1 << std::endl;
    std::cout << "r2 is :" << r2 << std::endl;
    return 0;
}
```

Output:

```
r1 is : 42
r2 is : 0.6
```



Function Overloading and Ambiguity

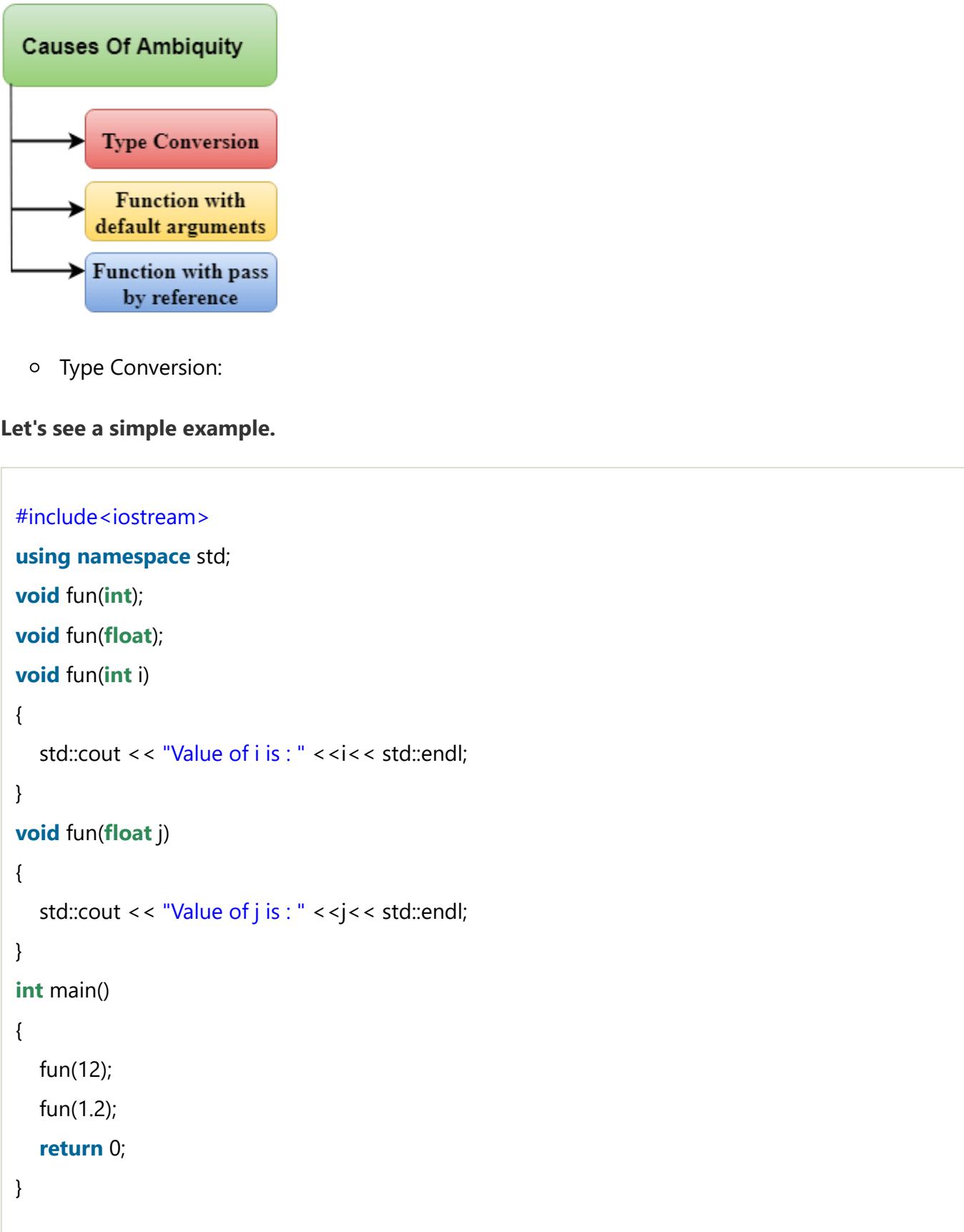
When the compiler is unable to decide which function is to be invoked among the overloaded functions, this situation is known as **function overloading**.

When the compiler shows the ambiguity error, the compiler does not run the program.

Causes of Function Overloading:

- Type Conversion.
- Function with default arguments.

- o Function with pass by reference.



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.

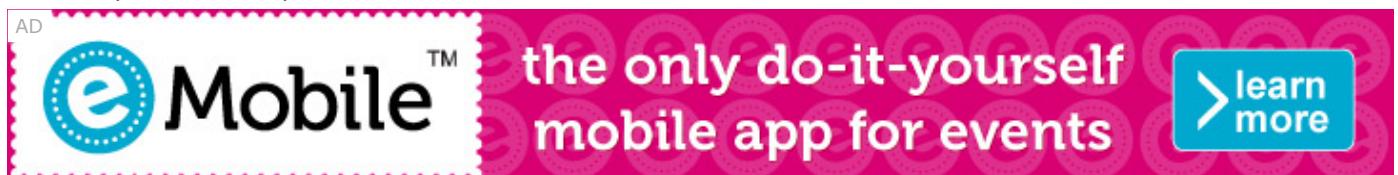
- o Function with Default Arguments

Let's see a simple example.

```
#include<iostream>
using namespace std;
void fun(int);
void fun(int,int);
void fun(int i)
{
    std::cout << "Value of i is : " <<i<< std::endl;
}
void fun(int a,int b=9)
{
    std::cout << "Value of a is : " <<a<< std::endl;
    std::cout << "Value of b is : " <<b<< std::endl;
}
int main()
{
    fun(12);

    return 0;
}
```

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).



- o Function with pass by reference

Let's see a simple example.

```
#include <iostream>
using namespace std;
void fun(int);
void fun(int &);

int main()
{
    int a=10;
    fun(a); // error, which f()?
    return 0;
}

void fun(int x)
{
    std::cout << "Value of x is : " <<x<< std::endl;
}

void fun(int &b)
{
    std::cout << "Value of b is : " <<b<< std::endl;
}
```

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 &).

C++ Operators Overloading

Operator overloading is a compile-time polymorphism in which the operator is overloaded to provide the special meaning to the user-defined data type. Operator overloading is used to overload or redefines most of the operators available in C++. It is used to perform the operation on the user-defined data type. For example, C++ provides the ability to add the variables of the user-defined data type that is applied to the built-in data types.

The advantage of Operators overloading is to perform different operations on the same operand.

Operator that cannot be overloaded are as follows:



- Scope operator (::)
- Sizeof
- member selector(.)
- member pointer selector(*)
- ternary operator(?:)

Syntax of Operator Overloading

```
return_type class_name :: operator op(argument_list)
{
    // body of the function.
}
```

Where the **return type** is the type of value returned by the function.

class_name is the name of the class.

operator op is an operator function where op is the operator being overloaded, and the operator is the keyword.

Rules for Operator Overloading

- Existing operators can only be overloaded, but the new operators cannot be overloaded.
- The overloaded operator contains atleast one operand of the user-defined data type.
- We cannot use friend function to overload certain operators. However, the member function can be used to overload those operators.
- When unary operators are overloaded through a member function take no explicit arguments, but, if they are overloaded by a friend function, takes one argument.
- When binary operators are overloaded through a member function takes one explicit argument, and if they are overloaded through a friend function takes two explicit arguments.

C++ Operators Overloading Example

Let's see the simple example of operator overloading in C++. In this example, void operator ++ () operator function is defined (inside Test class).

```
// program to overload the unary operator ++.
```

```
#include <iostream>
```

```
using namespace std;
class Test
{
private:
    int num;
public:
    Test(): num(8){}
    void operator ++() {
        num = num+2;
    }
    void Print() {
        cout<<"The Count is: "<<num;
    }
};

int main()
{
    Test tt;
    ++tt; // calling of a function "void operator ++()"
    tt.Print();
    return 0;
}
```

Output:

```
The Count is: 10
```

Let's see a simple example of overloading the binary operators.

```
// program to overload the binary operators.
```

```
#include <iostream>
using namespace std;
class A
{
    int x;
public:
```

```
A(){}
A(int i)
{
    x=i;
}
void operator+(A);
void display();
};

void A :: operator+(A a)
{
    int m = x+a.x;
    cout<<"The result of the addition of two objects is : "<<m;

}

int main()
{
    A a1(5);
    A a2(4);
    a1+a2;
    return 0;
}
```

Output:

```
The result of the addition of two objects is : 9
```

← Prev

Next →

C++ Function Overriding

If derived class defines same function as defined in its base class, it is known as function overriding in C++. It is used to achieve runtime polymorphism. It enables you to provide specific implementation of the function which is already provided by its base class.

C++ Function Overriding Example

Let's see a simple example of Function overriding in C++. In this example, we are overriding the eat() function.

```
#include <iostream>
using namespace std;
class Animal {
public:
void eat(){
cout<<"Eating...";
}
};

class Dog: public Animal
{
public:
void eat()
{
cout<<"Eating bread... ";
}
};

int main(void) {
Dog d = Dog();
d.eat();
return 0;
}
```

Output:

```
Eating bread...
```

C++ virtual function

- A C++ virtual function is a member function in the base class that you redefine in a derived class. It is declared using the `virtual` keyword.
- It is used to tell the compiler to perform dynamic linkage or late binding on the function.
- There is a necessity to use the single pointer to refer to all the objects of the different classes. So, we create the pointer to the base class that refers to all the derived objects. But, when base class pointer contains the address of the derived class object, always executes the base class function. This issue can only be resolved by using the '`virtual`' function.
- A '`virtual`' is a keyword preceding the normal declaration of a function.
- When the function is made `virtual`, C++ determines which function is to be invoked at the runtime based on the type of the object pointed by the base class pointer.

Late binding or Dynamic linkage

In late binding function call is resolved during runtime. Therefore compiler determines the type of object at runtime, and then binds the function call.

Rules of Virtual Function

- Virtual functions must be members of some class.
- Virtual functions cannot be static members.
- They are accessed through object pointers.
- They can be a friend of another class.
- A virtual function must be defined in the base class, even though it is not used.
- The prototypes of a virtual function of the base class and all the derived classes must be identical. If the two functions with the same name but different prototypes, C++ will consider them as the overloaded functions.
- We cannot have a virtual constructor, but we can have a virtual destructor
- Consider the situation when we don't use the `virtual` keyword.

AD

```
#include <iostream>
```

```
using namespace std;  
  
class A  
{  
    int x=5;  
  
public:  
    void display()  
    {  
        std::cout << "Value of x is : " << x << std::endl;  
    }  
};  
  
class B: public A  
{  
    int y = 10;  
  
public:  
    void display()  
    {  
        std::cout << "Value of y is : " << y << std::endl;  
    }  
};  
  
int main()  
{  
    A *a;  
    B b;  
    a = &b;  
    a->display();  
    return 0;  
}
```

Output:

```
Value of x is : 5
```

In the above example, `* a` is the base class pointer. The pointer can only access the base class members but not the members of the derived class. Although C++ permits the base pointer to point to any object derived from the base class, it cannot directly access the members of the derived class. Therefore, there is a need for virtual function which allows the base pointer to access the members of the derived class.

C++ virtual function Example

Let's see the simple example of C++ virtual function used to invoked the derived class in a program.

```
#include <iostream>
{
public:
virtual void display()
{
    cout << "Base class is invoked" << endl;
}
};

class B:public A
{
public:
void display()
{
    cout << "Derived Class is invoked" << endl;
}
};

int main()
{
    A* a; //pointer of base class
    B b; //object of derived class
    a = &b;
    a->display(); //Late Binding occurs
}
```

Output:

```
Derived Class is invoked
```

Pure Virtual Function

- o A virtual function is not used for performing any task. It only serves as a placeholder.
- o When the function has no definition, such function is known as "**do-nothing**" function.

- The "**do-nothing**" function is known as a **pure virtual function**. A pure virtual function is a function declared in the base class that has no definition relative to the base class.
- A class containing the pure virtual function cannot be used to declare the objects of its own, such classes are known as abstract base classes.
- The main objective of the base class is to provide the traits to the derived classes and to create the base pointer used for achieving the runtime polymorphism.

Pure virtual function can be defined as:

```
virtual void display() = 0;
```

Let's see a simple example:

```
#include <iostream>
using namespace std;
class Base
{
public:
    virtual void show() = 0;
};

class Derived : public Base
{
public:
    void show()
    {
        std::cout << "Derived class is derived from the base class." << std::endl;
    }
};

int main()
{
    Base *bptr;
    //Base b;
    Derived d;
    bptr = &d;
    bptr->show();
    return 0;
}
```

{

Output:

Derived class is derived from the base class.

In the above example, the base class contains the pure virtual function. Therefore, the base class is an abstract base class. We cannot create the object of the base class.

← Prev

Next →

AD

 [For Videos Join Our Youtube Channel: Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share



Learn Latest Tutorials

Data Abstraction in C++

- Data Abstraction is a process of providing only the essential details to the outside world and hiding the internal details, i.e., representing only the essential details in the program.
- Data Abstraction is a programming technique that depends on the separation of the interface and implementation details of the program.
- Let's take a real life example of AC, which can be turned ON or OFF, change the temperature, change the mode, and other external components such as fan, swing. But, we don't know the internal details of the AC, i.e., how it works internally. Thus, we can say that AC separates the implementation details from the external interface.
- C++ provides a great level of abstraction. For example, pow() function is used to calculate the power of a number without knowing the algorithm the function follows.

In C++ program if we implement class with private and public members then it is an example of data abstraction.

Data Abstraction can be achieved in two ways:

- Abstraction using classes
- Abstraction in header files.



Abstraction using classes: An abstraction can be achieved using classes. A class is used to group all the data members and member functions into a single unit by using the access specifiers. A class has the responsibility to determine which data member is to be visible outside and which is not.

Abstraction in header files: An another type of abstraction is header file. For example, pow() function available is used to calculate the power of a number without actually knowing which algorithm function uses to calculate the power. Thus, we can say that header files hides all the implementation details from the user.

Access Specifiers Implement Abstraction:

- **Public specifier:** When the members are declared as public, members can be accessed anywhere from the program.
- **Private specifier:** When the members are declared as private, members can only be accessed only by the member functions of the class.

Let's see a simple example of abstraction in header files.

// program to calculate the power of a number.

```
#include <iostream>
#include<math.h>
using namespace std;
int main()
{
    int n = 4;
    int power = 3;
    int result = pow(n,power);      // pow(n,power) is the power function
    std::cout << "Cube of n is : " << result << std::endl;
    return 0;
}
```

Output:

```
Cube of n is : 64
```

In the above example, pow() function is used to calculate 4 raised to the power 3. The pow() function is present in the math.h header file in which all the implementation details of the pow() function is hidden.

Let's see a simple example of data abstraction using classes.

```
#include <iostream>
using namespace std;
class Sum
{
private: int x, y, z; // private variables
public:
```

```
void add()
{
    cout<<"Enter two numbers: ";
    cin>>x>>y;
    z= x+y;
    cout<<"Sum of two number is: "<<z<<endl;
}
};

int main()
{
    Sum sm;
    sm.add();
    return 0;
}
```

Output:

```
Enter two numbers:
3
6
Sum of two number is: 9
```

In the above example, abstraction is achieved using classes. A class 'Sum' contains the private members x, y and z are only accessible by the member functions of the class.

Advantages Of Abstraction:

- Implementation details of the class are protected from the inadvertent user level errors.
- A programmer does not need to write the low level code.
- Data Abstraction avoids the code duplication, i.e., programmer does not have to undergo the same tasks every time to perform the similar operation.
- The main aim of the data abstraction is to reuse the code and the proper partitioning of the code across the classes.
- Internal implementation can be changed without affecting the user level code.

[← Prev](#)[Next →](#)

AD

 [For Videos Join Our Youtube Channel: Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share



Learn Latest Tutorials

 [Splunk tutorial](#)

Splunk

 [SPSS tutorial](#)

SPSS

 [Swagger tutorial](#)

Swagger

 [T-SQL tutorial](#)

Transact-SQL

 [Tumblr tutorial](#)

Tumblr

 [React tutorial](#)

ReactJS

 [Regex tutorial](#)

Regex

 [Reinforcement learning tutorial](#)

Reinforcement Learning

 [R Programming tutorial](#) [RxJS tutorial](#)

RxJS

 [React Native tutorial](#) [Python Design Patterns](#)

[R Programming](#)[React Native](#)[Python Design Patterns](#)

Python Pillow



Python Turtle



Keras

Preparation



Aptitude

Logical Reasoning
ReasoningVerbal Ability
Verbal AbilityInterview Questions
Interview QuestionsCompany Interview
Questions

Company Questions

Trending Technologies

Artificial
Intelligence

Artificial
Intelligence

AWS

Selenium
tutorial

SeleniumCloud
Computing

Cloud Computing

Hadoop

ReactJS
Tutorial

ReactJSData Science
Tutorial

Data ScienceAngular 7
Tutorial

Angular 7

Blockchain



Git

Machine
Learning Tutorial

Machine LearningDevOps
Tutorial

DevOps

B.Tech / MCA

 DBMS tutorial DBMS	 Data Structures tutorial Data Structures	 DAA tutorial DAA	 Operating System Operating System
 Computer Network tutorial Computer Network	 Compiler Design tutorial Compiler Design	 Computer Organization and Architecture Computer Organization	 Discrete Mathematics Tutorial Discrete Mathematics
 Ethical Hacking Ethical Hacking	 Computer Graphics Tutorial Computer Graphics	 Software Engineering Software Engineering	 html tutorial Web Technology
 Cyber Security tutorial Cyber Security	 Automata Tutorial Automata	 C Language tutorial C Programming	 C++ tutorial C++
 Java tutorial Java	 .Net Framework tutorial .Net	 Python tutorial Python	 List of Programs Programs
 Control Systems tutorial Control System	 Data Mining Tutorial Data Mining	 Data Warehouse Tutorial Data Warehouse	



Interfaces in C++ (Abstract Classes)

Abstract classes are the way to achieve abstraction in C++. Abstraction in C++ is the process to hide the internal details and showing functionality only. Abstraction can be achieved by two ways:

1. **Abstract class**
2. **Interface**

Abstract class and interface both can have abstract methods which are necessary for abstraction.

C++ Abstract class

In C++ class is made abstract by declaring at least one of its functions as **pure virtual function**. A pure virtual function is specified by placing "`= 0`" in its declaration. Its implementation must be provided by derived classes.

Let's see an example of abstract class in C++ which has one abstract method `draw()`. Its implementation is provided by derived classes: `Rectangle` and `Circle`. Both classes have different implementation.

```
#include <iostream>
using namespace std;
class Shape
{
public:
    virtual void draw()=0;
};

class Rectangle : Shape
{
public:
    void draw()
    {
        cout << "drawing rectangle..." << endl;
    }
};

class Circle : Shape
{
public:
```

```
void draw()
{
    cout << "drawing circle..." << endl;
}
int main( ) {
    Rectangle rec;
    Circle cir;
    rec.draw();
    cir.draw();
    return 0;
}
```

Output:

```
drawing rectangle...
drawing circle...
```

← Prev

Next →

AD



For Videos Join Our Youtube Channel: [Join Now](#)

C++ Namespaces

Namespaces in C++ are used to organize too many classes so that it can be easy to handle the application.

For accessing the class of a namespace, we need to use `namespaceName::className`. We can use **using** keyword so that we don't have to use complete name all the time.

In C++, global namespace is the root namespace. The `global::std` will always refer to the namespace "std" of C++ Framework.

C++ namespace Example

Let's see the simple example of namespace which include variable and functions.

```
#include <iostream>
using namespace std;
namespace First {
    void sayHello() {
        cout<<"Hello First Namespace"<<endl;
    }
}

namespace Second {
    void sayHello() {
        cout<<"Hello Second Namespace"<<endl;
    }
}

int main()
{
    First::sayHello();
    Second::sayHello();
    return 0;
}
```

Output:

```
Hello First Namespace  
Hello Second Namespace
```

C++ namespace example: by using keyword

Let's see another example of namespace where we are using "using" keyword so that we don't have to use complete name for accessing a namespace program.

```
#include <iostream>  
using namespace std;  
namespace First{  
    void sayHello(){  
        cout << "Hello First Namespace" << endl;  
    }  
}  
namespace Second{  
    void sayHello(){  
        cout << "Hello Second Namespace" << endl;  
    }  
}  
using namespace First;  
int main () {  
    sayHello();  
    return 0;  
}
```

Output:

```
Hello First Namespace
```

← Prev

Next →

C++ Strings

In C++, string is an object of **std::string** class that represents sequence of characters. We can perform many operations on strings such as concatenation, comparison, conversion etc.

C++ String Example

Let's see the simple example of C++ string.

```
#include <iostream>
using namespace std;
int main() {
    string s1 = "Hello";
    char ch[] = { 'C', '+', '+'};
    string s2 = string(ch);
    cout << s1 << endl;
    cout << s2 << endl;
}
```

Output:

```
Hello
C++
```

C++ String Compare Example

Let's see the simple example of string comparison using `strcmp()` function.

```
#include <iostream>
#include <cstring>
using namespace std;
int main()
{
    char key[] = "mango";
    char buffer[50];
```

```
do {  
    cout<<"What is my favourite fruit? ";  
    cin>>buffer;  
} while (strcmp (key,buffer) != 0);  
cout<<"Answer is correct!!" << endl;  
return 0;  
}
```

Output:

```
What is my favourite fruit? apple  
What is my favourite fruit? banana  
What is my favourite fruit? mango  
Answer is correct!!
```

C++ String Concat Example

Let's see the simple example of string concatenation using strcat() function.

```
#include <iostream>  
#include <cstring>  
using namespace std;  
int main()  
{  
    char key[25], buffer[25];  
    cout << "Enter the key string: ";  
    cin.getline(key, 25);  
    cout << "Enter the buffer string: ";  
    cin.getline(buffer, 25);  
    strcat(key, buffer);  
    cout << "Key = " << key << endl;  
    cout << "Buffer = " << buffer << endl;  
    return 0;  
}
```

Output:

```
Enter the key string: Welcome to
Enter the buffer string: C++ Programming.
Key = Welcome to C++ Programming.
Buffer = C++ Programming.
```

AD

C++ String Copy Example

Let's see the simple example of copy the string using `strcpy()` function.

```
#include <iostream>
#include <cstring>
using namespace std;
int main()
{
    char key[25], buffer[25];
    cout << "Enter the key string: ";
    cin.getline(key, 25);
    strcpy(buffer, key);
    cout << "Key = " << key << endl;
    cout << "Buffer = " << buffer << endl;
    return 0;
}
```

Output:

```
Enter the key string: C++ Tutorial
Key = C++ Tutorial
Buffer = C++ Tutorial
```

C++ String Length Example

Let's see the simple example of finding the string length using `strlen()` function.

```
#include <iostream>
#include <cstring>
using namespace std;
int main()
{
    char ary[] = "Welcome to C++ Programming";
    cout << "Length of String = " << strlen(ary)<<endl;
    return 0;
}
```

Output:

```
Length of String = 26
```

C++ String Functions

Function	Description
int compare(const string& str)	It is used to compare two string objects.
int length()	It is used to find the length of the string.
void swap(string& str)	It is used to swap the values of two string objects.
string substr(int pos,int n)	It creates a new string object of n characters.
int size()	It returns the length of the string in terms of bytes.
void resize(int n)	It is used to resize the length of the string up to n characters.
string& replace(int pos,int len,string& str)	It replaces portion of the string that begins at character position pos and spans len characters.
string& append(const string& str)	It adds new characters at the end of another string object.
char& at(int pos)	It is used to access an individual character at specified position pos.
int find(string& str,int pos,int n)	It is used to find the string specified in the parameter.

<code>int find_first_of(string& str,int pos,int n)</code>	It is used to find the first occurrence of the specified sequence.
<code>int find_first_not_of(string& str,int pos,int n)</code>	It is used to search the string for the first character that does not match with any of the characters specified in the string.
<code>int find_last_of(string& str,int pos,int n)</code>	It is used to search the string for the last character of specified sequence.
<code>int find_last_not_of(string& str,int pos)</code>	It searches for the last character that does not match with the specified sequence.
<code>string& insert()</code>	It inserts a new character before the character indicated by the position pos.
<code>int max_size()</code>	It finds the maximum length of the string.
<code>void push_back(char ch)</code>	It adds a new character ch at the end of the string.
<code>void pop_back()</code>	It removes a last character of the string.
<code>string& assign()</code>	It assigns new value to the string.
<code>int copy(string& str)</code>	It copies the contents of string into another.
<code>char& back()</code>	It returns the reference of last character.
<code>Iterator begin()</code>	It returns the reference of first character.
<code>int capacity()</code>	It returns the allocated space for the string.
<code>const_iterator cbegin()</code>	It points to the first element of the string.
<code>const_iterator cend()</code>	It points to the last element of the string.
<code>void clear()</code>	It removes all the elements from the string.
<code>const_reverse_iterator crbegin()</code>	It points to the last character of the string.
<code>const_char* data()</code>	It copies the characters of string into an array.
<code>bool empty()</code>	It checks whether the string is empty or not.
<code>string& erase()</code>	It removes the characters as specified.
<code>char& front()</code>	It returns a reference of the first character.

<code>string& operator+=()</code>	It appends a new character at the end of the string.
<code>string& operator=()</code>	It assigns a new value to the string.
<code>char operator[](pos)</code>	It retrieves a character at specified position pos.
<code>int rfind()</code>	It searches for the last occurrence of the string.
<code>iterator end()</code>	It references the last character of the string.
<code>reverse_iterator rend()</code>	It points to the first character of the string.
<code>void shrink_to_fit()</code>	It reduces the capacity and makes it equal to the size of the string.
<code>char* c_str()</code>	It returns pointer to an array that contains null terminated sequence of characters.
<code>const_reverse_iterator crend()</code>	It references the first character of the string.
<code>reverse_iterator rbegin()</code>	It reference the last character of the string.
<code>void reserve(inr len)</code>	It requests a change in capacity.
<code>allocator_type get_allocator();</code>	It returns the allocated object associated with the string.

[← Prev](#)[Next →](#)

AD



C++ Exception Handling

Exception Handling in C++ is a process to handle runtime errors. We perform exception handling so the normal flow of the application can be maintained even after runtime errors.

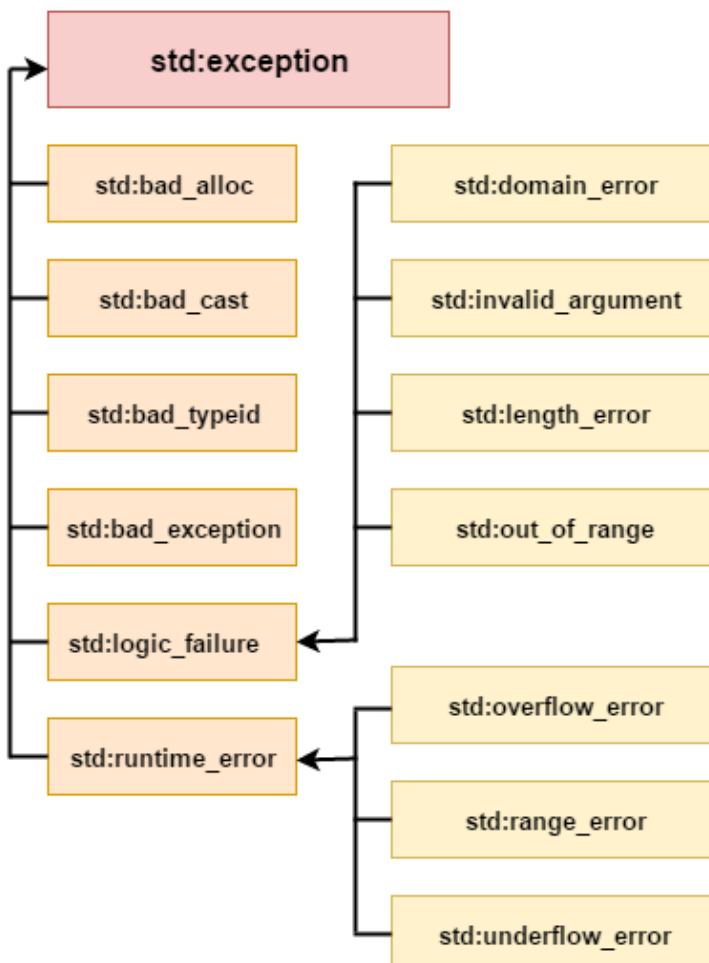
In C++, exception is an event or object which is thrown at runtime. All exceptions are derived from `std::exception` class. It is a runtime error which can be handled. If we don't handle the exception, it prints exception message and terminates the program.

Advantage

It maintains the normal flow of the application. In such case, rest of the code is executed even after exception.

C++ Exception Classes

In C++ standard exceptions are defined in `<exception>` class that we can use inside our programs. The arrangement of parent-child class hierarchy is shown below:



All the exception classes in C++ are derived from std::exception class. Let's see the list of C++ common exception classes.

Exception	Description
std::exception	It is an exception and parent class of all standard C++ exceptions.
std::logic_failure	It is an exception that can be detected by reading a code.
std::runtime_error	It is an exception that cannot be detected by reading a code.
std::bad_exception	It is used to handle the unexpected exceptions in a c++ program.
std::bad_cast	This exception is generally be thrown by dynamic_cast .
std::bad_typeid	This exception is generally be thrown by typeid .
std::bad_alloc	This exception is generally be thrown by new .

C++ Exception Handling Keywords

In C++, we use 3 keywords to perform exception handling:

- try
- catch, and
- throw

Moreover, we can create user-defined exception which we will learn in next chapters.

← Prev

Next →

C++ try/catch

In C++ programming, exception handling is performed using try/catch statement. The C++ **try block** is used to place the code that may occur exception. The **catch block** is used to handle the exception.

C++ example without try/catch

```
#include <iostream>
using namespace std;
float division(int x, int y) {
    return (x/y);
}
int main () {
    int i = 50;
    int j = 0;
    float k = 0;
    k = division(i, j);
    cout << k << endl;
    return 0;
}
```

Output:

```
Floating point exception (core dumped)
```

C++ try/catch example

```
#include <iostream>
using namespace std;
float division(int x, int y) {
    if( y == 0 ) {
        throw "Attempted to divide by zero!";
    }
}
```

```
return (x/y);  
}  
  
int main () {  
    int i = 25;  
    int j = 0;  
    float k = 0;  
  
    try {  
        k = division(i, j);  
        cout << k << endl;  
    }catch (const char* e) {  
        cerr << e << endl;  
    }  
    return 0;  
}
```

Output:

```
Attempted to divide by zero!
```

← Prev

Next →

AD



For Videos Join Our Youtube Channel: [Join Now](#)

C++ User-Defined Exceptions

The new exception can be defined by overriding and inheriting **exception** class functionality.

C++ user-defined exception example

Let's see the simple example of user-defined exception in which **std::exception** class is used to define the exception.

```
#include <iostream>
#include <exception>
using namespace std;
class MyException : public exception{
public:
    const char * what() const throw()
    {
        return "Attempted to divide by zero!\n";
    }
};

int main()
{
    try
    {
        int x, y;
        cout << "Enter the two numbers : \n";
        cin >> x >> y;
        if (y == 0)
        {
            MyException z;
            throw z;
        }
        else
        {
            cout << "x / y = " << x/y << endl;
        }
    }
    catch(exception& e)
```

```
{  
    cout << e.what();  
}  
}
```

Output:

```
Enter the two numbers :  
10  
2  
x / y = 5
```

Output:

```
Enter the two numbers :  
10  
0  
Attempted to divide by zero!
```

-->

Note: In above example what() is a public method provided by the exception class. It is used to return the cause of an exception.

← Prev

Next →

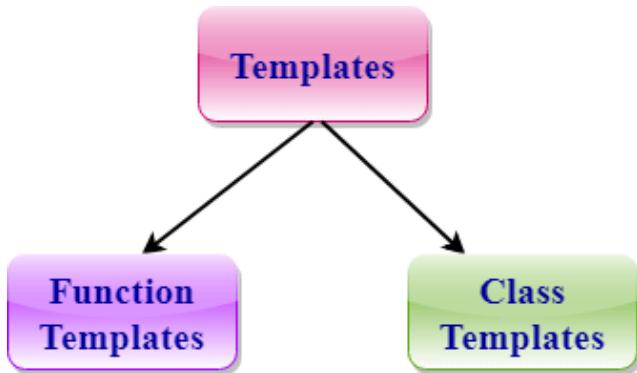
AD

C++ Templates

A C++ template is a powerful feature added to C++. It allows you to define the generic classes and generic functions and thus provides support for generic programming. Generic programming is a technique where generic types are used as parameters in algorithms so that they can work for a variety of data types.

Templates can be represented in two ways:

- Function templates
- Class templates



Function Templates:

We can define a template for a function. For example, if we have an add() function, we can create versions of the add function for adding the int, float or double type values.

Class Template:

We can define a template for a class. For example, a class template can be created for the array class that can accept the array of various types such as int array, float array or double array.

Function Template

- Generic functions use the concept of a function template. Generic functions define a set of operations that can be applied to the various types of data.
- The type of the data that the function will operate on depends on the type of the data passed as a parameter.
- For example, Quick sorting algorithm is implemented using a generic function, it can be implemented to an array of integers or array of floats.
- A Generic function is created by using the keyword template. The template defines what function will do.

AD



VIDEO STREAMING FOR THE AWAKENED MIND

GET STARTED NOW

Syntax of Function Template

```
template < class Ttype> ret_type func_name(parameter_list)
{
    // body of function.
}
```

Where **Ttype**: It is a placeholder name for a data type used by the function. It is used within the function definition. It is only a placeholder that the compiler will automatically replace this placeholder with the actual data type.

class: A class keyword is used to specify a generic type in a template declaration.

Let's see a simple example of a function template:

```
#include <iostream>
using namespace std;
template<class T> T add(T &a,T &b)
{
    T result = a+b;
    return result;
}

int main()
{
    int i =2;
    int j =3;
    float m = 2.3;
    float n = 1.2;
    cout<<"Addition of i and j is :" <<add(i,j);
    cout<<'\n';
    cout<<"Addition of m and n is :" <<add(m,n);
    return 0;
}
```

{

Output:

```
Addition of i and j is :5  
Addition of m and n is :3.5
```

In the above example, we create the function template which can perform the addition operation on any type either it can be integer, float or double.

Function Templates with Multiple Parameters

We can use more than one generic type in the template function by using the comma to separate the list.

Syntax

```
template<class T1, class T2,...>  
return_type function_name (arguments of type T1, T2....)  
{  
    // body of function.  
}
```

In the above syntax, we have seen that the template function can accept any number of arguments of a different type.

Let's see a simple example:

```
#include <iostream>  
using namespace std;  
template<class X, class Y> void fun(X a, Y b)  
{  
    std::cout << "Value of a is : " << a << std::endl;  
    std::cout << "Value of b is : " << b << std::endl;  
}  
  
int main()  
{
```

```
fun(15,12.3);
```

```
return 0;
}
```

Output:

```
Value of a is : 15
Value of b is : 12.3
```

In the above example, we use two generic types in the template function, i.e., X and Y.

Overloading a Function Template

We can overload the generic function means that the overloaded template functions can differ in the parameter list.



Let's understand this through a simple example:

```
#include <iostream>

using namespace std;

template<class X> void fun(X a)
{
    std::cout << "Value of a is : " <<a<< std::endl;
}

template<class X, class Y> void fun(X b ,Y c)
{
    std::cout << "Value of b is : " <<b<< std::endl;
    std::cout << "Value of c is : " <<c<< std::endl;
}

int main()
{
    fun(10);
    fun(20,30.5);
    return 0;
}
```

{

Output:

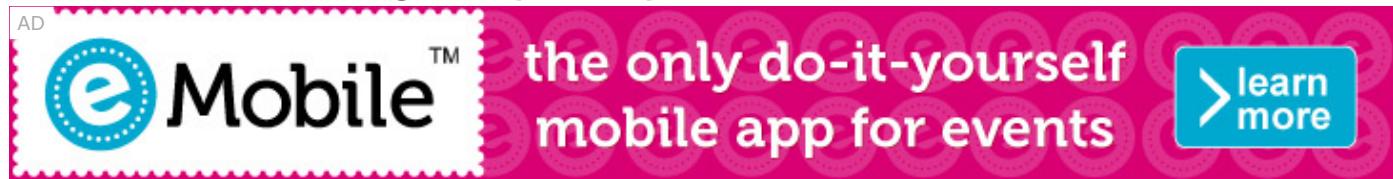
```
Value of a is : 10
Value of b is : 20
Value of c is : 30.5
```

In the above example, template of fun() function is overloaded.

Restrictions of Generic Functions

Generic functions perform the same operation for all the versions of a function except the data type differs. Let's see a simple example of an overloaded function which cannot be replaced by the generic function as both the functions have different functionalities.

Let's understand this through a simple example:



```
#include <iostream>
using namespace std;
void fun(double a)
{
    cout<<"value of a is :" <<a<<'\n';
}

void fun(int b)
{
    if(b%2==0)
    {
        cout<<"Number is even";
    }
    else
    {
        cout<<"Number is odd";
    }
}
```

```
}
```

```
int main()
{
    fun(4.6);
    fun(6);
    return 0;
}
```

Output:

```
value of a is : 4.6
Number is even
```

In the above example, we overload the ordinary functions. We cannot overload the generic functions as both the functions have different functionalities. First one is displaying the value and the second one determines whether the number is even or not.

CLASS TEMPLATE

Class Template can also be defined similarly to the Function Template. When a class uses the concept of Template, then the class is known as generic class.



Syntax

```
template<class Ttype>
class class_name
{
    .
    .
}
```

Ttype is a placeholder name which will be determined when the class is instantiated. We can define more than one generic data type using a comma-separated list. The Ttype can be used inside the class body.

Now, we create an instance of a class

```
class_name<type> ob;
```

where class_name: It is the name of the class.

type: It is the type of the data that the class is operating on.

ob: It is the name of the object.

Let's see a simple example:

```
#include <iostream>
using namespace std;
template<class T>
class A
{
public:
    T num1 = 5;
    T num2 = 6;
    void add()
    {
        std::cout << "Addition of num1 and num2 :" << num1+num2<<std::endl;
    }
};

int main()
{
    A<int> d;
    d.add();
    return 0;
}
```

Output:

```
Addition of num1 and num2 : 11
```

In the above example, we create a template for class A. Inside the main() method, we create the instance of class A named as, 'd'.

CLASS TEMPLATE WITH MULTIPLE PARAMETERS

We can use more than one generic data type in a class template, and each generic data type is separated by the comma.

Syntax

```
template<class T1, class T2, .....>
class class_name
{
    // Body of the class.
}
```

Let's see a simple example when class template contains two generic data types.

```
#include <iostream>
using namespace std;
template<class T1, class T2>
class A
{
    T1 a;
    T2 b;
public:
    A(T1 x,T2 y)
    {
        a = x;
        b = y;
    }
    void display()
    {
        std::cout << "Values of a and b are :" << a << "," << b << std::endl;
    }
}
```

```
};

int main()
{
    A<int,float> d(5,6.5);
    d.display();
    return 0;
}
```

Output:

```
Values of a and b are : 5,6.5
```

Nontype Template Arguments

The template can contain multiple arguments, and we can also use the non-type arguments. In addition to the type T argument, we can also use other types of arguments such as strings, function names, constant expression and built-in types. **Let's see the following example:**

```
template<class T, int size>
class array
{
    T arr[size];      // automatic array initialization.
};
```

In the above case, the nontype template argument is size and therefore, template supplies the size of the array as an argument.

Arguments are specified when the objects of a class are created:

```
array<int, 15> t1;          // array of 15 integers.
array<float, 10> t2;        // array of 10 floats.
array<char, 4> t3;          // array of 4 chars.
```

Let's see a simple example of nontype template arguments.

```
#include <iostream>
```

```
using namespace std;
template<class T, int size>
class A
{
public:
    T arr[size];
    void insert()
    {
        int i = 1;
        for (int j=0;j<size;j++)
        {
            arr[j] = i;
            i++;
        }
    }

    void display()
    {
        for(int i=0;i<size;i++)
        {
            std::cout << arr[i] << " ";
        }
    }
};

int main()
{
    A<int,10> t1;
    t1.insert();
    t1.display();
    return 0;
}
```

Output:

```
1 2 3 4 5 6 7 8 9 10
```

In the above example, the class template is created which contains the nontype template argument, i.e., size. It is specified when the object of class 'A' is created.

Points to Remember

- C++ supports a powerful feature known as a template to implement the concept of generic programming.
- A template allows us to create a family of classes or family of functions to handle different data types.
- Template classes and functions eliminate the code duplication of different data types and thus makes the development easier and faster.
- Multiple parameters can be used in both class and function template.
- Template functions can also be overloaded.
- We can also use nontype arguments such as built-in or derived data types as template arguments.

← Prev

Next →

AD

 [For Videos Join Our Youtube Channel: Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

C++ Signal Handling

- Signals are the interrupts which are delivered to a process by the operating system to stop its ongoing task and attend the task for which the interrupt has been generated.
- Signals can also be generated by the operating system on the basis of system or error condition.
- You can generate interrupts by pressing Ctrl+ C on Linux, UNIX, Mac OS X, or Windows system.

There are signals which cannot be caught by the program but there is a following list of signals which you can catch in your program and can take appropriate actions based on the signal.

These signals are defined in <csignal> header file.

Here are the list of signals along with their description and working capability:

Signals	Description
SIGABRT	(Signal Abort) Abnormal termination of the program, such as a call to abort.
SIGFPE	(Signal floating- point exception) An erroneous arithmetic operation, such as a divide by zero or an operation resulting in overflow.
SIGILL	(Signal Illegal Instruction) It is used for detecting an illegal instruction.
SIGINT	(Signal Interrupt) It is used to receipt an interactive program interrupt signal.
SIGSEGV	(Signal segmentation Violation) An invalid access to storage.
SIGTERM	(Signal Termination) A termination request sent to the program.
SIGHUP	(Signal Hang up) Hang Up (POSIX), its report that user's terminal is disconnected. It is used to report the termination of the controlling process.
SIGQUIT	Used to terminate a process and generate a core dump.
SIGTRAP	Trace trap.
SIGBUS	This is a BUS error which indicates an access to an invalid address.
SIGUSR1	User defined signal 1.
SIGUSR2	User defined signal 2.

SIGALRM	Alarm clock, which indicates an access to an invalid address.
SIGTERM	Used for termination. This signal can be blocked, handled, and ignored. Generated by kill command.
SIGCOUNT	This signal sent to process to make it continue.
SIGSTOP	Stop, unblockable. This signal is used to stop a process. This signal cannot be handled, ignored or blocked.

The signal() Function

C++ signal-handling library provides function signal to trap unexpected interrupts or events.

Syntax

```
void (*signal (int sig, void (*func)(int)))(int);
```

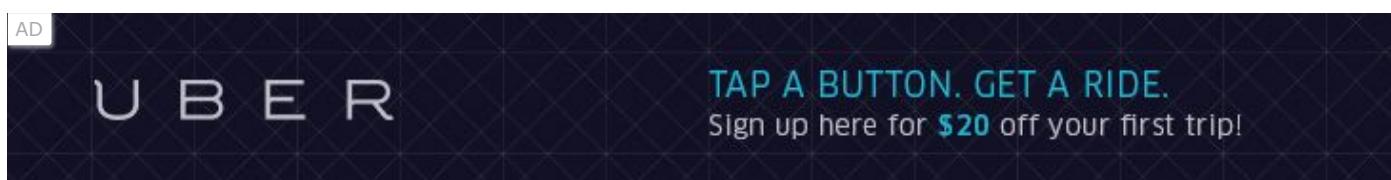
Parameters

This function is set to handle the signal.

It specifies a way to handle the signals number specified by **sig**.

Parameter **func** specifies one of the three ways in which a signal can be handled by a program.

- **Default handling (SIG_DFL):** The signal handled by the default action for that particular signal.
- **Ignore Signal (SIG_IGN):** The signal is ignored and the code execution will continue even if not purposeful.
- **Function handler:** A particular function is defined to handle the signal.



We must keep in mind that the signal that we would like to catch must be registered using a signal function and it must be associated with a signal handling function.

Note: The signal handling function should be of the void type.

Return value

The return type of this function is the same as the type of parameter func.

If the request of this function is successful, the function returns a pointer to the particular handler function which was in charge of handling this signal before the call, if any.

Data Races

Data race is undefined. If you call this function in a multi-threaded program then it will cause undefined behavior.

Exceptions

This function never throws exception.

Example 1

Let's see a simple example to demonstrate the use of signal() function:

```
#include <iostream>
#include <csignal>

using namespace std;

sig_atomic_t signalled = 0;

void handler(int sig)
{
    signalled = 1;
}

int main()
{
    signal(SIGINT, handler);

    raise(SIGINT);
    if (signalled)
        cout << "Signal is handled";
    else
```

```
    cout << "Signal is not handled";  
  
    return 0;  
}
```

Output:

```
Signal is handled
```

Example 2

Let's see another simple example:

```
#include <csignal>  
#include <iostream>  
  
namespace  
{  
    volatile std::sig_atomic_t gSignalStatus;  
  
    void signal_handler(int signal)  
    {  
        gSignalStatus = signal;  
    }  
  
    int main()  
    {  
        // Install a signal handler  
        std::signal(SIGINT, signal_handler);  
  
        std::cout << "SignalValue: " << gSignalStatus << '\n';  
        std::cout << "Sending signal " << SIGINT << '\n';  
        std::raise(SIGINT);  
        std::cout << "SignalValue: " << gSignalStatus << '\n';  
    }
```

Output:

```
SignalValue: 0
Sending signal 2
SignalValue: 2
```

The raise() Function

The C++ signal raise() function is used to send signals to the current executing program.



<csignal> header file declared the function raise() to handle a particular signal.

Syntax

```
int raise (int sig);
```

Parameters

sig: The signal number to be sent for handling. It can take one of the following values:

- SIGINT
- SIGABRT
- SIGFPE
- SIGILL
- SIGSEGV
- SIGTERM
- SIGHUP

Return value

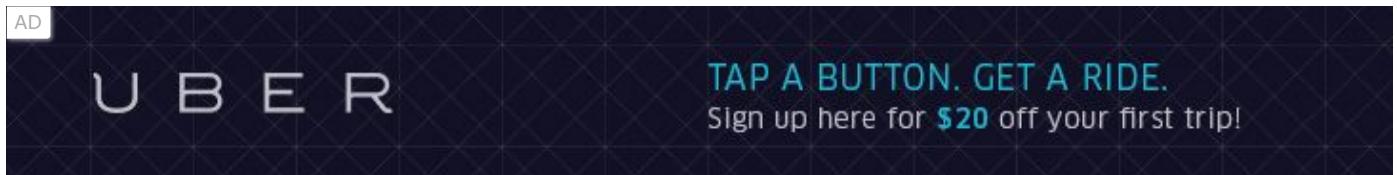
On success, it returns 0 and on failure, a non-zero is returned.

Data Races

Concurrently calling this function is safe, causing no data races.

Exceptions

This function never throws exceptions, if no function handlers have been defined with signal to handle the raised signal.



Example 1

Let's see a simple example to illustrate the use of raise() function when SIGABRT is passed:

```
#include <iostream>
#include <csignal>

using namespace std;

sig_atomic_t sig_value = 0;

void handler(int sig)
{
    sig_value = sig;
}

int main()
{
    signal(SIGABRT, handler);
    cout << "Before signal handler is called" << endl;
    cout << "Signal = " << sig_value << endl;
    raise(SIGABRT);
    cout << "After signal handler is called" << endl;
    cout << "Signal = " << sig_value << endl;

    return 0;
}
```

Output:

```
Before signal handler is called  
Signal = 0  
After signal handler is called  
Signal = 6
```

Example 2

Let's see a simple example to illustrate the use of raise() function when SIGINT is passed:

```
#include <csignal>  
#include <iostream>  
using namespace std;  
  
sig_atomic_t s_value = 0;  
void handle(int signal_){  
    s_value = signal_;  
}  
  
int main(){  
    signal(SIGINT, handle);  
    cout << "Before called Signal = " << s_value << endl;  
    raise(SIGINT);  
    cout << "After called Signal = " << s_value << endl;  
    return 0;  
}
```

Output:

```
Before called Signal = 0  
After called Signal = 2
```

Example 3

Let's see a simple example to illustrate the use of raise() function when SIGTERM is passed:

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

sig_atomic_t s_value = 0;

void handle(int signal_)

{

    s_value = signal_;

}

int main()

{

    signal(SIGTERM, handle);

    cout << "Before called Signal = " << s_value << endl;

    raise(SIGTERM);

    cout << "After called Signal = " << s_value << endl;

    return 0;

}
```

Output:

```
Before called Signal = 0
After called Signal = 15
```

Example 4

Let's see a simple example to illustrate the use of raise() function when SIGSEGV is passed:

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

sig_atomic_t s_value = 0;

void handle(int signal_)

{

    s_value = signal_;
```

```

}

int main()
{
    signal(SIGSEGV, handle);
    cout << "Before called Signal = " << s_value << endl;
    raise(SIGSEGV);
    cout << "After called Signal = " << s_value << endl;
    return 0;
}

```

Output:

```

Before called Signal = 0
After called Signal = 11

```

Example 5

Let's see a simple example to illustrate the use of `raise()` function when `SIGFPE` is passed:

```

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

sig_atomic_t s_value = 0;
void handle(int signal_)
{
    s_value = signal_;
}

int main()
{
    signal(SIGFPE, handle);
    cout << "Before called Signal = " << s_value << endl;
    raise(SIGFPE);
    cout << "After called Signal = " << s_value << endl;
    return 0;
}

```

{}

Output:

```
Before called Signal = 0
After called Signal = 8
```

[← Prev](#)[Next →](#)

AD

 [For Videos Join Our Youtube Channel: Join Now](#)**Feedback**

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share**Learn Latest Tutorials** [Splunk tutorial](#)

Splunk

 [SPSS tutorial](#)

SPSS

 [Swagger tutorial](#) [T-SQL tutorial](#)

Transact-SQL

C++ Files and Streams

In **C++ programming** we are using the **iostream** standard library, it provides **cin** and **cout** methods for reading from input and writing to output respectively.

To read and write from a file we are using the standard C++ library called **fstream**. Let us see the data types define in fstream library is:

Data Type	Description
fstream	It is used to create files, write information to files, and read information from files.
ifstream	It is used to read information from files.
ofstream	It is used to create files and write information to the files.

C++ FileStream example: writing to a file

Let's see the simple example of writing to a text file **testout.txt** using C++ FileStream programming.

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

int main () {
    ofstream filestream("testout.txt");
    if (filestream.is_open())
    {
        filestream << "Welcome to javaTpoint.\n";
        filestream << "C++ Tutorial.\n";
        filestream.close();
    }
    else cout <<"File opening is fail.";
    return 0;
}
```

Output:

```
The content of a text file testout.txt is set with the data:  
Welcome to javaTpoint.  
C++ Tutorial.
```

C++ FileStream example: reading from a file

Let's see the simple example of reading from a text file **testout.txt** using C++ FileStream programming.

```
#include <iostream>  
#include <fstream>  
using namespace std;  
int main () {  
    string srg;  
    ifstream filestream("testout.txt");  
    if (filestream.is_open())  
    {  
        while ( getline (filestream,srg) )  
        {  
            cout << srg << endl;  
        }  
        filestream.close();  
    }  
    else {  
        cout << "File opening is fail." << endl;  
    }  
    return 0;  
}
```

Note: Before running the code a text file named as "**testout.txt**" is need to be created and the content of a text file is given below:

Welcome to javaTpoint.
C++ Tutorial.

Output:

Welcome to javaTpoint.

C++ Tutorial.

C++ Read and Write Example

Let's see the simple example of writing the data to a text file **testout.txt** and then reading the data from the file using C++ FileStream programming.

```
#include <fstream>
#include <iostream>
using namespace std;
int main () {
    char input[75];
    ofstream os;
    os.open("testout.txt");
    cout << "Writing to a text file:" << endl;
    cout << "Please Enter your name: ";
    cin.getline(input, 100);
    os << input << endl;
    cout << "Please Enter your age: ";
    cin >> input;
    cin.ignore();
    os << input << endl;
    os.close();
    ifstream is;
    string line;
    is.open("testout.txt");
    cout << "Reading from a text file:" << endl;
    while (getline (is,line))
    {
        cout << line << endl;
    }
    is.close();
    return 0;
}
```

Output:

```
Writing to a text file:  
Please Enter your name: Nakul Jain  
Please Enter your age: 22  
Reading from a text file: Nakul Jain  
22
```

[← Prev](#)[Next →](#)

AD

 [YouTube](#) For Videos Join Our YouTube Channel: [Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share



Learn Latest Tutorials

[!\[\]\(505eadc5c89d19d2a7f925594f88bb0a_img.jpg\) Splunk tutorial](#)[!\[\]\(45b3afd6e8d78cdf9364ae22d3996c01_img.jpg\) SPSS tutorial](#)

C++ getline()

The `cin` is an object which is used to take input from the user but does not allow to take the input in multiple lines. To accept the multiple lines, we use the `getline()` function. It is a pre-defined function defined in a `<string.h>` header file used to accept a line or a string from the input stream until the delimiting character is encountered.

Syntax of `getline()` function:

There are two ways of representing a function:

- The first way of declaring is to pass three parameters.

```
istream& getline( istream& is, string& str, char delim );
```

The above syntax contains three parameters, i.e., **is**, **str**, and **delim**.

Where,

is: It is an object of the `istream` class that defines from where to read the input stream.

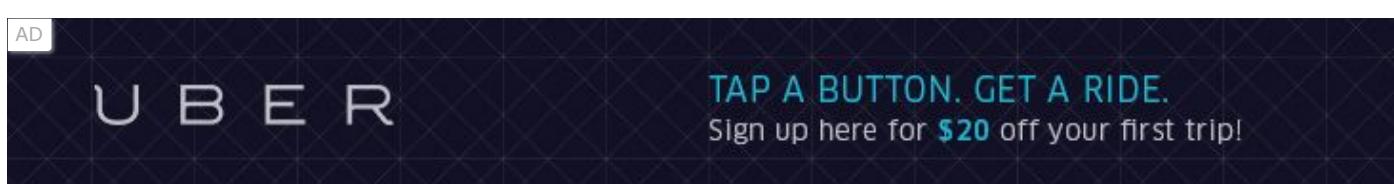
str: It is a `string` object in which string is stored.

delim: It is the delimiting character.

Return value

This function returns the input stream object, which is passed as a parameter to the function.

- The second way of declaring is to pass two parameters.



```
istream& getline( istream& is, string& str );
```

The above syntax contains two parameters, i.e., **is** and **str**. This syntax is almost similar to the above syntax; the only difference is that it does not have any delimiting character.

Where,

is: It is an object of the istream class that defines from where to read the input stream.

str: It is a string object in which string is stored.

Return value

This function also returns the input stream, which is passed as a parameter to the function.

Let's understand through an example.

First, we will look at an example where we take the user input without using getline() function.



```
#include <iostream>
#include<string.h>
using namespace std;
int main()
{
    string name; // variable declaration
    std::cout << "Enter your name :" << std::endl;
    cin>>name;
    cout<<"\nHello "<<name;
    return 0;
}
```

In the above code, we take the user input by using the statement **cin>>name**, i.e., we have not used the **getline()** function.

Output

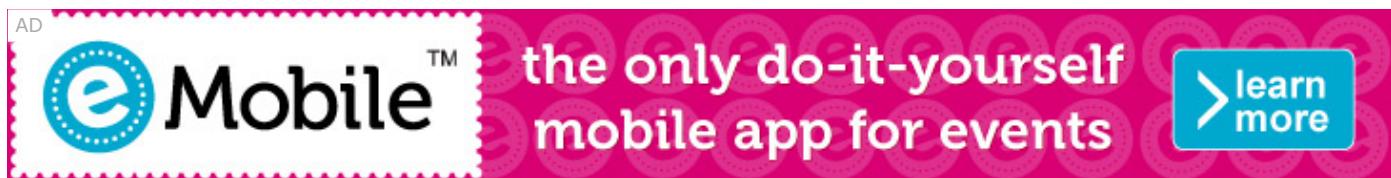
```
Enter your name :
John Miller
Hello John
```

In the above output, we gave the name 'John Miller' as user input, but only 'John' was displayed. Therefore, we conclude that cin does not consider the character when the space character is encountered.

Let's resolve the above problem by using getline() function.

```
#include <iostream>
#include<string.h>
using namespace std;
int main()
{
    string name; // variable declaration.
    std::cout << "Enter your name :" << std::endl;
    getline(cin, name); // implementing a getline() function
    cout << "\nHello " << name;
    return 0;
}
```

In the above code, we have used the **getline()** function to accept the character even when the space character is encountered.



Output

```
Enter your name :
John Miller
Hello John Miller
```

In the above output, we can observe that both the words, i.e., John and Miller, are displayed, which means that the **getline()** function considers the character after the space character also.

When we do not want to read the character after space then we use the following code:

```
#include <iostream>
#include<string.h>
using namespace std;
int main()
{
    string profile; // variable declaration
    std::cout << "Enter your profile :" << std::endl;
    getline(cin, profile, ' '); // implementing getline() function with a delimiting character.
    cout << "\nProfile is :" << profile;
```

{

In the above code, we take the user input by using getline() function, but this time we also add the delimiting character(" ") in a third parameter. Here, delimiting character is a space character, means the character that appears after space will not be considered.

Output

```
Enter your profile :  
Software Developer  
Profile is: Software
```

Getline Character Array

We can also define the getline() function for character array, but its syntax is different from the previous one.

Syntax

```
istream& getline(char* , int size);
```

In the above syntax, there are two parameters; one is **char***, and the other is **size**.

Where,

char*: It is a character pointer that points to the array.

Size: It acts as a delimiter that defines the size of the array means input cannot cross this size.

Let's understand through an example.

```
#include <iostream>  
#include<string.h>  
using namespace std;  
int main()  
{  
    char fruits[50]; // array declaration  
    cout<< "Enter your favorite fruit: ";  
    cin.getline(fruits, 50); // implementing getline() function  
    std::cout << "\nYour favorite fruit is :" << fruits << std::endl;
```

```
return 0;  
}
```

Output

```
Enter your favorite fruit: Watermelon  
Your favorite fruit is: Watermelon
```

← Prev

Next →

AD

 YouTube For Videos Join Our Youtube Channel: [Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

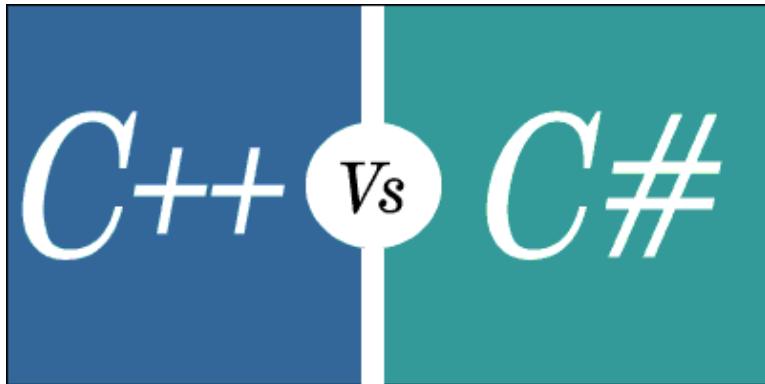
Help Others, Please Share



Learn Latest Tutorials

C++ vs C#

The following are the differences between C++ and C#:



- **Type of language**

C++ is a low-level language, while C# is a high-level language.

- **Lightweight language**

C++ is a lightweight language as compared to C# language as the libraries of C# language need to be included before compilation due to which size of binaries in C# language is more than C++ language.

- **Performance**

C++ code runs faster than the C# code and makes a better solution for those applications that require higher performance.

- **Garbage Collection**

C# provides the automatic garbage collection while C++ does not provide the automatic garbage collection, i.e., the objects are allocated or deallocated manually.

- **Platform dependency**

C# language is a standardized language so it works only on Windows operating system while C++ supports all the platforms such as Windows, Unix, Linux, Mac, etc.

- **Types of projects**

C++ language mainly works on those applications that communicate directly with the hardware while C# language is mainly used for mobile, web, desktop or gaming applications.

- **Compiler warnings**

C++ allows you to do everything if the syntax is correct, but sometimes cause real damage to the operating system. C# language is a much-protected language as compiler gives errors and warnings without allowing you to create serious damage.

- **Compilation**

C++ code is compiled to machine code C# code compiles to CLR(Common Language Runtime) which is interpreted by the JIT(Just In Time) compiler.

- o **Multiple Inheritance**

C++ language supports multiple inheritances, while C# language does not support the multiple inheritances.

- o **Level of Difficulty**

C++ language contains more complex features than C# language while C# language is a simple hierarchy which is quite easy to understand.

- o **Default access specifier**

In C++, the default access specifier is public while in C#, the default access specifier is private.

- o **Object Oriented**

C++ language is not a complete object-oriented language while C# language is a pure object-oriented programming language.

- o **Bound checking**

C++ language does not support the bound checking for arrays while C# language supports the bound checking for arrays.

- o **For each loop**

C++ language does not support the for each loop while C# language supports the for each loop.

- o **Use of pointers**

In C++, we can use the pointers anywhere in the program while in C# language, pointers are used in the unsafe area.

- o **Switch statement**

In C++, string variable cannot be passed in the switch statement, but in C# language, string variable can be passed in the switch statement.

- o **Standalone applications**

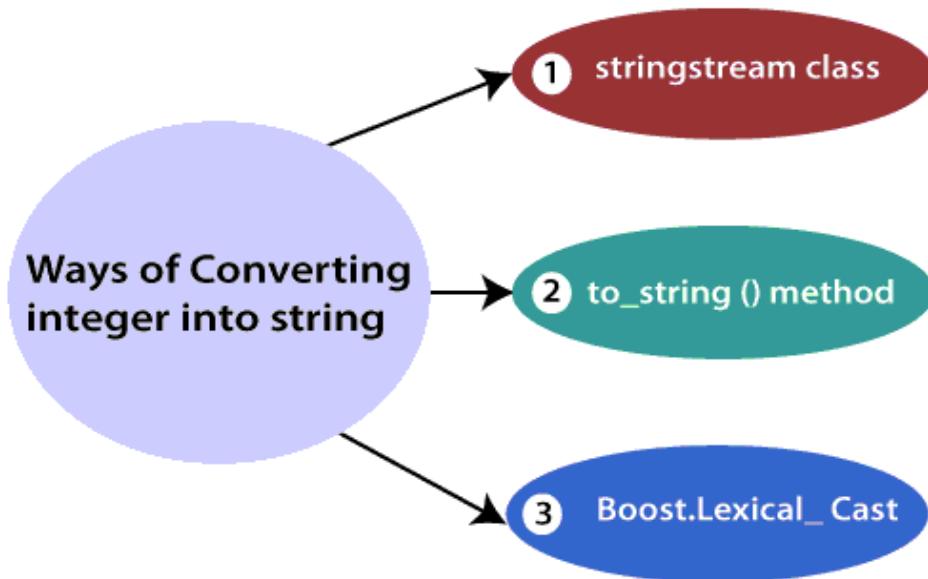
C++ language can be used to develop standalone applications, but C# language cannot be used to develop standalone applications.

← Prev

Next →

C++ int to string

There are three ways of converting an integer into a string:



- **By using stringstream class**
- **By using to_string() method**
- **By using boost.lexical cast**

Conversion of an integer into a string by using stringstream class.

The stringstream class is a stream class defined in the header file. It is a stream class used to perform the input-output operations on string-based streams.

The following are the operators used to insert or extract the data:

- **Operator >>**: It extracts the data from the stream.
- **Operator <<**: It inserts the data into the stream.



Let's understand the concept of operators through an example.

- In the below statement, the << insertion operator inserts the 100 into the stream.

```
stream1 << 100;
```

- In the below statement, the >> extraction operator extracts the data out of the stream and stores it in 'i' variable.

```
stream1 >> i;
```

Let's understand through an example.

```
#include <iostream>
#include<sstream>
using namespace std;
int main() {
    int k;
    cout<<"Enter an integer value";
    cin>>k;
    stringstream ss;
    ss<<k;
    string s;
    ss>>s;
    cout<<"\n" <<"An integer value is : "<<k<<"\n";
    cout<<"String representation of an integer value is : "<<s;
}
```

Output

```
clang version 7.0.0-3~ubuntu0.18.04.1 (tags/RELEASE_700/final)
> clang++-7 -pthread -o main main.cpp
> ./main
Enter an integer value 45

An integer value is : 45
String representation of an integer value is : 45
```

In the above example, we created the **k** variable, and want to convert the value of k into a string value. We have used the **stringstream** class, which is used to convert the k integer value into a string value. We can also achieve in vice versa, i.e., conversion of string into an integer value is also possible through the use of **stringstream** class only.

Let's understand the concept of conversion of string into number through an example.

```
#include <iostream>
#include<sstream>
using namespace std;
int main()
{
    string number = "100";
    stringstream ss;
    ss<<number;
    int i;
    ss>>i;
    cout<<"The value of the string is : "<<number<<"\n";
    cout<<"Integer value of the string is : "<<i;

}
```

Output

```
clang version 7.0.0-3~ubuntu0.18.04.1 (tags/RELEASE_700/final)
-> clang++-7 -pthread -o main main.cpp
-> ./main
The value of the string is : 100
Integer value of the string is : 100
```

Conversion of an integer into a string by using `to_string()` method.

The `to_string()` method accepts a single integer and converts the integer value or other data type value into a string.

Let's understand through an example:

```
#include <iostream>
#include<string>
using namespace std;
int main()
{
    int i=11;
    float f=12.3;
```

```
string str= to_string(i);
string str1= to_string(f);
cout<<"string value of integer i is :"<<str<<"\n";
cout<<"string value of f is : "<< str1;
}
```

Output

```
clang version 7.0.0-3~ubuntu0.18.04.1 (tags/RELEASE_700/
final)
> clang++-7 -pthread -o main main.cpp
> ./main
string value of integer i is :11
string value of f is : 12.300000
```

Conversion of an integer into a string by using a boost.lexical cast.

The boost.lexical cast provides a cast operator, i.e., boost.lexical_cast which converts the string value into an integer value or other data type value vice versa.

Let's understand the conversion of integer into string through an example.

```
#include <iostream>
#include <boost/lexical_cast.hpp>
using namespace std;
int main()
{
    int i=11;
    string str = boost::lexical_cast<string>(i);
    cout<<"string value of integer i is :"<<str<<"\n";
}
```

Output

```
clang version 7.0.0-3~ubuntu0.18.04.1 (tags/RELEASE_700/final)
> clang++-7 -pthread -o main main.cpp
> ./main
string value of integer i is :11
```

In the above example, we have converted the value of 'i' variable into a string value by using `lexical_cast()` function.

Let's understand the conversion of string into integer through an example.



```
#include <iostream>
#include <boost/lexical_cast.hpp>
using namespace std;
int main()
{
    string s="1234";
    int k = boost::lexical_cast<int>(s);
    cout<<"Integer value of string s is :"<<k<<"\n";
}
```

Output

```
clang version 7.0.0-3~ubuntu0.18.04.1 (tags/RELEASE_700/final)
> clang++-7 -pthread -o main main.cpp
> ./main
Integer value of string s is : 1234
```

In the above example, we have converted the string value into an integer value by using `lexical_cast()` function.

C++ vs Python

What is C++?

C++ is a high-level and general-purpose programming language developed by Bjarne Stroustrup in 1979. It is an extension C programming language, i.e., C with classes. The concept of object-oriented programming was first introduced in the C++ language. C++ is also known as an object-oriented programming language.

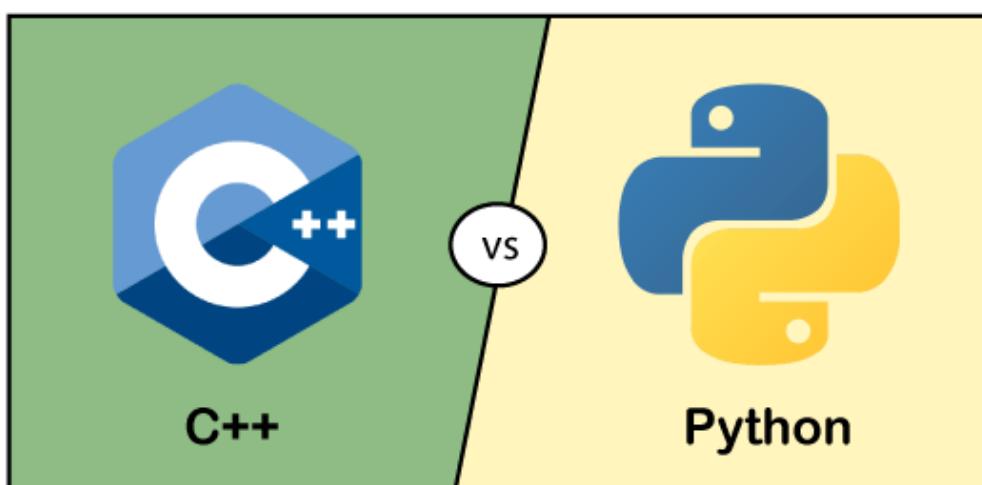
It was designed for system programming and embedded system, but later on, it was used in developing various applications such as desktop applications, video games, servers such as e-commerce, Web search or SQL servers and performance-critical applications such as telephone switches.

What is Python?

Python is a general-purpose and high-level programming language developed by Guido van Rossum in 1991. The main aim of developing python language was its simplicity. It contains features like indentation, which makes the code more readable, and it also contains library functions that make this language more powerful.

It is declared as a top language in IEEE's 2018 Top Programming Languages. Due to its popularity and simplicity makes python more powerful in the industry.

Differences b/w C++ and Python



Definition

C++ is a high-level and object-oriented programming language that allows you to do procedural programming, which is very close to CPU and provides full control over the hardware.

Python is an interpreted, high-level, and general-purpose programming language used to develop all types of projects.

Ease of Learning

One of the major factors for beginners is the ease of learning. If the programming language is hard, then it becomes difficult for the programmer to learn. The syntax of python is similar to English. Therefore, it is very easy to learn. On the other hand, C++ is based on the object-oriented concepts that deal with the memory allocation, if we write the wrong program in C++, then that can destroy the system also.

Speed

C++ is faster than the python programming language. Python is written in the C programming language, so memory management is very difficult in python. In C++, we can allocate the memory to the variables and can deallocate the memory when the variable is no longer used in the code.

Memory Management

In C++, we need to allocate the memory to the new variables and deallocate the memory whenever the variable is no longer required. If we do not do so, then it can lead to a memory leak. Therefore, we can say that C++ does not provide inbuilt garbage collection and dynamic memory management. On the other hand, python provides the inbuilt garbage collection and dynamic memory management mechanism, i.e., it allocates and deallocates the memory on its own.

Compilation

Python is an interpreted programming language, so it requires an interpreter at the time of compilation. On the other hand, C++ is a pre-compiled programming language, so it does not need any interpreter at the time of compilation.

Readability

C++ has a complex syntax, which is difficult to read and write. It follows the programming rules like we need to use the curly brackets and semicolon at the end of the statement. On the other hand, python does not follow these programming rules. It uses the indentation rules, which are similar to the English; this indentation allows the programmer to understand the code more easily.

Variable declaration

In C++, we need to declare the variable by mentioning the type and name of the variable before using it. Therefore, C++ is a statically typed programming language. On the other hand, python is a dynamically typed programming language, which means that we do not need to declare the variable before using that variable.

C++ Program

```
#include <iostream>
using namespace std;
int main()
{
    int a=20;
    std::cout << "value of a is : " <<a<< std::endl;
    return 0;
}
```

Python Program

```
# python program
#integer assignment
a=20
print(a)
```

In the above two programs, the output would be 20. The difference in the above two programs is that in C++, we need to declare the variable with its type, while in python, we do not need to declare the variables.

Functions

In C++, the function accepts and returns the type of value according to the definition, which is pre-defined. For example, suppose we have a function int add(int a, int b), then this function will accept only integer values as an argument and returns the integer type value. On the other hand, in python, there is no limitation on the type of the argument and type of its return value.

Let's summarize the above differences in a tabular form.



C++	Python
It is a high-level and pre-compiled programming language that allows you to do procedural programming.	It is a high-level and interpreted programming language used to develop all types of projects.

It is not easy to learn because of its complex syntax.	It is easy to learn, as it does not follow any programming rules. It follows the indentation rules, which is very much similar to English.
It does not contain a garbage collector.	It contains the garbage collector.
It is a pre-compiled programming language, so it does not require an interpreter during compilation.	It is an interpreted programming language, so it requires an interpreter to run the program.
It is a statically typed programming language.	It is a dynamically typed programming language.
Variable is declared by mentioning the type of the variable.	It does not require the declaration of a variable.
The function that accepts the value as an argument and returns the type of the value will depend on the definition of the function.	The function does not have any restriction on the type of the parameter and the return type.
Installation is easy.	It is not easy to install the python on Windows.
Variables inside the loop are not accessible outside the loop.	Variables inside the loop are also accessible outside the loop.
It has long lines of code as compared to Python.	It contains fewer lines of code compared to C++.
It supports both procedural and object-oriented programming.	It supports procedural, object-oriented, and functional programming.
It contains 52 keywords.	It contains 33 keywords.
In C++, the programmer needs to manually allocate the new variable and deallocate when no longer required.	Python performs the allocation.

[← Prev](#)[Next →](#)

Difference between Structure and Class in C++

In C++, the structure is the same as the class with some differences. Security is the most important thing for both structure and class. A structure is not safe because it could not hide its implementation details from the end-user, whereas a class is secure as it may hide its programming and design details. In this article, we are going to discuss the difference between a structure and class in C++. But before discussing the differences, we will know about the structure and class in C++.

What is the structure in C++?

A structure is a **grouping** of variables of various **data types** referenced by the same name. A structure declaration serves as a template for creating an instance of the structure.

Syntax:

The syntax of the structure is as follows:

```
Struct Structurename
{
    Struct_member1;
    Struct_member2;
    Struct_member3;
    .
    .
    .
    Struct_memberN;
};
```

The "**struct**" keyword indicates to the compiler that a structure has been declared. The "**structurename**" defines the name of the structure. Since the structure declaration is treated as a statement, so it is often ended by a semicolon.

What is Class in C++?

A class in C++ is similar to a C structure in that it consists of a list of **data members** and a set of operations performed on the class. In other words, a class is the **building block** of Object-Oriented programming. It is a user-defined object type with its own set of data members and member

functions that can be accessed and used by creating a class instance. A C++ class is similar to an object's blueprint.

Syntax:

The structure and the class are syntactically similar. The syntax of class in C++ is as follows:

```
class class_name
{
    // private data members and member functions.

    Access specifier;
    Data member;
    Member functions (member list){ . . }

};
```

In this syntax, the **class** is a keyword to indicate the compiler that a class has been declared. OOP's main function is data hiding, which is achieved by having three access specifiers: "**public**", "**private**", and "**safe**". If no access specifier is specified in the class when declaring data members or member functions, they are all considered private by default.

The public access specifier allows others to access program functions or data. A member of that class may reach only the class's private members. During inheritance, the safe access specifier is used. If the access specifier is declared, it cannot be changed again in the program.

Main differences between the structure and class

Here, we are going to discuss the main differences between the structure and class. Some of them are as follows:

- By default, all the members of the structure are public. In contrast, all members of the class are private.
- The structure will automatically initialize its members. In contrast, constructors and destructors are used to initialize the class members.
- When a structure is implemented, memory allocates on a stack. In contrast, memory is allocated on the heap in class.
- Variables in a structure cannot be initialized during the declaration, but they can be done in a class.
- There can be no null values in any structure member. On the other hand, the class variables may have null values.

- A structure is a value type, while a class is a reference type.
- Operators to work on the new data form can be described using a special method.



Head-to-head comparison between the structure and class

Here, we are going to discuss a head-to-head comparison between the structure and class. Some of them are as follows:

Features	Structure	Class
Definition	A structure is a grouping of variables of various data types referenced by the same name.	In C++, a class is defined as a collection of related variables and functions contained within a single structure.
Basic	If no access specifier is specified, all members are set to 'public'.	If no access specifier is defined, all members are set to 'private'.
Declaration	<pre>struct structure_name{ type struct_member 1; type struct_member 2; type struct_member 3; . type struct_memberN; };</pre>	<pre>class class_name{ data member; member function; };</pre>
Instance	Structure instance is called the 'structure variable'.	A class instance is called 'object'.
Inheritance	It does not support inheritance.	It supports inheritance.
Memory Allocated	Memory is allocated on the stack.	Memory is allocated on the heap.
Nature	Value Type	Reference Type

Purpose	Grouping of data	Data abstraction and further inheritance.
Usage	It is used for smaller amounts of data.	It is used for a huge amount of data.
Null values	Not possible	It may have null values.
Requires constructor and destructor	It may have only parameterized constructor.	It may have all the types of constructors and destructors.

Similarities

The following are similarities between the structure and class:

- Both class and structure may declare any of their members private.
- Both class and structure support inheritance mechanisms.
- Both class and structure are syntactically identical in C++.
- A class's or structure's name may be used as a stand-alone type.



Conclusion

Structure in C has some limitations, such as the inability to hide data, the inability to treat 'struct' data as built-in types, and the lack of inheritance support. The C++ structure overcame these drawbacks.

The extended version of the structure in C++ is called a class. The programmer makes it easy to use the class to hold both the data and functions, whereas the structure only holds data.

← Prev

Next →

Virtual Destructor in C++

A **destructor in C++** is a member function of a class used to free the space occupied by or delete an object of the class that goes out of scope. A destructor has the same name as the name of the constructor function in a class, but the destructor uses a tilde (~) sign before its function name.

Virtual Destructor

A virtual destructor is used to free up the memory space allocated by the derived class object or instance while deleting instances of the derived class using a base class pointer object. A base or parent class destructor use the **virtual** keyword that ensures both base class and the derived class destructor will be called at run time, but it called the derived class first and then base class to release the space occupied by both destructors.

Why we use virtual destructor in C++?

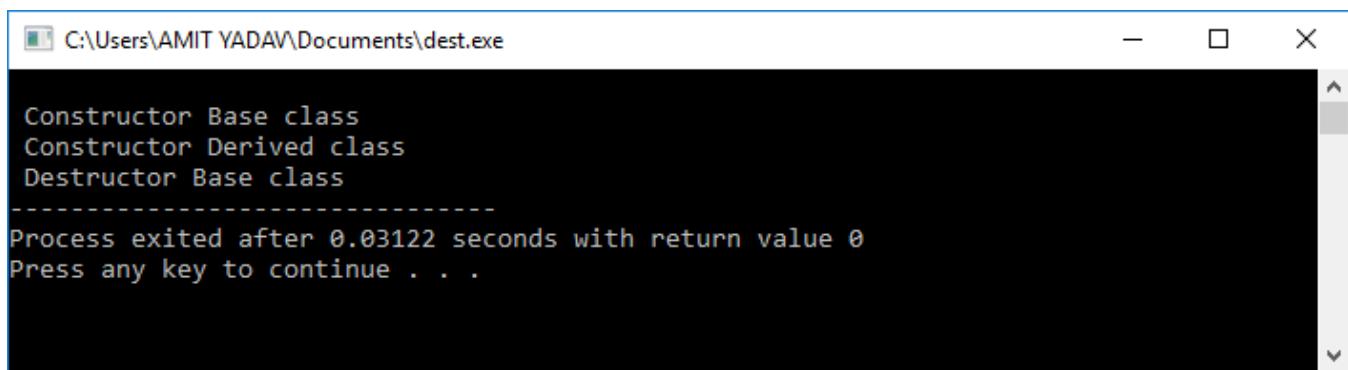
When an object in the class goes out of scope or the execution of the main() function is about to end, a destructor is automatically called into the program to free up the space occupied by the class' destructor function. When a pointer object of the base class is deleted that points to the derived class, only the parent class destructor is called due to the early bind by the compiler. In this way, it skips calling the derived class' destructor, which leads to memory leaks issue in the program. And when we use virtual keyword preceded by the destructor tilde (~) sign inside the base class, it guarantees that first the derived class' destructor is called. Then the base class' destructor is called to release the space occupied by both destructors in the inheritance class.

Write a program to display a class destructor's undefined behaviour without using a virtual destructor in C ++.

```
#include<iostream>
using namespace std;
class Base
{
public: /* A public access specifier defines Constructor and Destructor function to call by any obj
Base() // Constructor function.
{
    cout<< "\n Constructor Base class";
}
~Base() // Destructor function
```

```
{  
    cout<< "\n Destructor Base class";  
}  
};  
  
class Derived: public Base  
{  
    public: /* A public access specifier defines Constructor and Destructor function to call by any obj  
    Derived() // Constructor function  
{  
        cout << "\n Constructor Derived class" ;  
    }  
    ~Derived() // Destructor function  
{  
        cout << "\n Destructor Derived class" ; /* Destructor function is not called to release its space. */  
    }  
};  
int main()  
{  
    Base *bptr = new Derived; // Create a base class pointer object  
    delete bptr; /* Here pointer object is called to delete the space occupied by the destructor.*/  
}
```

Output:



The screenshot shows a terminal window titled 'C:\Users\AMIT YADAV\Documents\dest.exe'. The window displays the following text:
Constructor Base class
Constructor Derived class
Destructor Base class

Process exited after 0.03122 seconds with return value 0
Press any key to continue . . .

As we can see in the above output, when the compiler compiles the code, it calls a pointer object in the main function that refers to the base class. So, it executes the base class' constructor() function and then moves to the derived class' constructor() function. After that, it deletes the pointer object occupied by the base class' destructor and the derived class' destructor. The base class pointer only removes the base class's destructor without calling the derived class' destructor in the program. Hence, it leaks the memory in the program.

Note: If the base class destructor does not use a virtual keyword, only the base class destructor will be called or deleted its occupied space because the pointer object is pointing to the base class. So it does not call the derived class destructor to free the memory used by the derived class, which leads to memory leak for the derived class.

Write a program to display a class destructor's behavior using a virtual destructor in C++.

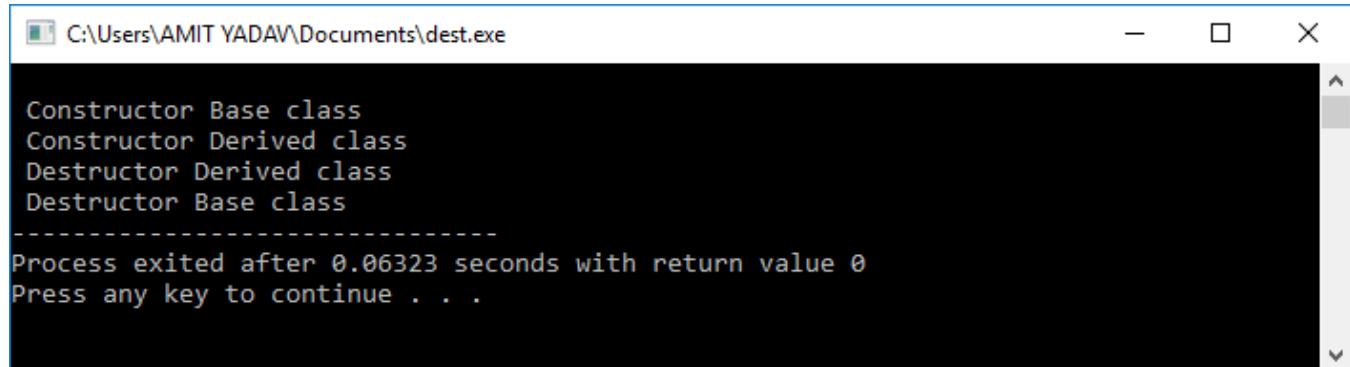
```
#include<iostream>
using namespace std;
class Base
{
public:
    Base() // Constructor member function.
    {
        cout << "\n Constructor Base class"; // It prints first.
    }
    virtual ~Base() // Define the virtual destructor function to call the Destructor Derived function.
    {
        cout << "\n Destructor Base class"; /
    }
};

// Inheritance concept
class Derived: public Base
{
public:
    Derived() // Constructor function.
    {
        cout << "\n Constructor Derived class" ; /* After print the Constructor Base, now it will prints. */
    }
    ~Derived() // Destructor function
    {
        cout << "\n Destructor Derived class"; /* The virtual Base Class? Destructor calls it before calling */
    }
};

int main()
```

```
{  
    Base *bptr = new Derived; // A pointer object reference the Base class.  
    delete bptr; // Delete the pointer object.  
}
```

Output:



```
C:\Users\AMIT YADAV\Documents\dest.exe  
Constructor Base class  
Constructor Derived class  
Destructor Derived class  
Destructor Base class  
-----  
Process exited after 0.06323 seconds with return value 0  
Press any key to continue . . .
```

In the above program, we have used a virtual destructor inside the base class that calls the derived class' destructor before calling the base class' destructor and release the spaces or resolve the memory leak issue in the program.

← Prev

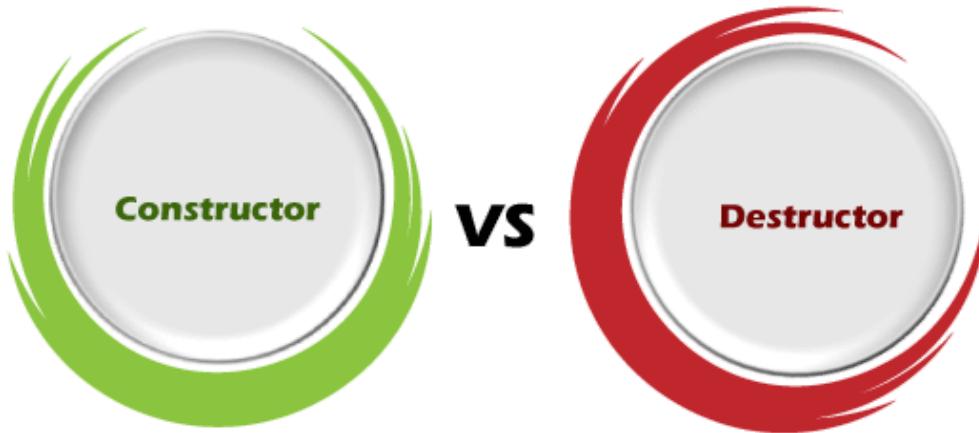
Next →

AD

 For Videos Join Our YouTube Channel: [Join Now](#)

Difference between Constructor and Destructor in C++?

In this article, we will see the comparisons between the constructors and destructors in the C++ programming language. The first thing that we will learn from this article is the basic idea of constructors and destructors. After that, we will learn the various comparisons of constructors and destructors in C++ programming.



What is C++?

C++ is a superset of C because it supports both procedural-oriented and object-oriented programming languages. It is a middle-level language. It has various features such as encapsulation, inheritance, abstraction, data hiding, constructor, and destructor.

Constructor in C++?

A **constructor** is a particular member function having the same name as the class name. It calls automatically whenever the object of the class is created.

Syntax:

The syntax of the constructor in C++ are given below.

```
class class_name
{
.....
public
class_name ([parameter list])
{
```

```
.....  
}  
};
```

In the above-given syntax, class_name is the constructor's name, the public is an access specifier, and the parameter list is optional.

Example of Constructor:

```
#include <iostream.h>  
#include <conio.h>  
using namespace std;  
class hello { // The class  
    public: // Access specifier  
    hello () { // Constructor  
        cout << "Hello World! Program in C++ by using Constructor";  
    }  
    void display() {  
        cout <<"Hello World!" << endl;  
    }  
};  
int main() {  
    hello myObj; /  
    return 0;  
}
```

There are four types of constructors used in C++.

- Default constructor
- Dynamic constructor
- Parameterized constructor
- Copy constructor

Default Constructor: A constructor is a class which accepts no parameter and is called a default constructor. If there is no constructor for a class, the compiler implicitly creates a default constructor.

Following is the syntax of the default constructor:

```
class class_name {
```

private:

.....

.....

public:

```
class_name ()
```

```
{
```

.....

```
}
```

```
}
```

In this type of constructor, there are no arguments and parameter lists.

If no constructor is defined in the class, the compiler automatically creates the class's default constructor.

Example:

```
class student {
```

private:

.....

.....

public:

```
student ()
```

```
{
```

.....

```
}
```

```
}
```

The default constructor for a class student is given below:

```
hello::hello()
```

Parameterised Constructor: A constructor is a class that can take parameters and is called a parameterized constructor. It is used to initialize objects with a different set of values.

Syntax of the Parameterised constructor is given below.

```

Class classname
{
....;
....;
Public:
Class name (parameter list)
{
....;
}
};

```

Here, we can define the parameter list of the constructor.

Copy Constructor: A particular constructor used for the creation of an existing object. The copy constructor is used to initialize the thing from another of the same type.

Syntax:

AD



VIDEO STREAMING FOR THE AWAKENED MIND

GET STARTED NOW

Syntax of the copy constructor is given below.

```

Class (classname, &object)
{
....;
....;
}

```

In the above syntax, the object refers to a thing used to initialize another object.

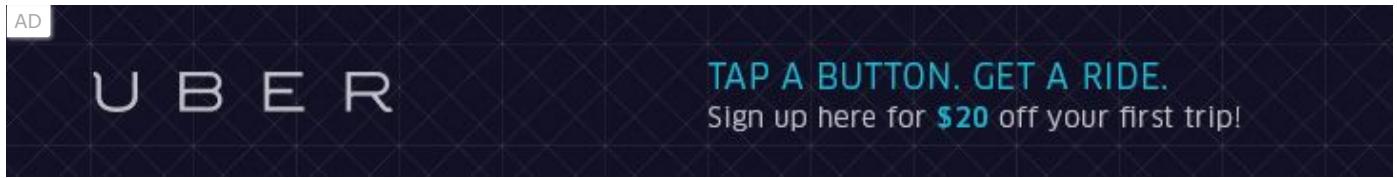
Dynamic Constructor: This type of constructor can be used to allocate the memory while creating the objects. The data members of an object after creation can be initialized, called dynamic initialization.

Destructor in C++?

Destructors have the same class name preceded by (~) tilde symbol. It removes and destroys the memory of the object, which the constructor allocated during the creation of an object.

Syntax:

The syntax of destructor in C++ are given below.



```
class class_name
{
....;
....;
public:
xyz();      //constructor
~xyz();     //destructor
};
```

Here, we use the tilde symbol for defining the destructor in C++ programming.

The Destructor has no argument and does not return any value, so it cannot be overloaded.

Example of Destructor:

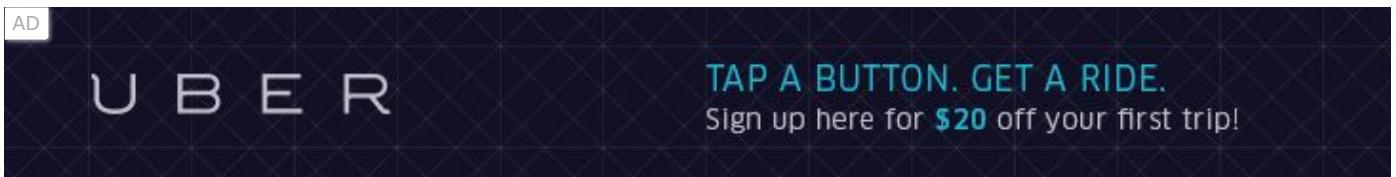
```
#include <iostream.h>
#include <conio.h>
using namespace std;
class Hello {
public:
//Constructor
Hello () {
    cout<< "Constructor function is called" <<endl;
}
//Destructor
~Hello () {
    cout << "Destructor function is called" <<endl;
}
//Member function
void display() {
    cout << "Hello World!" <<endl;
```

```

    }
};

int main(){
    //Object created
    Hello obj;
    //Member function called
    obj.display();
    return 0;
}

```



Difference between Constructor and Destructor in C++ programming

Following table shows the various differences between constructor and destructor in the C++ programming language:

Basis	Constructor	Destructor
Purpose of use	To allocate memory to the object, we used a constructor in C++.	To deallocate the memory that the constructor allocated to an object for this purpose, we use the concept of destructor in C++.
Arguments	It may or may not contain arguments.	It cannot contain the arguments.
Calling	It is called automatically whenever the object of the class is created.	It is called automatically whenever the program terminates.
Memory	Constructor occupies memory.	The Destructor releases memory.
Return type	It has return types.	It doesn't have any return type.
Special symbol	While declaring constructor in the C++ programming language, there is no requirement of the special symbol.	While declaring a destructor in C++ programming language, a particular symbol is required, i.e., tilde symbol.

In numbers	We can use more than one constructor in our program.	We cannot use more than one destructor in the program.
Inheritance	It can be inherited.	It cannot be inherited.
Overloading	It can be overloaded.	It cannot be overloaded.
Execution Order	They are executed in successive order.	They are executed in the constructor's reverse order; basically, they are the inverse of the constructors.
Types	<p>Constructor has four types:</p> <ul style="list-style-type: none"> o Default constructor o Copy constructor o Parameterized constructor o Dynamic constructor 	Destructors have no classes.
Declaration	<p>The following declaration is used for creating a constructor:</p> <pre>class class_name { public class_name ([parameter list]) { };</pre>	<p>The following declaration is used for creating a destructor:</p> <pre>class class_name { public: ~xyz(); {..... };</pre>

[← Prev](#)[Next →](#)

Bit manipulation C++

The computer does not understand the high-level language in which we communicate. For these reasons, there was a standard method by which any instruction given to the computer was understood. At an elementary level, each instruction was sent into some digital information known as bits. The series of bits indicates that it is a particular instruction.



Bit

A bit is defined as the basic unit which stores the data in digital notation.

Two values represent it as follows -

1 - It indicates the signal is present or True

0 - It indicates the signal is absent or False

Bits represent the logical state of any instruction. The series of bits have a base which is 2. Thus if we say if we have a series of binary digits, it is read from left to right, and the power of 2 increases.

$$\begin{array}{r} 1 \ 0 \ 1 \ 0 \\ 2^3 \ 2^2 \ 2^1 \ 2^0 \end{array}$$

A binary digit

So after understanding the basics of bit, let us understand its manipulation in C++.

Bit manipulation

Bit manipulation is defined as performing some basic operations on bit level of n number of digits. It is a speedy and primitive method as it directly works at the machine end.

With that, let us get into the basics of bit manipulation in C++.

- o **Logical AND**

Logical AND takes two operands and returns true if both of them are true. The sign is `&&`.

Let us look at the truth table of the AND operator.

A	B	A&&B
0	0	0
0	1	0
1	0	0
1	1	1

In the last row, A and B are high, resulting in a high output.

C++ Program

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

int main() {
    int a = 5;
    int b = 9;

    // false && false = false
    cout << ((a == 0) && (a > b)) << endl;

    // false && true = false
    cout << ((a == 0) && (a < b)) << endl;

    // true && false = false
    cout << ((a == 5) && (a > b)) << endl;

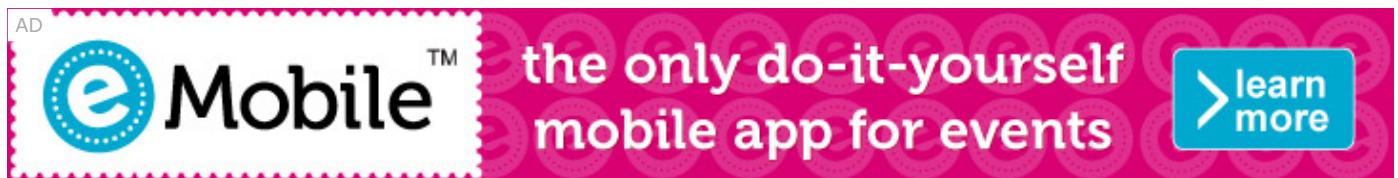
    // true && true = true
    cout << ((a == 5) && (a < b)) << endl;

    return 0;
}
```

}

Output:

```
Output
/tmp/wP5fhZeJmr.o
0
0
0
1
```

o **Logical OR**

Logical OR gives us high output if either of the input of the two operands is high. The symbol is ||

Let us look at the truth table of the OR operator.

A	B	A B
0	0	0
0	1	1
1	0	1
1	1	1

Here we can see the first row. Both inputs A and B are low, which results in 0(a low output).

**C++ Program**

```
#include <iostream>
```

```
using namespace std;
```

```
int main() {
    int a = 5;
    int b = 9;

    // false && false = false
    cout << ((a == 0) || (a > b)) << endl;

    // false && true = true
    cout << ((a == 0) || (a < b)) << endl;

    // true && false = true
    cout << ((a == 5) || (a > b)) << endl;

    // true && true = true
    cout << ((a == 5) || (a < b)) << endl;

    return 0;
}
```

Output:

```
Output
/tmp/wP5fhZeJmr.o
0
1
1
1
|
```

- o **Logical NOT**

Logical NOT is taking only one operand and reverts it. If the operand is low, it makes it high and vice versa. The symbol is !.

Let us look at the truth table of the NOT operator.

A	!A
0	1
1	0

C++ Program



```
#include <iostream>

using namespace std;

int main() {
    int a = 5;

    // !false = true
    cout << !(a == 0) << endl;

    // !true = false
    cout << !(a == 5) << endl;

    return 0;
}
```

Output:

Output

```
/tmp/wP5fhZeJmr.o
1
0
```

- o **Left shift operator**

The left shift operator takes an operand and the value of the left operand is moved left by the number of bits specified by the right operand.

It is denoted by `<<`.

C++ Program

```
#include<iostream>
using namespace std;
int main()
{
    // a = 5(00000101), b = 9(00001001)
    unsigned char a = 5, b = 9;

    // The result is 00001010
    cout << "a<<1: " << (a<<1) << "\n";

    // The result is 00010010
    cout << "b<<1: " << (b<<1);
    return 0;
}
```

Output:

Output

```
/tmp/wP5fhZeJmr.o
a<<1: 10
b<<1: 18
```

- o **Right shift operator**

The right shift operator takes an operand and the value of the right operand is moved right by the number of bits specified by the right operand.

It is denoted by `>>`.

C++ Program

```
#include <bits/stdc++.h>
using namespace std;
int main()
{
    // a = 5(00000101), b = 9(00001001)
    unsigned char a = 5, b = 9;

    // The result is 00000010

    cout<< "a>>1: " << (a >> 1) << "\n";
    cout<< "b>>1: " << (b >> 1);
    return 0;
}
```

Output:

Output

```
/tmp/wP5fhZeJmr.o
a>>1: 2
b>>1: 4
```

← Prev

Next →

What is a reference variable?

A reference is defined as an alias for another variable. In short, it is like giving a different name to a pre-existing variable. Once a reference is initialized to the variable, we can use either the reference name or the variable to refer to that variable.

Creating references in C++

The basic syntax to create a reference is -

Data type**&** new variable = previous variable

The newly created variable will now refer to the previous variable.

For example -

int i = 17 // The variable i is declared as 17

Creating the reference of i will be as -

int& x = i // Here x will be called as the integer variable initialised to r

C++ code

```
#include <iostream>

using namespace std;

int main () {
    int i; // Declare variable I as int
    double d; // Declare variable d as double type

    // declare reference variables for I and d
    int& r = i;// r is reference to i
    double& s = d;// s is reference to d

    i = 5;
    cout << "Value of i : " << i << endl;
    cout << "Value of i reference : " << r << endl;
```

```
d = 11.7;  
cout << "Value of d : " << d << endl;  
cout << "Value of d reference : " << s << endl;  
  
return 0;  
}
```

Output

```
Output  
  
/tmp/0JkPt1VevX.o  
Value of i : 5  
Value of i reference : 5  
Value of d : 11.7  
Value of d reference : 11.7
```

Difference between Reference and Pointers

References	Pointers
We cannot have a NULL reference.	The concept of NULL pointers is allowed.
A reference assigned to a particular object can't be changed.	Pointers, on the other hand, can point to different objects at any time.
A reference is also initialized at the time of its creation.	Pointers can be initialized at any time.

← Prev

Next →

Friend Function in C++

As we already know that in an object-oriented programming language, the members of the class can only access the data and functions of the class but outside functions cannot access the data of the class. There might be situation that the outside functions need to access the private members of the class so in that case, friend function comes into the picture.

What is a friend function?

A friend function is a function of the class defined outside the class scope but it has the right to access all the private and protected members of the class.

The friend functions appear in the class definition but friends are the member functions.

Characteristics of a Friend function:

- The function is not in the scope of the class to which it has been declared as a friend.
- It cannot be called using the object as it is not in the scope of that class.
- It can be invoked like a normal function without using the object.
- It cannot access the member names directly and has to use an object name and dot membership operator with the member name.
- It can be declared either in the private or the public part.

Why do we need a friend function in C++?

- Friend function in C++ is used when the class private data needs to be accessed directly without using object of that class.
- Friend functions are also used to perform operator overloading. As we already know about the function overloading, operators can also be overloaded with the help of operator overloading.



Characteristics of a Friend function

- The friend function is declared using friend keyword.
- It is not a member of the class but it is a friend of the class.

- As it is not a member of the class so it can be defined like a normal function.
- Friend functions do not access the class data members directly but they pass an object as an argument.
- It is like a normal function.
- If we want to share the multiple class's data in a function then we can use the friend function.

Syntax for the declaration of a friend function.

```
class class_name
{
    friend data_type function_name(argument/s);      // syntax of friend function
};
```

In the above declaration, the friend function is preceded by the keyword friend. The function can be defined anywhere in the program like a normal C++ function. The function definition does not use either the keyword **friend or scope resolution operator**.

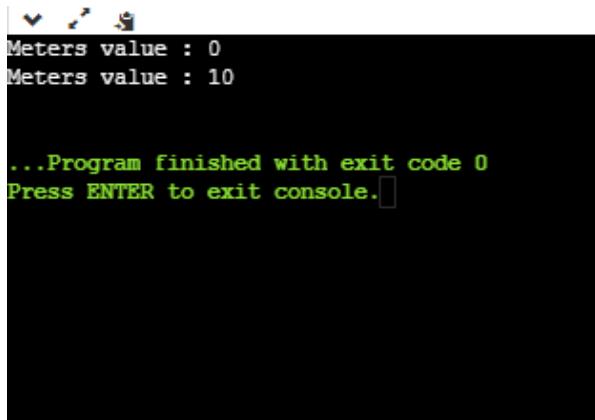
Let's understand the friend function through an example.

```
#include <iostream>
using namespace std;
class Distance
{
    private:
        int meters;
    public:
        // constructor
        Distance()
        {
            meters = 0;
        }
        // definition of display_data() method
        void display_data()
        {
            std::cout << "Meters value : " << meters << std::endl;
        }
}
```

```
//prototype of a friend function.  
friend void addvalue(Distance &d);  
  
};  
  
// Definition of friend function  
void addvalue(Distance &d) // argument contain the reference  
{  
    d.meters = d.meters+10; // incrementing the value of meters by 10.  
}  
  
// main() method  
int main()  
{  
    Distance d1; // creating the object of class distance.  
    d1.display_data(); // meters = 0  
    addvalue(d1); // calling friend function  
    d1.display_data(); // meters = 10  
    return 0;  
}
```

In the above code, **Distance** is the class that contains private field named as '**meters**'. The **Distance()** is the constructor method that initializes the 'meters' value with 0. The **display_data()** is a method that displays the 'meters' value. The **addvalue()** is a friend function of Distance class that modifies the value of '**meters**'. Inside the **main()** method, d1 is an object of a Distance class.

Output



```
Meters value : 0
Meters value : 10

...Program finished with exit code 0
Press ENTER to exit console.
```

Friend function can also be useful when we are working on objects of two different classes.

Let's understand through an example.

```
// Add members of two different classes using friend functions
```

```
#include <iostream>
```

```
using namespace std;
```

```
// forward declaration of a class
```

```
class ClassB;
```

```
// declaration of a class
```

```
class ClassA {
```

```
public:
```

```
    // constructor ClassA() to initialize num1 to 12
```

```
    ClassA()
```

```
{
```

```
    num1 = 12;
```

```
}
```

```
private:
```

```
    int num1; // declaration of integer variable
```

```
    // friend function declaration
```

```
    friend int multiply(ClassA, ClassB);
```

```
};
```

```
class ClassB {
```

```
public:
```

```
    // constructor ClassB() to initialize num2 to 2
```

```
    ClassB()
```

```
{
```

```
    num2 = 2;
```

```
}
```

```
private:
```

```
    int num2; // declaration of integer variable
```

```
    // friend function declaration
```

```
    friend int multiply(ClassA, ClassB);
```

```
};
```

```
// access members of both classes
```

```
int multiply(ClassA object1, ClassB object2)
```

```
{
```

```

return (object1.num1 * object2.num2);

}

int main() {
    ClassA object1; // declaration of object of ClassA
    ClassB object2; // declaration of object of ClassB
    cout << "Result after multiplication of two numbers is : " << multiply(object1, object2);
    return 0;
}

```

In the above code, we have defined two classes named as **ClassA** and **ClassB**. Both these classes contain the friend function '**multiply()**'. The friend function can access the data members of both the classes, i.e., **ClassA** and **ClassB**. The **multiply()** function accesses the **num1** and **num2** of **ClassA** and **ClassB** respectively. In the above code, we have created object1 and object2 of ClassA and ClassB respectively. The multiply() function multiplies the num1 and num2 and returns the result.

As we can observe in the above code that the friend function in **ClassA** is also using **ClassB** without prior declaration of **ClassB**. So, in this case, we need to provide the forward declaration of **ClassB**.

Output

```

Result after multiplication of two numbers is : 24
...Program finished with exit code 0
Press ENTER to exit console.

```

AD



VIDEO STREAMING FOR THE AWAKENED MIND

GET STARTED NOW

Friend class in C++

We can also create a friend class with the help of **friend** keyword.

```

class Class1;

class Class2
{
    // Class1 is a friend class of Class2
}

```

```
friend class Class1;  
... ....  
}  
  
class Class1  
{  
...  
}
```

In the above declaration, Class1 is declared as a friend class of Class2. All the members of Class2 can be accessed in Class1.

Let's understand through an example.

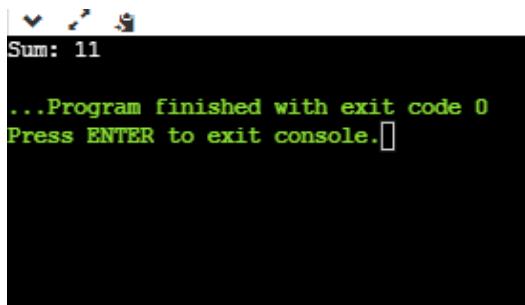


```
// C++ program to demonstrate the working of friend class  
#include <iostream>  
  
using namespace std;  
  
// forward declaration  
class ClassB;  
  
  
class ClassA {  
private:  
    int num1;  
  
    // friend class declaration  
    friend class ClassB;  
  
  
public:  
    // constructor to initialize numA to 10  
    ClassA()  
    {  
        num1 = 10;  
    }  
};
```

```
class ClassB {  
    private:  
        int num2;  
  
    public:  
        // constructor to initialize numB to 1  
        ClassB()  
        {  
            num2 = 1;  
        }  
  
        // member function to add num1  
        // from ClassA and num2 from ClassB  
        int add() {  
            ClassA objectA;  
            return objectA.num1 + num2;  
        }  
};  
  
int main() {  
    ClassB objectB;  
    cout << "Sum: " << objectB.add();  
    return 0;  
}
```

In the above code, we have created two classes named as ClassA and ClassB. Since ClassA is declared as friend of ClassB, so ClassA can access all the data members of ClassB. In ClassB, we have defined a function add() that returns the sum of num1 and num2. Since ClassB is declared as friend of ClassA, so we can create the objects of ClassA in ClassB.

Output



```
Sum: 11  
...Program finished with exit code 0  
Press ENTER to exit console.[]
```

Snake Code in C++

In this article, we will create a snake game with the help of C++ and graphics functions. In this, we will use the concept of c++ classes and computer graphics functions.

What do you mean by the Snake game?

Snake game is one of the most famous games available on all types of device and works on every operating system. Snakes can move in every direction in this game, e.g., left, right, up, and down; after taking the food, the length of the snake increases. Food of the Snake will be generated at a given interval of time.

What do you mean by C++?

C++ is an **object-oriented programming language**. It is also called c++ with classes. It is a cross-platform language that can be used to create high-level applications. It gives programmers a high level of control over system resources and memory.

What do you mean by Computer Graphics?

The term computer graphics is the information displayed on a visual display unit or a computer printout in diagrams, graphics, pictures, and symbols.

Graphics primitives in C++

A graphics primitive is an essential non divisible graphical element for input or output within a computer graphics system. We need a header file called <graphics.h> to draw and create any graphics on the screen. It can also be defined as how a computer displays the data pictorially and manipulates it. Apart from drawing figures of various shapes, all animations and multimedia mainly work in the graphics platform.

The following functions are used to create graphics in the snake game:

Initgraph():

To initialize the graphics function, we must use the initgraph () function.

Syntax:

```
void Intergraph(int *graph driver, int *graph mode, char *path);
```

Initgraph function uses three-parameter:

- **gd:** It is used for the graphics driver.
- **gm:** It is used for the graphics mode.
- **path:** It specifies the path where the graphic file is located.

closegraph():

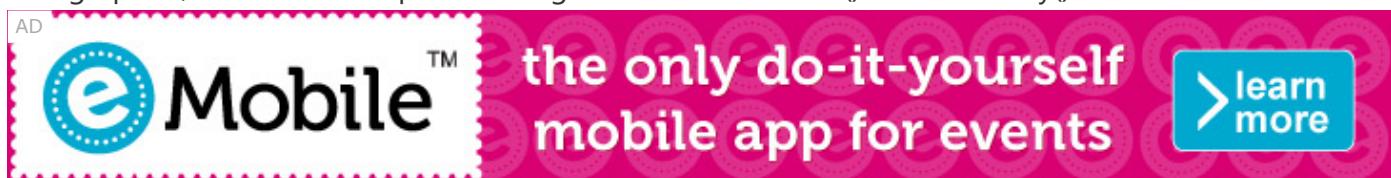
It is used to close the graphics function.

Syntax:

```
void closegraph();
```

Outputting text:

In C graphics, text can be outputted using the function outtext() and outtextxy().



outtext():

It is used to display the text at the current position.

Syntax:

```
void outtext(char *str);
```

outtextxy():



It is used to display the text at the specified position.

Syntax:

```
void outtextxy(int x, int y, char *str);
```

Let's take the example of the snake game in C++.

Example 1:

```
#include<iostream.h>
#include<conio.h>
```

```
#include<graphics.h>
#include<dos.h>
#include<stdlib.h>
#include<stdio.h>
#include<time.h>
#include<string.h>

class Snake
{
    int p1,p2,v1,v2,v3,e1,e2,prev,now,n,colr,dsp,cnt,dly,m;
    int stp, egGen;
    int xr, yr;
    void caught();
    public:
        long scr;
    int strtX,strtY,endX,endY;
    int pos[100][2];
    void show();
    void init();
    void egg();
    void transpose();
    void gnrtCond();
    void gnrtUnCond();
    void check();
    void checkEgg();
    void move();
    void chngDir();
    void sndEt();
    void sndCgt();
    int test();
    void score();
    Snake();
    Snake(Snake*);
    ~Snake();
};

Snake::Snake()
{
}

Snake::~Snake()
```

```
{  
}  
void Snake::checkEgg()  
{  
    if((e1 == p1) && (e2 == p2))  
    { sndEt();  
        egg();  
        dly--;  
        score();  
        n++;  
    }  
}  
void Snake::sndEt()  
{ nosound();  
    sound(2500);  
    delay(2);  
    nosound();  
}  
void Snake::sndCgt()  
{ nosound();  
    for(int x=1000;x>0;x--)  
    { sound(x);  
        delay(1);  
    }  
    nosound();  
}  
void Snake::score()  
{ char *p;  
    ltoa(scr,p,10);  
    settextstyle(8,0,1);  
    setcolor(0);  
    outtextxy(585,40,p);  
    if(egGen != 1){  
        scr = scr + dly / 10;  
    }  
    ltoa(scr,p,10);  
    setcolor(10);  
    outtextxy(585,40,p);  
}
```

```
}

void Snake::gnrtCond()
{
    if(n < 367)
    {
        if(now == 8 && (prev != 8 && prev != 2))
        {
            pos[0][0] = p1;
            pos[0][1] = p2 - dsp;
            prev = now;
        }

        if(now == 4 && (prev != 4 && prev != 1))
        {
            pos[0][0] = p1 + dsp;
            pos[0][1] = p2;
            prev = now;
        }

        if(now == 2 && (prev != 8 && prev != 2))
        {
            pos[0][0] = p1;
            pos[0][1] = p2 + dsp;
            prev = now;
        }

        if(now == 1 && (prev != 1 && prev != 4))
        {
            pos[0][0] = p1 - dsp;
            pos[0][1] = p2;
            prev = now;
        }
    }
}

void Snake::gnrtUnCond()
{
    if( prev == 8 )
    {
        pos[0][0] = p1;
        pos[0][1] = p2 - dsp;
    }

    if( prev == 4 )
    {
        pos[0][0] = p1 + dsp;
        pos[0][1] = p2;
    }

    if( prev == 2 )
    {
        pos[0][0] = p1;
        pos[0][1] = p2 + dsp;
    }
}
```

```
    }

if( prev == 1 )
{pos[0][0] = p1 - dsp;
 pos[0][1] = p2;
}

p1 = pos[0][0];
p2 = pos[0][1];
}

void Snake::check()
{
    if(p1 > endX)
{p1 = strtX;}
else if(p1 < strtX)
{ p1 = endX;}
if(p2 > endY)
{ p2 = strtY;}
else if(p2 < strtY)
{ p2 = endY;}
pos[0][0] = p1;
pos[0][1] = p2;
for(int i = 1;i < n;i++)
{ if(p1 == pos[i][0] && p2 == pos[i][1])
{ caught();
break;
}
}
}

void Snake::show()
{
    int x = getcolor();
if(egGen != 1)
{
    setcolor(getbkcolor());
    setfillstyle(1,getbkcolor());
    fillellipse(v1,v2,yr,yr);
}
else
    egGen = 0;
```

```
if(egGen == 2)
    egGen--;
    setcolor(colr);
    setfillstyle(1,9);
if(now == 8 || now == 2)
    fillellipse(pos[0][0],pos[0][1],xr,yr);
else if(now == 4 || now == 1)
    fillellipse(pos[0][0],pos[0][1],yr,xr);
    setcolor(x);
}

void Snake::transpose()
{ int i,j,x,y;
    p1 = pos[0][0];
    p2 = pos[0][1];
    if(!egGen){
        v1 = pos[n-1][0];
        v2 = pos[n-1][1];
    }
    else
        egGen = 0;
    for(i = n-1;i >= 1;i--)
    {pos[i][0] = pos[i-1][0];
     pos[i][1] = pos[i-1][1];
    }
}
void Snake::move()
{ int st = 0;
    do{
        if(!kbhit())
        { checkEgg();
            if(!st)
                show();
        }
        else
            st = 0;
        delay(dly/4);
        transpose();
        delay(dly/4);
        gnrtUnCond();
```

```
delay(dly/4);
check();
delay(dly/4);
}

else if(stp){
chngDir();
gnrtCond();
check();
show();
st = 1;
}
} while(stp);

}

void Snake::init()
{
time_t tm;
srand(time(&tm));
dsp = 20;
n = 5;
prev = 4;
for(int i = 4;i >= 0;i--)
{ pos[i][0] = 201 + (n - i - 1) * dsp;
pos[i][1] = 301;
}
strtX = 21;
strtY = 21;
endX = 481;
endY = 361;
colr = 14;
now = prev;
dsp = 20;
stp = 111;
cnt = -1;
scr = 0;
dly = 150;
xr = 3;
yr = 9;
egg();
egGen = 1;
```

```
score();
int x = getcolor();
setlinestyle(0,1,3);
setcolor(15);
rectangle(strtX-15,strtY-15,endX+15,endY+15);
rectangle(endX+25,strtY-15,getmaxx()-15,endY+15);
rectangle(strtX-15,endY+25,getmaxx()-15,getmaxy()-5);
line(endX+25,strtY+75,getmaxx()-15,strtY+75);
line(endX+25,strtY+200,getmaxx()-15,strtY+200);
line(endX+25,strtY+275,getmaxx()-15,strtY+275);
setlinestyle(0,1,1);
settextstyle(8,0,1);
setcolor(11);
outtextxy(514,40,"SCORE");
setcolor(14);
settextstyle(11,0,5);
outtextxy(524,110," CONTROLS ");
outtextxy(522,135,"p = PAUSE");
outtextxy(522,155,"g = RESUME");
outtextxy(522,175,"e = EXIT");
outtextxy(513,195,"ARROWS");
outtextxy(512,205," -MOVEMENT");
setcolor(14);
settextstyle(4,0,9);
outtextxy(getmaxx()-500,getmaxy()-110,"SNAKE");
settextstyle(8,0,1);
setcolor(x);
}

void Snake::caught()
{
    stp = 0;
    sndCgt();
    for(int i=0;i<=7;i++)
    { if(i%2)
        { setcolor(10);
            outtextxy(512,250,"GAME OVER");
            delay(900);
        }
    }
```

```
else
{setcolor(0);
outtextxy(512,250,"GAME OVER");
delay(500);
}
}

sleep(1);
}

void Snake::chngDir()
{ int clr;
fillsettingstype *p;
char x = getch();
if(x == 72)
now = 8;
else if(x == 77)
now = 4;
else if(x == 80)
now = 2;
else if(x == 75)
now = 1;
else if(x == 'e')
caught();
else if(x == 'p')
{ //int y = getcolor();
int twnkI = 1;
settextstyle(11,0,9);
while(1)
{if(kbhit())
{ int c = getch();
if(c == 'g')
{ clr = getcolor();
setcolor(0);
rectangle(endX+40,endY-10,getmaxx()-35,getmaxy()-160);
outtextxy(endX+60,endY-29,"PAUSE");
break;
}
}
else
```

```
{ if(twnkl%2)
{ clr = getcolor();
setcolor(10);
rectangle(endX+40,endY-10,getmaxx()-35,getmaxy()-160);
outtextxy(endX+60,endY-29,"PAUSE");
setcolor(clr);
delay(1000);
}

else
{
clr = getcolor();
setcolor(0);
rectangle(endX+40,endY-10,getmaxx()-35,getmaxy()-160);
outtextxy(endX+60,endY-29,"PAUSE");
delay(1000);
}

twnkl++;
}

settextstyle(8,0,1);

}

}

Snake::Snake(Snake *p)
{
*p=NULL;
}

void Snake::egg()
{ do
{ e1 = (rand() % 100) * dsp + strtX;
e2 = (rand() % 100) * dsp + strtY;
} while(test());
int x = getcolor();
setcolor(7);
setfillstyle(1,random(15)+1);
filellipse(e1,e2,xr+2,xr+2);
setcolor(x);
egGen = 2;
}
```

```

int Snake::test()
{
    for(int i=0;i<n;i++)
    { if(e1 == pos[i][0] && e2 == pos[i][1])
        break;
    if(v1 == e1 && v2 == e2)
        break;
    if((e1 >= endX+1) || (e2 >= endY+1))
        break;
    }

    if(i != n)
        return 1;
    else
        return 0;
}

void main()
{
    Snake snk;

    int gd=DETECT,gm,i,j,k,x,y;
    clrscr();
    initgraph(&gd,&gm,"C:\\Turboc3\\bgi");
    snk.init();
    snk.move();
    closegraph();
    restorecrtmode();
}

```

Explanation:

In the above example, we have implemented a basic **Snake Game**. In this, we have used the concept of classes and some computer graphics functions.

The basic functionalities of this game are given below.

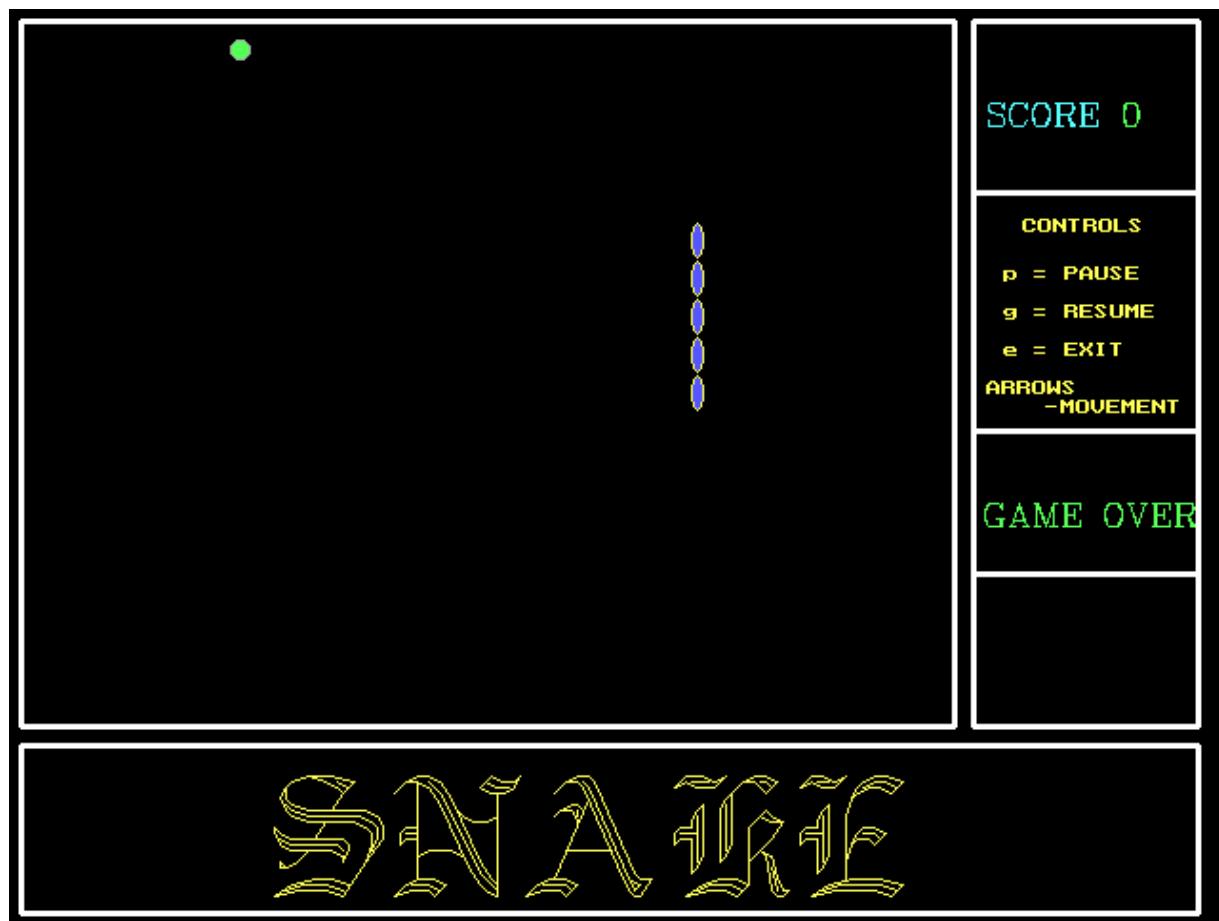
- The Snake is created with the help of a graphics function.
- The fruit of the Snake is generated by the rand() function of computer graphics.
- The Snake can be moved in any direction with the help of the keyboard (**Right, up, and down** keys).

- When the Snake eats a fruit, the score will increase by 14 points.
- **In this game, we can create basic controls:**
 - **p** control is used to pause the game. We can click a **p** character from the keyword to pause the game.
 - **g** control is used to resume the game. We can click a **g** character from the keyword to resume the game.
 - **e** control is used to exit from the game. We can click an **e** character from the keyword to exit from the game.



Output:

Following is the output of this example:



← Prev

Next →

Inline function in C++

One of the key features of C++ is the inline function. Therefore, let's first examine the utilization of inline functions and their intended application. If make a function is inline, then the compiler replaces the function calling location with the definition of the inline function at compile time.

Any changes made to an inline function will require the inline function to be recompiled again because the compiler would need to replace all the code with a new code; otherwise, it will execute the old functionality.

In brief, When the program executes a function call instruction, the CPU copies the function's arguments to the stack, caches the memory address of the next instruction and then hands control to the targeted function. The CPU then performs the function's code, saves the return value in a designated memory address or register, and hands control back to the function that is called the function. This may become overhead if the function's execution time is shorter than the time required to move from the calling function to the called function (callee). The overhead of the function call is typically negligible compared to the length of time required to run large or complicated functions. The time needed to call a small, frequently used function, however, is often far longer than the time necessary to run the function's code. Due to the fact that their execution time is less than switching time, tiny functions experience this overhead. To minimize the overhead of function calls, C++ offers inline functions. When a function is invoked, it expands in line and is known as an inline function. When an inline function is invoked, its entire body of code is added or replaced at the inline function call location. At compile time, the C++ compiler makes this substitution. Efficiency may be increased by inline function if it is tiny. To the compiler, inlining is merely a request; it is not a command. The compiler may reject the inlining request. The compiler may not implement inlining in situations like these:

1. If a function contains a loop. (for, while, do-while)
2. if a function has static variables.
3. Whether a function recurses.
4. If the return statement is absent from the function body and the return type of the function is not void.
5. Whether a function uses a goto or switch statement.

Syntax for an inline function:

```
inline return_type function_name(parameters)
{
    // function code?
```

{

Let's understand the difference between the normal function and the inline function.

Inside the **main()** method, when the function **fun1()** is called, the control is transferred to the definition of the called function. The addresses from where the function is called and the definition of the function are different. This control transfer takes a lot of time and increases the overhead.

When the inline function is encountered, then the definition of the function is copied to it. In this case, there is no control transfer which saves a lot of time and also decreases the overhead.

Let's understand through an example.

```
#include <iostream>
using namespace std;
inline int add(int a, int b)
{
    return(a+b);
}
int main()
{
    cout<<"Addition of 'a' and 'b' is:"<<add(2,3);
    return 0;
}
```

Once the compilation is done, the code would be like as shown as below:

```
#include<iostream>
using namespace std;
inline int add(int a, int b)
{
    return(a+b);
}
int main()
{
    cout<<"Addition of 'a' and 'b' is:"<<return(2+3);
    return 0;
```

{

Why do we need an inline function in C++?

The main use of the inline function in C++ is to save memory space. Whenever the function is called, then it takes a lot of time to execute the tasks, such as moving to the calling function. If the length of the function is small, then the substantial amount of execution time is spent in such overheads, and sometimes time taken required for moving to the calling function will be greater than the time taken required to execute that function.

The solution to this problem is to use macro definitions known as macros. The preprocessor macros are widely used in C, but the major drawback with the macros is that these are not normal functions which means the error checking process will not be done during the compilation.

C++ has provided one solution to this problem. In the case of function calling, the time for calling such small functions is huge, so to overcome such a problem, a new concept was introduced known as an inline function. When the function is encountered inside the main() method, it is expanded with its definition thus saving time.

We cannot provide the inlining to the functions in the following circumstances:

- If a function is recursive.
- If a function contains a loop like for, while, do-while loop.
- If a function contains static variables.
- If a function contains a switch or go to statement

When do we require an inline function?

An inline function can be used in the following scenarios:

- An inline function can be used when the performance is required.
- It can be used over the macros.
- We can use the inline function outside the class so that we can hide the internal implementation of the function.



Advantages of inline function

- In the inline function, we do not need to call a function, so it does not cause any overhead.
- It also saves the overhead of the return statement from a function.
- It does not require any stack on which we can push or pop the variables as it does not perform any function calling.
- An inline function is mainly beneficial for the embedded systems as it yields less code than a normal function.

Disadvantages of inline function

The following are the disadvantages of an inline function:



- The variables that are created inside the inline function will consume additional registers. If the variables increase, then the use of registers also increases, which may increase the overhead on register variable resource utilization. It means that when the function call is replaced with an inline function body, then the number of variables also increases, leading to an increase in the number of registers. This causes an overhead on resource utilization.
- If we use many inline functions, then the binary executable file also becomes large.
- The use of so many inline functions can reduce the instruction cache hit rate, reducing the speed of instruction fetch from the cache memory to that of the primary memory.
- It also increases the compile-time overhead because whenever the changes are made inside the inline function, then the code needs to be recompiled again to reflect the changes; otherwise, it will execute the old functionality.
- Sometimes inline functions are not useful for many embedded systems because, in some cases, the size of the embedded is considered more important than the speed.
- It can also cause thrashing due to the increase in the size of the binary executable file. If the thrashing occurs in the memory, then it leads to the degradation in the performance of the computer.

What is wrong with the macro?

Readers who are familiar with C are aware that it uses macros. All direct macro calls within the macro code are replaced by the preprocessor. It is advised to always use inline functions rather than macros. The inventor of C++, Dr. Bjarne Stroustrup, claims that macros are rarely required in C++.

and are error-prone. The use of macros in C++ is not without issues. Private class members are inaccessible to macro. Although macros resemble function calls, they do not truly do so.

Example

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

{
    int m;
public:
#define MAC(S::m) // error
};
```

The C++ compiler verifies that the necessary conversions are made and that the inline functions' parameter types are valid. A preprocessor macro cannot accomplish this. Additionally, the preprocessor manages macros, while the C++ compiler manages inline functions. Keep in mind that while it is true that all functions specified inside a class are implicitly inlined and that the C++ compiler will execute inline calls to these functions, inlining cannot be performed if the function is virtual. The cause is that a virtual function's call is resolved at runtime rather than at compile time. If the compiler doesn't know which function will be called, how can it conduct inlining during compilation when virtual means waiting till runtime? Another thing to keep in mind is that moving a function intoline will only be beneficial if the time required to call the function is longer than the time required to execute the function body. As an illustration, consider the following:

Example

```
inline void show()
{
    cout << "value of S = " << S << endl;
}
```

Executing the aforementioned function takes a while. The basic rule is that a function that performs input output (I/O) operations shouldn't be defined as inline because it takes a lot of time. Since the I/O statement would take far longer than a function call would, technically, inlining the show() function is only of limited utility.

If the function is not expanded inline, depending on the compiler you are using, a warning may be displayed. Inline functions cannot be used in programming languages like Java or C#.

But because final methods cannot be overridden by subclasses and calls to final methods are handled at compile time, the Java compiler can conduct inlining when the small final process is called. By inlining short function calls, the JIT compiler can further optimize C# code (like replacing body of a small function when it is called in a loop).

[← Prev](#)[Next →](#)

AD

[For Videos Join Our YouTube Channel: Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share



Virtual function vs Pure virtual function in C++

Before understanding the differences between the virtual function and pure virtual function in C++, we should know about the virtual function and pure virtual function in C++.

What is virtual function?

Virtual function is a member function that is declared within the base class and can be redefined by the derived class.

Let's understand through an example.

```
#include <iostream>
using namespace std;
class base
{
public:
void show()
{
    std::cout << "Base class" << std::endl;
}
};

class derived1 : public base
{
public:
void show()
{
    std::cout << "Derived class 1" << std::endl;
}
};

class derived2 : public base
{
public:
void show()
{
    std::cout << "Derived class 2" << std::endl;
}
};
```

```

};

int main()
{
    base *b;
    derived1 d1;
    derived2 d2;
    b=&d1;
    b->show();
    b=&d2;
    b->show();
    return 0;
}

```

In the above code, we have not used the virtual method. We have created a base class that contains the show() function. The two classes are also created named as '**derived1**' and '**derived2**' that are inheriting the properties of the base class. Both 'derived1' and 'derived2' classes have redefined the show() function. Inside the main() method, pointer variable 'b' of class base is declared. The objects of classes derived1 and derived2 are d1 and d2 respectively. Although the 'b' contains the addresses of d1 and d2, but when calling the show() method; it always calls the show() method of the base class rather than calling the functions of the derived1 and derived2 class.

To overcome the above problem, we need to make the method as virtual in the base class. Here, virtual means that the method exists in appearance but not in reality. We can make the method as virtual by simply adding the virtual keyword preceding to the function. In the above program, we need to add the virtual keyword that precedes to the show() function in the base class shown as below:

```

virtual void show()
{
    std::cout << "Base class" << std::endl;
}

```

Once the above changes are made, the output would be:

Important points:

- It is a run-time polymorphism.
- Both the base class and the derived class have the same function name, and the base class is assigned with an address of the derived class object then also pointer will execute the base

class function.

- If the function is made virtual, then the compiler will determine which function is to execute at the run time on the basis of the assigned address to the pointer of the base class.

What is pure virtual function?

A pure virtual function is a virtual function that has no definition within the class. Let's understand the concept of pure virtual function through an example.

In the above pictorial representation, shape is the base class while rectangle, square and circle are the derived class. Since we are not providing any definition to the virtual function, so it will automatically be converted into a pure virtual function.

Characteristics of a pure virtual function

- A pure virtual function is a "do nothing" function. Here "do nothing" means that it just provides the template, and derived class implements the function.
- It can be considered as an empty function means that the pure virtual function does not have any definition relative to the base class.
- Programmers need to redefine the pure virtual function in the derived class as it has no definition in the base class.
- A class having pure virtual function cannot be used to create direct objects of its own. It means that the class is containing any pure virtual function then we cannot create the object of that class. This type of class is known as an abstract class.



Syntax

There are two ways of creating a virtual function:

```
virtual void display() = 0;
```

or

```
virtual void display() {}
```

Let's understand through an example.

```
#include <iostream>
using namespace std;
// Abstract class
class Shape
{
public:
    virtual float calculateArea() = 0; // pure virtual function.
};

class Square : public Shape
{
float a;
public:
    Square(float l)
    {
        a = l;
    }
    float calculateArea()
    {
        return a*a;
    }
};

class Circle : public Shape
{
float r;
public:
    Circle(float x)
    {
        r = x;
    }
    float calculateArea()
    {
        return 3.14*r*r;
    }
};

class Rectangle : public Shape
{
```

```
float l;
float b;

public:

Rectangle(float x, float y)
{
    l=x;
    b=y;
}

float calculateArea()
{
    return l*b;
}

};

int main()
{
```

```
Shape *shape;
Square s(3.4);
Rectangle r(5,6);
Circle c(7.8);
shape =&s;
int a1 =shape->calculateArea();
shape = &r;
int a2 = shape->calculateArea();
shape = &c;
int a3 = shape->calculateArea();
std::cout << "Area of the square is " <<a1<< std::endl;
std::cout << "Area of the rectangle is " <<a2<< std::endl;
std::cout << "Area of the circle is " <<a3<< std::endl;
return 0;
```

Differences between the virtual function and pure virtual function



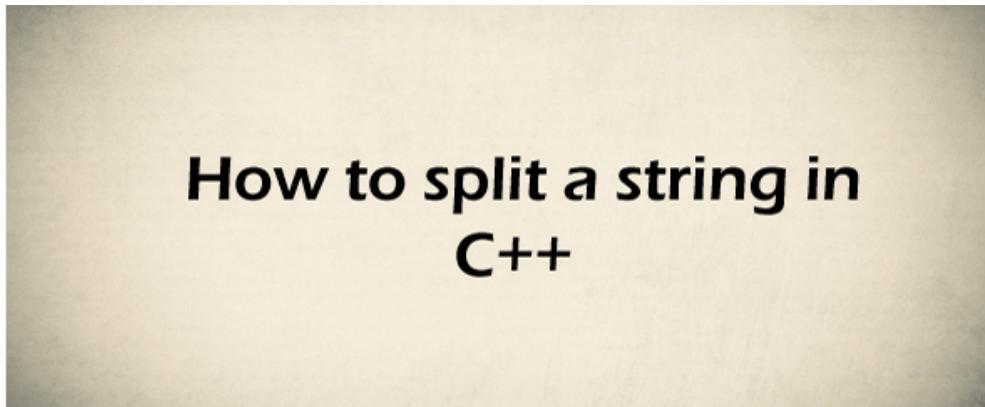
Virtual function	Pure virtual function
A virtual function is a member function in a base class that can be redefined in a derived class.	A pure virtual function is a member function in a base class whose declaration is provided in a base class and implemented in a derived class.
The classes which are containing virtual functions are not abstract classes.	The classes which are containing pure virtual function are the abstract classes.
In case of a virtual function, definition of a function is provided in the base class.	In case of a pure virtual function, definition of a function is not provided in the base class.
The base class that contains a virtual function can be instantiated.	The base class that contains a pure virtual function becomes an abstract class, and that cannot be instantiated.
If the derived class will not redefine the virtual function of the base class, then there will be no effect on the compilation.	If the derived class does not define the pure virtual function; it will not throw any error but the derived class becomes an abstract class.
All the derived classes may or may not redefine the virtual function.	All the derived classes must define the pure virtual function.

← Prev

Next →

How to Split strings in C++?

This topic will discuss how we can split given strings into a single word in the [C++ programming language](#). When we divide a group of words or string collections into single words, it is termed the **split** or division of the string. However, splitting strings is only possible with some delimiters like white space (), comma (,), a hyphen (-), etc., making the words an individual. Furthermore, there is no predefined split function to divide the collection of strings into an individual string. So, here we will learn the different methods to split strings into a single one in C++.



Different method to achieve the splitting of strings in C++

1. Use strtok() function to split strings
2. Use custom split() function to split strings
3. Use std::getline() function to split string
4. Use find() and substr() function to split string

Use strtok() function to split strings

strtok(): A strtok() function is used to split the original string into pieces or tokens based on the delimiter passed.

Syntax

```
char *ptr = strtok( str, delim)
```

In the above syntax, a strtok() has two parameters, the **str**, and the **delim**.

str: A str is an original string from which strtok() function split strings.

delim: It is a character that is used to split a string. For example, comma (,), space (), hyphen (-), etc.

Return: It returns a pointer that references the next character tokens. Initially, it points to the first token of the strings.

Note: A strtok() function modifies the original string and puts a NULL character ('\0') on the delimiter position on

each call of the `strtok()` function. In this way, it can easily track the status of the token.

Program to split strings using `strtok()` function

Let's consider an example to split string in C++ using `strtok()` function.

Program.cpp

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

int main()
{
    char str[100]; // declare the size of string
    cout << " Enter a string: " << endl;
    cin.getline(str, 100); // use getline() function to read a string from input stream

    char *ptr; // declare a ptr pointer
    ptr = strtok(str, ", "); // use strtok() function to separate string using comma () delimiter.
    cout << "\n Split string using strtok() function: " << endl;
    // use while loop to check ptr is not null
    while (ptr != NULL)
    {
        cout << ptr << endl; // print the string token
        ptr = strtok (NULL, ", ");
    }
    return 0;
}
```

Output

```
Enter a string:
Learn how to split a string in C++ using the strtok() function.

Split string using strtok() function:
Learn
how
to
split
a
string
in
```

C++
Using
the
strtok()
function.

Program to use custom split() function to split strings

Let's write a program to split sequences of strings in C++ using a custom split() function.

Program2.cpp

```
#include <iostream>
#include <string>
#define max 8 // define the max string
using namespace std;

string strings[max]; // define max string

// length of the string
int len(string str)
{
    int length = 0;
    for (int i = 0; str[i] != '\0'; i++)
    {
        length++;
    }
    return length;
}

// create custom split() function
void split (string str, char seperator)
{
    int currIndex = 0, i = 0;
    int startIndex = 0, endIndex = 0;
    while (i <= len(str))
    {
        if (str[i] == seperator || i == len(str))
        {
            endIndex = i;
            string subStr = "";
            subStr.append(str, startIndex, endIndex - startIndex);
            strings[currIndex] = subStr;
            currIndex++;
            startIndex = endIndex + 1;
        }
        i++;
    }
}
```

```
        strings[currIndex] = subStr;
        currIndex += 1;
        startIndex = endIndex + 1;
    }
    i++;
}
}

int main()
{
    string str = "Program to split strings using custom split function.";
    char separator = ' '; // space
    split(str, separator);
    cout << "The split string is: ";
    for (int i = 0; i < max; i++)
    {
        cout << "\n i : " << i << " " << strings[i];
    }
    return 0;
}
```

Output

```
The split string is:
i : 0 Program
i : 1 to
i : 2 split
i : 3 strings
i : 4 using
i : 5 custom
i : 6 split
i : 7 function.
```

Use std::getline() function to split string

A getline() function is a standard library function of C++ used to read the string from an input stream and put them into the vector string until delimiter characters are found. We can use **std::getline()** function by importing the <string> header file.

Syntax

```
getline(str, token, delim);
```

It has three parameters:

str: A str is a variable that stores original string.



token: It stores the string tokens extracted from original string.

delim: It is a character that are used to split the string. For example, comma (,), space (), hyphen (-), etc.

Program to use getline() function to split strings

Let's consider an example to split strings using the getline() function in C++.

Program3.cpp

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

int main()
{
    string S, T; // declare string variables

    getline(cin, S); // use getline() function to read a line of string and store into S variable.

    stringstream X(S); // X is an object of stringstream that references the S string

    // use while loop to check the getline() function condition
    while (getline(X, T, ' ')) {
        /* X represents to read the string from stringstream, T use for store the token string and,
         ' ' whitespace represents to split the string where whitespace is found. */

        cout << T << endl; // print split string
    }

    return 0;
}
```

Output



Welcome to the JavaTpoint and Learn C++ Programming Language.

Welcome
to
the
JavaTpoint
and
Learn
C++
Programming
Language.

Program to split the given string using the getline() function

Let's consider an example to split a given string in C++ using the getline() function.

Program4.cpp

```
#include <iostream>
#include <string>
#include <vector>
#include <sstream>

void split_str( std::string const &str, const char delim,
    std::vector <std::string> &out )
{
    // create a stream from the string
    std::stringstream s(str);

    std::string s2;
    while (std::getline (s, s2, delim) )
    {
        out.push_back(s2); // store the string in s2
    }
}

int main()
{
    std::string s2 = "Learn How to split a string in C++";
    const char delim = ' '; /* define the delimiter like space (' '), comma (,), hyphen (-), etc. */
}
```

```

std::cout << "Your given string is: " << s2;
std::vector<std::string> out; // store the string in vector
split_str(s2, delim, out); // call function to split the string

// use range based for loop
for (const auto &s2: out)
{
    std::cout << "\n" << s2;
}
return 0;
}

```

Output

```

Your given string is: Learn How to split a string in C++
Learn
How
to
split
a
string
in
C++

```

Use find() and substr() function to split strings

Let's write a program to use find() function and substr() function to split given strings in C++.

Program4.cpp

```

#include <iostream>
#include <string>
using namespace std;
int main()
{
    // given string with delimiter
    string given_str = "How_to_split_a_string_using_find()_and_substr()_function_in_C++";
    string delim = "_"; // delimiter

    cout << " Your string with delimiter is: " << given_str << endl;
    size_t pos = 0;
    string token1; // define a string variable

```

```
// use find() function to get the position of the delimiters
while (( pos = given_str.find (delim)) != std::string::npos)
{
    token1 = given_str.substr(0, pos); // store the substring
    cout << token1 << endl;
    given_str.erase(0, pos + delim.length()); /* erase() function store the current positon and move to next token. */
}
cout << given_str << endl; // it print last token of the string.
```

Output

```
Your string with delimiter is: How_to_split_a_string_using_find()_and_substr()_function_in_C++
How
to
split
a
string
using
find()
and
substr()
function
in
C++
```

In the above program, we use a **find()** function inside the loop to repeatedly find the occurrence of the delimiter in the given string and then split it into tokens when the delimiter occurs. And the **substr()** function stores the sub-string to be printed. On the other hand, an **erase()** function stores the current position of the string and moves to the next token, and this process continues until we have got all the split strings.

← Prev

Next →

Range-based for loop in C++

In this topic, we will discuss the range-based for loop in the C++ programming language. The C++ language introduced a new concept of the range-based for loop in C++11 and later versions, which is much better than the regular For loop. A range-based for loop does not require large coding to implement for loop iteration. It is a sequential iterator that iterated each element of the container over a range (from beginning to end).

Range Based For Loop in C++

Syntax

```
for (range_declaration : range_expression ) loop statement
```

1. **range_declaration:** It is used to declare a variable whose type is the same as the types of the collected elements represented by the range_expression or reference to that type.
2. **range_expression:** It defines an expression that represents the suitable sequence of elements.
3. **loop statement:** It defines the body of the range-based for loop that contains one or more statements to be repeatedly executed till the end of the range- expression.

Note: If we don't know the data type of the container elements, we can use the auto keyword that automatically identifies the data type of the range_expression.

Program to print each element of the array using-range based for loop

Let's consider an example to print the int and double array using the range-based for loop in C++.

program.cpp

```
#include <iostream>
using namespace std;
int main ()
{
    int arr1 [5] = { 10, 20, 30, 40, 50};
    double darr [5] = { 2.4, 4.5, 1.5, 3.5, 4.0 };

    // use range based for loop
    for ( const auto &var : arr1 )
    {
        cout << var << " ";
    }

    // use auto keyword to automatically specify the data type of darr container.
    for ( const auto &var : darr )
    {
        cout << var << " ";
    }

    return 0;
}
```

Output

```
10 20 30 40 50
2.4 4.5 1.5 3.5 4.0
```

Program to demonstrate the vector in range based for loop

Let's write a simple program to implement the vector in range based for loop.

Program2.cpp

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

int main()
{
    int x; // declare integer variable
```

```
// declare vector variable  
vector <int> vect = {5, 10 , 25, 20, 25};  
  
// display vector elements  
for ( int x : vect)  
{  
    cout << x << " ";  
}  
return 0;  
}
```

Output

```
5 10 25 20 25
```

Program to print the arrays using Range based for loop in C++ with reference

Let's consider an example to print the array elements using range based for loop in C++.

Program3.cpp

```
#include <iostream>  
#include <array>  
#include <cstdlib>  
using namespace std;  
  
int main(){  
array<int, 7> data = {1, 3, -2, 4, 6, 7, 9};  
cout << " Before updating the elements: " << endl;  
for (int x : data){  
    cout << x << " ";  
}  
// pass the references  
for (int &itemRef : data){  
    itemRef *= 3;  
}  
cout << endl << " After modification of the elements: " << endl;
```

```
for (int x : data){  
    cout << x << " ";  
}  
cout << endl;  
return 0;  
}
```

Output

Before updating the elements:

1 3 -2 4 6 7 9

After modification of the elements:

3 9 -6 12 18 21 27

Nested range-based for loop

When a loop is defined inside the body of another loop, the loop is called a nested for loop. Similarly, when we define a range in a loop inside another range-based loop, the technique is known as a nested range-based for loop.

Syntax:

```
for ( int x : range_expression) // outer loop  
{  
    for ( int y : range_expression) // inner loop  
    {  
        // statement to be executed  
    }  
    // statement to be executed  
}
```

In the above syntax, we define one range-based for loop inside another loop. Here we call inner and outer range-based for loop in C++.

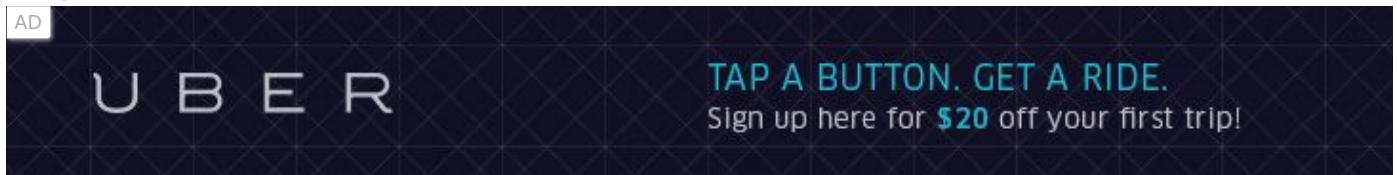
Program to print the nested range-based for loop in C++

Consider an example to demonstrate the nested range based for loop in C++ programming language.

Range.cpp

```
#include <iostream>
using namespace std;
int main ()
{
    int arr1[4] = { 0, 1, 2, 3 };
    int arr2[5] = { 1, 2, 3, 4, 5 };
    // use nested range based for loop
    for ( int x : arr1 )
    {
        // declare nested loop
        for ( int y : arr2 )
        {
            cout << " x = " << x << " and j = " << y << endl;
        }
    }
    return 0;
}
```

Output



```
x = 0 and j = 1
x = 0 and j = 2
x = 0 and j = 3
x = 0 and j = 4
x = 0 and j = 5
x = 1 and j = 1
x = 1 and j = 2
x = 1 and j = 3
x = 1 and j = 4
x = 1 and j = 5
x = 2 and j = 1
x = 2 and j = 2
x = 2 and j = 3
```

```
x = 2 and j = 4
x = 2 and j = 5
x = 3 and j = 1
x = 3 and j = 2
x = 3 and j = 3
x = 3 and j = 4
x = 3 and j = 5
```

What is the difference between traditional for loop and range-based for loop?

A **traditional for loop** is used to repeatedly execute the block of code till the specified condition is true. A traditional for loop has three parameters, initialization of the variable, specify the condition, and the last one is counter that is incremented by one if the condition remains true.

Syntax:

```
for ( variable_initialization; specify_condition; updated_counter)
{
    // statement to be executed;
}
```

Range-based loop

On the other hand, we have a new range-based for loop available in the C++ 11 and later version. It has two parameters, range declaration, and the range_ expression. It is also used to repeatedly execute the block of code over a range.

Syntax



```
for ( range_declaration : range_ expression )
{
    loop _statement;
    // statement to be executed;
}
```

The range_declaration is used to declare the type of variable related to the range_expression (container). The range_expression: It is just like a container that holds the same types of elements in a sequential manner. The loop_statement defines the statement which is executed inside for loop.

Advantages of the range-based for loop

1. It is easy to use, and its syntax is also simple.
2. A range-based for loop does not require the calculation of the number of elements in a containers
3. It recognizes the starting and ending elements of the containers.
4. We can easily modify the size and elements of the container.
5. It does not create any copy of the elements.
6. It is much faster than the traditional for loop.
7. It usually uses the auto keyword to recognize the data type of the container elements.

Disadvantage of the range-based for loop

1. It cannot traverse a part of a list.
2. It cannot be used to traverse in reverse order
3. It cannot be used in pointers.
4. It does not offer to index of the current elements.

← Prev

Next →

AD

Type Conversion in C++

In this topic, we will discuss the conversion of one data type into another in the C++ programming language. Type conversion is the process that converts the predefined data type of one variable into an appropriate data type. The main idea behind type conversion is to convert two different data type variables into a single data type to solve mathematical and logical expressions easily without any data loss.



For example, we are adding two numbers, where one variable is of int type and another of float type; we need to convert or typecast the int variable into a float to make them both float data types to add them.

Type conversion can be done in two ways in C++, one is **implicit type conversion**, and the second is **explicit type conversion**. Those conversions are done by the compiler itself, called the implicit type or automatic type conversion. The conversion, which is done by the user or requires user interferences called the explicit or user define type conversion. Let's discuss the implicit and explicit type conversion in C++.

Implicit Type Conversion

The implicit type conversion is the type of conversion done automatically by the compiler without any human effort. It means an implicit conversion automatically converts one data type into another type based on some predefined rules of the C++ compiler. Hence, it is also known as the **automatic type conversion**.

For example:

```
int x = 20;
```

```
short int y = 5;  
int z = x + y;
```

In the above example, there are two different data type variables, x, and y, where x is an int type, and the y is of short int data type. And the resultant variable z is also an integer type that stores x and y variables. But the C++ compiler automatically converts the lower rank data type (short int) value into higher type (int) before resulting the sum of two numbers. Thus, it avoids the data loss, overflow, or sign loss in implicit type conversion of C++.

Order of the typecast in implicit conversion

The following is the correct order of data types from lower rank to higher rank:

```
bool -> char -> short int -> int -> unsigned int -> long int -> unsigned long int -> long long int -> float -> double -> long double
```

Program to convert int to float type using implicit type conversion

Let's create a program to convert smaller rank data types into higher types using implicit type conversion.

Program1.cpp

```
#include <iostream>  
using namespace std;  
int main ()  
{  
    // assign the integer value  
    int num1 = 25;  
    // declare a float variable  
    float num2;  
    // convert int value into float variable using implicit conversion  
    num2 = num1;  
    cout << " The value of num1 is: " << num1 << endl;  
    cout << " The value of num2 is: " << num2 << endl;  
    return 0;  
}
```

Output

```
The value of num1 is: 25  
The value of num2 is: 25
```

Program to convert double to int data type using implicit type conversion

Let's create a program to convert the higher data type into lower type using implicit type conversion.

Program2.cpp

```
#include <iostream>  
using namespace std;  
int main()  
{  
    int num; // declare int type variable  
    double num2 = 15.25; // declare and assign the double variable  
  
    // use implicit type conversion to assign a double value to int variable  
    num = num2;  
    cout << " The value of the int variable is: " << num << endl;  
    cout << " The value of the double variable is: " << num2 << endl;  
    return 0;  
}
```

Output

```
The value of the int variable is: 15  
The value of the double variable is: 15.25
```

In the above program, we have declared num as an integer type and num2 as the double data type variable and then assigned num2 as 15.25. After this, we assign num2 value to num variable using the assignment operator. So, a C++ compiler automatically converts the double data value to the integer type before assigning it to the num variable and print the truncate value as 15.

Explicit type conversion

Conversions that require **user intervention** to change the data type of one variable to another, is called the **explicit type conversion**. In other words, an explicit conversion allows the programmer to manually changes or typecasts the data type from one variable to another type. Hence, it is also known as typecasting. Generally, we force the explicit type conversion to convert data from one type to another because it does not follow the implicit conversion rule.

The explicit type conversion is divided into two ways:

1. Explicit conversion using the cast operator
2. Explicit conversion using the assignment operator

Program to convert float value into int type using the cast operator

Cast operator: In C++ language, a cast operator is a unary operator who forcefully converts one type into another type.



Let's consider an example to convert the float data type into int type using the cast operator of the explicit conversion in C++ language.

Program3.cpp

```
#include <iostream>
using namespace std;
int main ()
{
    float f2 = 6.7;
    // use cast operator to convert data from one type to another
    int x = static_cast <int> (f2);
    cout << " The value of x is: " << x;
    return 0;
}
```

Output

```
The value of x is: 6
```

Program to convert one data type into another using the assignment operator

Let's consider an example to convert the data type of one variable into another using the assignment operator in the C++ program.

Program4.cpp



```
#include <iostream>
using namespace std;
int main ()
{
    // declare a float variable
    float num2;
    // initialize an int variable
    int num1 = 25;

    // convert data type from int to float
    num2 = (float) num1;
    cout << " The value of int num1 is: " << num1 << endl;
    cout << " The value of float num2 is: " << num2 << endl;
    return 0;
}
```

Output

```
The value of int num1 is: 25
The value of float num2 is: 25.0
```

LCM of two numbers in C++

The LCM stands for **Least Common Multiple**, which is used to get the smallest common multiple of two numbers (n_1 and n_2), and the common multiple should be divisible by the given numbers. A common multiple is a number that is common in both numbers. The representation of the LCM of two numbers, as LCM (a, b) or lcm (a, b).



For example, LCM of two positive numbers, as LCM (12, 24) is 24. Because both numbers 12 and 24 divides 24 without leaving any remainder. Similarly, the LCM of 3 and 4 is 12 because the smallest common multiple of both numbers is 12.

Algorithm of the LCM of two numbers

Step 1: Take two inputs from the user n_1 and n_2

Step 2: Store the smallest common multiple of n_1 and n_2 into the max variable.

Step 3: Validate whether the max variable is divisible by n_1 and n_2 , print the max as the LCM of two numbers.

Step 4: Otherwise, the max value is updated by 1 on every iteration, and jump to step 3 to check the divisibility of the max variable.

Step 5: Terminate the program.

Program to get LCM of two numbers using if statement and while loop

Program1.cpp

```
#include <iostream>
using namespace std;
int main()
{
    int n1, n2, max_num, flag = 1;
    cout << " Enter two numbers: \n";
    cin >> n1 >> n2;

    // use ternary operator to get the large number
    max_num = (n1 > n2) ? n1 : n2;

    while (flag)
    {
        // if statement checks max_num is completely divisible by n1 and n2.
        if(max_num % n1 == 0 && max_num % n2 == 0)
        {
            cout << " The LCM of " <<n1 << " and " << n2 << " is " << max_num;
            break;
        }
        ++max_num; // update by 1 on each iteration
    }
    return 0;
}
```

Output

```
Enter two numbers:
30
50
The LCM of 30 and 50 is 150
```

Program to get the LCM of two numbers using while loop

Program2.cpp

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

```
int main()
{
    // declare variables
    int num1, num2, lcm, gcd, temp;
    cout << " Enter the first number: ";
    cin >> num1;
    cout << " Enter the second number: ";
    cin >> num2;

    // assign num1 and num2 values to int a and b
    int a = num1;
    int b = num2;
    // use while loop to define the condition
    while (num2 != 0)
    {
        temp = num2;
        num2 = num1 % num2;
        num1 = temp;
    }

    // assign num1 to gcd variable
    gcd = num1;
    lcm = (a * b) / gcd;
    cout << "\n LCM of " << a << " and " << b << " = " << lcm;
    return 0;
}
```

Output

```
Enter the first number: 15
Enter the second number: 10

LCM of 15 and 10 = 30
```

Program to get the LCM of two numbers using GCD

Program3.cpp

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

// definition of getGCD() function
int getGCD( int n1, int n2)
{
    // here if statement checks the given number is not equal to 0.
    if ( n1 == 0 || n2 == 0)
        return 0;

    // if n1 equal to n2, return n1
    if (n1 == n2)
        return n1;

    // if n1 is greater than n2, execute the followed statement
    if ( n1 > n2)
        return getGCD (n1 - n2, n2);

    return getGCD (n1, n2 - n1);
}

// definition of getLCM() function
int getLCM (int n1, int n2)
{
    /* divide the multiplication of n1 and n2 by getGCD() function to return LCM. */
    return (n1 * n2) / getGCD (n1,n2);
}

int main()
{
    // declare local variables
    int n1, n2;
    cout << " Enter two positive numbers: ";
    cin >> n1 >> n2; // get numbers from user
```

```
// print the LCM(n1, n2)
cout << "\n LCM of " <<n1 << " and " << n2 << " is " << getLCM(n1, n2);
return 0;
}
```

Output

```
Enter two positive numbers: 50
60
```

```
LCM of 50 and 60 is 300
```

Program to get the LCM of two numbers using recursive function

Program 4.cpp

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

// define the getGCD() function and pass n1 and n2 parameter
int getGCD( int n1, int n2)
{
    if (n1 == 0)
    {
        return n2;
    }

    return getGCD(n2 % n1, n1);
}

// define the getLCM function to return the LCM
int getLCM( int n1, int n2)
{
    return (n1 / getGCD(n1, n2)) * n2;
}
```

```
int main()
{
    // declare local variable
    int num1, num2;
    cout << " Enter two numbers: " << endl;
    cin >> num1 >> num2;
    cout << " LCM of two numbers " << num1 << " and " << num2 << " is " << getLCM(num1, num2);
}
```

Output

```
Enter two numbers:
12
36
LCM of two numbers 12 and 36 is 36
```

Program to get the LCM of multiple array elements using function and while loop

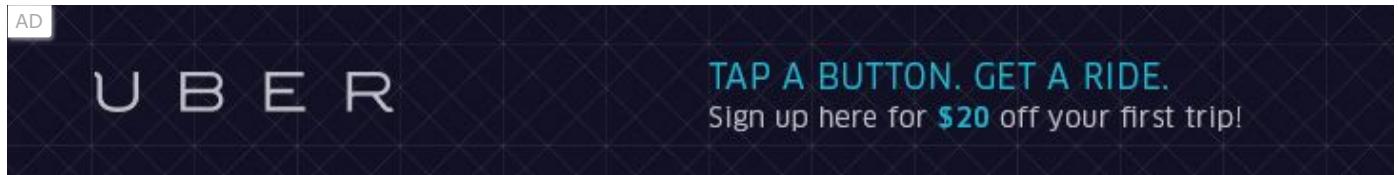
Program5.cpp

```
#include <iostream>
using namespace std;
int lcm_num (int n1, int n2)
{
    int max;
    max = (n1 > n2) ? n1 : n2;
    while (true)
    {
        if (max % n1 == 0 && max % n2 == 0)
            return max;
        max++;
    }
}

// definition of lcm_array() function
int lcm_array (int arr[], int num)
```

```
{  
    int i;  
    int lcm = lcm_num( arr[0], arr[1]);  
  
    // use for loop to get the lcm  
    for (i = 2; i < num; i++)  
    {  
        lcm = lcm_num(lcm , arr[i]);  
    }  
    return lcm;  
  
}  
  
int main()  
{  
    // declare integer array  
    int arr[] = {10, 5, 15, 30};  
    int num = sizeof(arr) / sizeof(arr[0]); // get the number of elements  
    cout << " LCM of multiple array elements is: " << lcm_array(arr, num);  
}
```

Output



LCM of multiple array elements is: 30

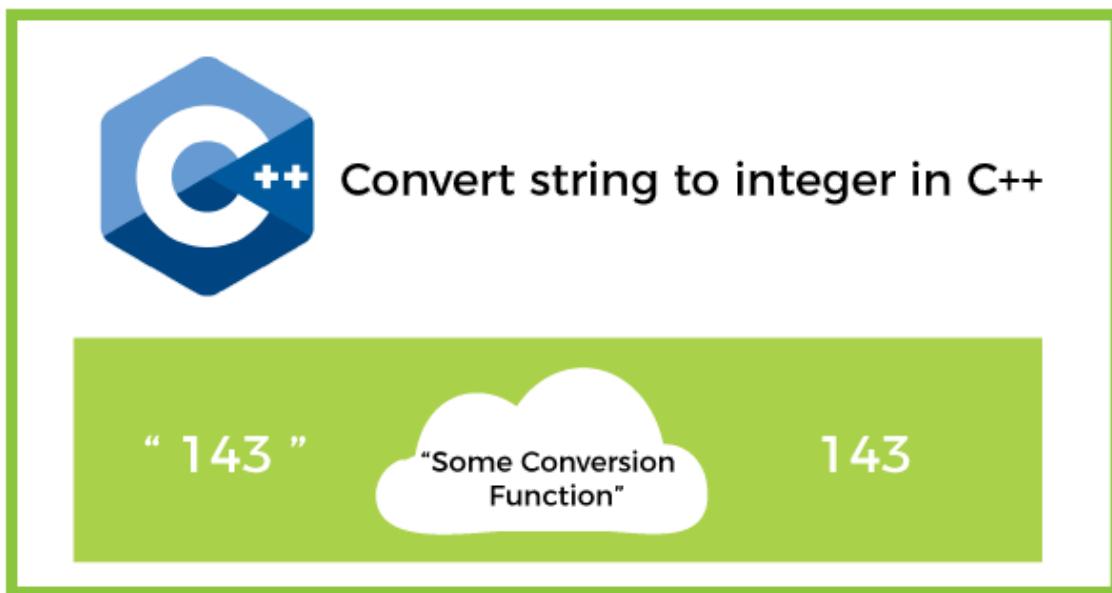
← Prev

Next →

Convert string to integer in C++

This section will discuss the different methods to convert the given string data into an integer using the C++ programming language. There are some situations or instances where we need to convert a certain data to another type, and one such situation is to convert string to int data in programming.

For example, we have a numeric string as "143", and we want to convert it to a numeric type. We need to use a function that converts a string to an integer and returns the numeric data as 143. Now, we will learn each method that helps convert string data into integers in the C++ programming language.



Different methods to convert the string data into integers in the C++ programming language.

1. Using the `stringstream` class
2. Using the `stoi()` function
3. Using the `atoi()` function
4. Using the `sscanf()` function

Using the `stringstream` class

The **stringstream** is a class used to convert a numeric string to an int type. The `stringstream` class declares a stream object to insert a string as a stream object and then extracts the converted integer data based on the streams. The `stringstream` class has "`<<`" and "`>>`" operators, which are used to fetch data from (`<<`) operator and insert data by passing stream to (`>>`) left operator.

Let's create a program to demonstrate the `stringstream` class for converting the string data into an integer in the C++ programming language.

Program1.cpp

```
#include <iostream>
#include <sstream> // use stringstream class
using namespace std;

int main()
{
    string str1 = "143"; // declare a string

    int intdata; // declare integer variable

    /* use stringstream class to declare a stream object to insert a string and then fetch as integer type */

    stringstream obj;
    obj << str1; // insert data into obj
    obj >> intdata; // fetch integer type data

    cout << "The string value is: " << str1 << endl;
    cout << "The representation of the string to integer type data is: " << intdata << endl;
    return 0;
}
```

Output

```
The string value is: 143
The representation of the string to integer type data is: 143
```

In the above program, we use the `stringstream` class to create an `obj` object, and it helps to convert string data into an integer. We then use the "`<<`" operator to insert string characters into the `obj` object, and then we use the "`>>`" operator to extract the converted string from `obj` into numeric data.

Using the sscanf() function

A sscanf() function converts a given string into a specified data type like an integer.

Syntax

```
sscanf ( str, %d, &intvar);
```

The sscanf() function has three arguments to specify the char string (str), data specifier (%d), and the integer variable (&intvar) to store the converted string.

Algorithm of the sscanf() function

1. The sscanf() function belongs to the stringstream class, so we need to import the class into our program.
2. Initialize a constant character string str.
3. Create an integer variable to hold the converted string to integer values.
4. Pass the string variable into the sscanf() function, and assign the sscanf() function to the integer variable to store the integer value generated by the function.
5. Print the integer values.

Let's consider an example to use the sscanf() function to convert the string into numeric number in C++.

Program2.cpp

```
#include <iostream>
#include <sstream> // use stringstream class
using namespace std;

int main ()
{
    // declare the character strings
    const char *str1 = "555";
    const char *str2 = "143";
    const char *str3 = "101";

    // declare the integer variables
    int numdata1;
```

```
int numdata2;
int numdata3;

/* use sscanf() function and to pass the character string str1,
and an integer variable to hold the string.*/
sscanf (str1, "%d", &numdata1);
cout << " The value of the character string is: " << str1;
cout << "\n The representation of string to int value of numdata1 is: " << numdata1 << endl;

sscanf (str2, "%d", &numdata2);
cout << "\n The value of the character string is: " << str2;
cout << "\n The representation of string to int value of numdata2 is: " << numdata2 << endl;

sscanf (str3, "%d", &numdata3);
cout << "\n The value of the character string is: " << str3;
cout << "\n The representation of string to int value of numdata3 is: " << numdata3 << endl;
return 0;
}
```

Output

```
The value of the character string is: 555
The representation of string to int value of numdata is: 555

The value of the character string is: 143
The representation of string to int value of numdata is: 143

The value of the character string is: 101
The representation of string to int value of numdata is: 101
```

Using the stoi() function

The `stoi()` function converts a string data to an integer type by passing the string as a parameter to return an integer value.

Syntax



```
stoi(str);
```

The `stoi()` function contains an `str` argument. The `str` string is passed inside the `stoi()` function to convert string data into an integer value.

Algorithm of the `stoi()` function

1. Initialize the string variable to store string values.
2. After that, it creates an integer type variable that stores the conversion of string into integer type data using the `stoi()` function.
3. Print the integer variable value.

Let's create a program to use the `stoi()` function to convert the string value into the integer type in the C++ programming language.

Program3.cpp

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

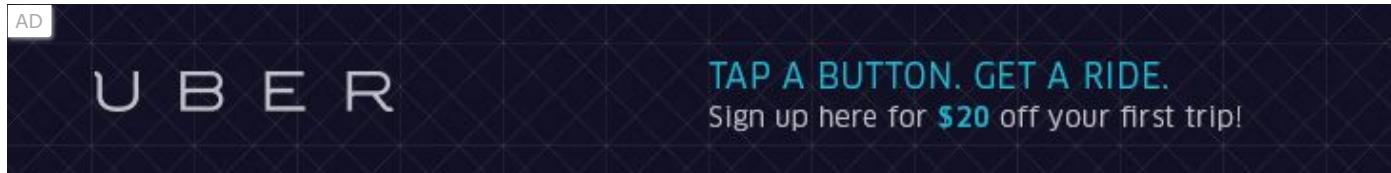
int main ()
{
    string strdata1 = "108";
    string strdata2 = "56.78";
    string strdata3 = "578 Welcome";

    // use stoi() function to pass string as arguments
    int intdata1 = stoi (strdata1);
    int intdata2 = stoi (strdata2);
    int intdata3 = stoi (strdata3);

    // print the converted values
    cout << " The conversion of string to an integer using stoi(\"" << strdata1 << "\") is " << intdata1
```

```
cout << " The conversion of string to an integer using stoi(\"" << strdata2 << "\\") is " << intdata  
cout << " The conversion of string to an integer using stoi(\"" << strdata3 << "\\") is " << intdata  
return 0;  
}
```

Output



```
The conversion of string to an integer using stoi("108") is 108  
The conversion of string to an integer using stoi("56.78") is 56  
The conversion of string to an integer using stoi("578 Welcome") is 578
```

Using the atoi() function

An atoi() function is used to convert the character string into an integer value. An atoi() function passes the character type string to return the integer data.

Syntax

```
atoi (const char *str);
```

Algorithm of the atoi() function

1. Initialize a pointer-type character array to store a string.
2. After that, it creates an integer type variable that stores the conversion of string into integer type data using the atoi() function.
3. Print the integer variable value.

Let's create a program to use the atoi() function to convert the string value into the integer type in the C++ programming language.

Program4.cpp

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

```
int main ()  
{  
    // declare a constant character array of string  
    const char *strdata1 = "256";  
    const char *strdata2 = "16.18";  
    const char *strdata3 = "1088 Good Bye";  
  
    // use atoi() function to pass character type array as arguments  
    int intdata1 = atoi (strdata1);  
    int intdata2 = atoi (strdata2);  
    int intdata3 = atoi (strdata3);  
  
    // print the converted values  
    cout << " The conversion of string to an integer value using atoi(\"256\") is " << i  
    cout << " The conversion of string to an integer value using atoi(\"16.18\") is " << i  
    cout << " The conversion of string to an integer value using atoi(\"1088 Good Bye\") is " << i  
    return 0;  
}
```

Output

```
The conversion of string to an integer value using atoi("256") is 256  
The conversion of string to an integer value using atoi("16.18") is 16  
The conversion of string to an integer value using atoi("1088 Good Bye") is 1088
```

← Prev

Next →

C++ STL (Standard Tag Library)

C++ STL Containers

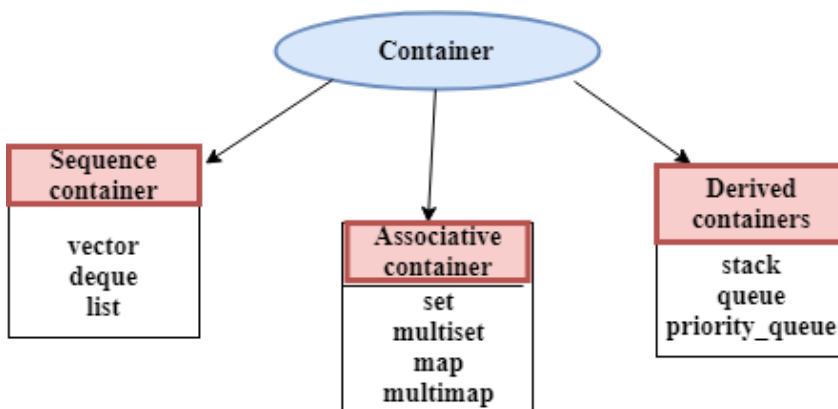
Containers can be described as the objects that hold the data of the same type. Containers are used to implement different data structures for example arrays, list, trees, etc.

Following are the containers that give the details of all the containers as well as the header file and the type of iterator associated with them :

Container	Description	Header file	iterator
vector	vector is a class that creates a dynamic array allowing insertions and deletions at the back.	<vector>	Random access
list	list is the sequence containers that allow the insertions and deletions from anywhere.	<list>	Bidirectional
deque	deque is the double ended queue that allows the insertion and deletion from both the ends.	<deque>	Random access
set	set is an associate container for storing unique sets.	<set>	Bidirectional
multiset	Multiset is an associate container for storing non-unique sets.	<set>	Bidirectional
map	Map is an associate container for storing unique key-value pairs, i.e. each key is associated with only one value(one to one mapping).	<map>	Bidirectional
multimap	multimap is an associate container for storing key-value pair, and each key can be associated with more than one value.	<map>	Bidirectional
stack	It follows last in first out(LIFO).	<stack>	No iterator
queue	It follows first in first out(FIFO).	<queue>	No iterator
Priority-queue	First element out is always the highest priority element.	<queue>	No iterator

Classification of containers :

- Sequence containers
- Associative containers
- Derived containers



Note : Each container class contains a set of functions that can be used to manipulate the contents.

ITERATOR

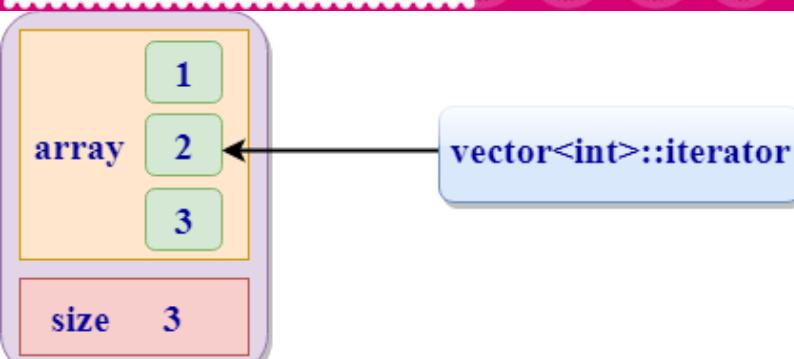
- Iterators are pointer-like entities used to access the individual elements in a container.
- Iterators are moved sequentially from one element to another element. This process is known as iterating through a container.

AD



eMobile™ the only do-it-yourself mobile app for events

learn more



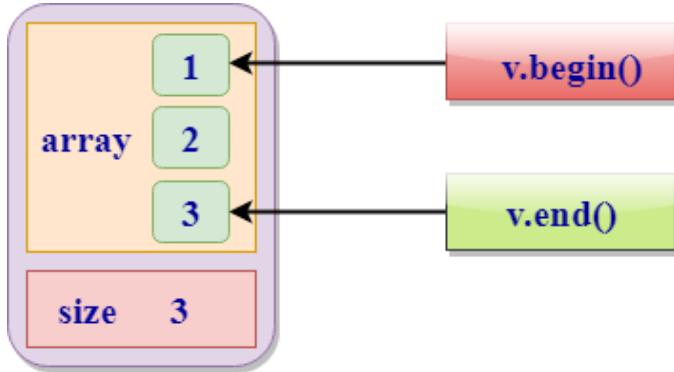
```

graph LR
    Iterator[vector<int>::iterator] --> Element2[2]
    subgraph Vector [ ]
        direction TB
        1[1]
        2[2]
        3[3]
    end
    Vector --- Size["size 3"]
  
```

- Iterator contains mainly two functions:

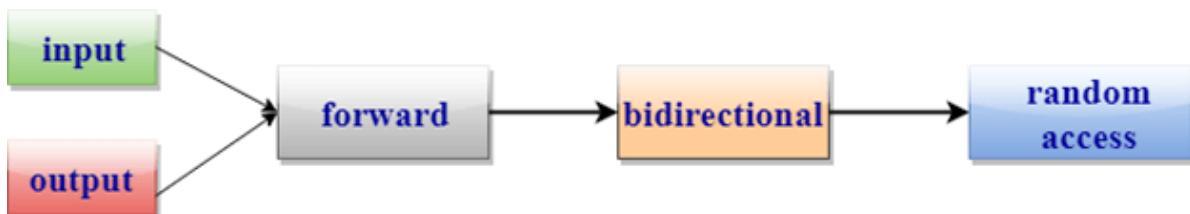
begin(): The member function begin() returns an iterator to the first element of the vector.

end(): The member function end() returns an iterator to the past-the-last element of a container.



Iterator Categories

Iterators are mainly divided into five categories:



1. Input iterator:

- An Input iterator is an iterator that allows the program to read the values from the container.
- Dereferencing the input iterator allows us to read a value from the container, but it does not alter the value.
- An Input iterator is a one way iterator.
- An Input iterator can be incremented, but it cannot be decremented.

2. Output iterator:

- An output iterator is similar to the input iterator, except that it allows the program to modify a value of the container, but it does not allow to read it.
- It is a one-way iterator.
- It is a write only iterator.

3. Forward iterator:

- Forward iterator uses the `++` operator to navigate through the container.
- Forward iterator goes through each element of a container and one element at a time.

4. Bidirectional iterator:

- A Bidirectional iterator is similar to the forward iterator, except that it also moves in the backward direction.
- It is a two way iterator.

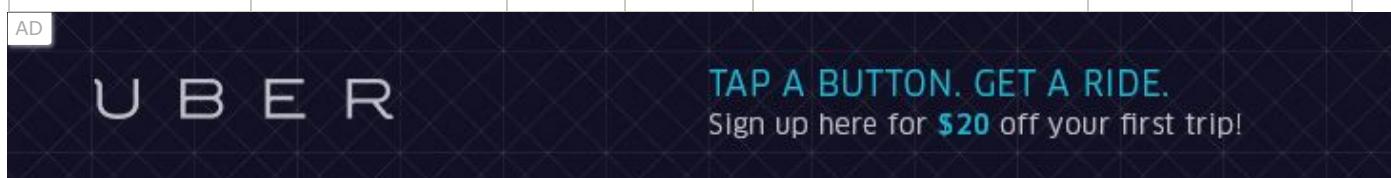
- o It can be incremented as well as decremented.

5. Random Access Iterator:

- o Random access iterator can be used to access the random element of a container.
- o Random access iterator has all the features of a bidirectional iterator, and it also has one more additional feature, i.e., pointer addition. By using the pointer addition operation, we can access the random element of a container.

Operations supported by iterators

iterator	Element access	Read	Write	Increment operation	Comparison
input	->	v = *p		++	==, !=
output			*p = v	++	
forward	->	v = *p	*p = v	++	==, !=
Bidirectional	->	v = *p	*p = v	++,--	==, !=
Random access	->,[]	v = *p	*p = v	++,--,+, -, +=, -=	==, !=, <, >, <=, >=



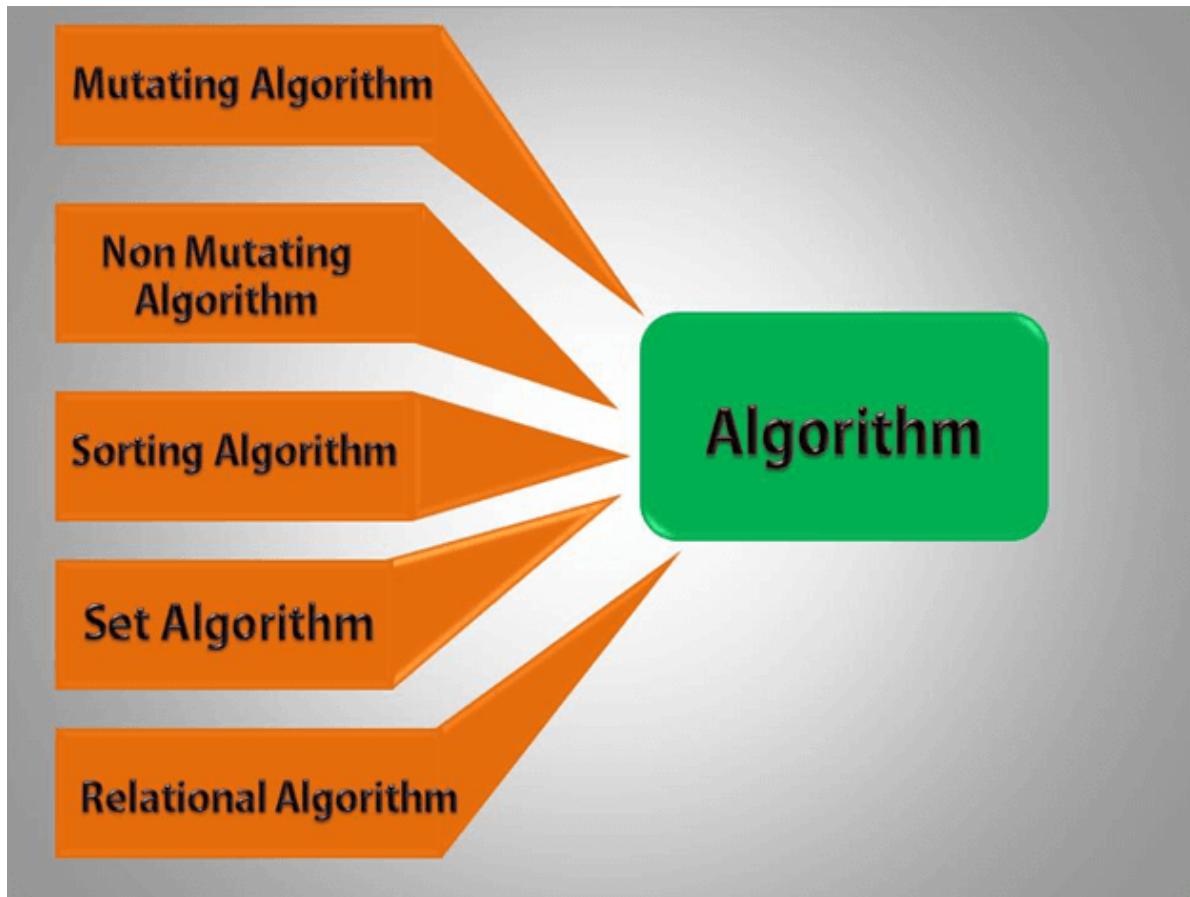
Algorithms

Algorithms are the functions used across a variety of containers for processing its contents.

Points to Remember:

- o Algorithms provide approx 60 algorithm functions to perform the complex operations.
- o Standard algorithms allow us to work with two different types of the container at the same time.
- o Algorithms are not the member functions of a container, but they are the standalone template functions.
- o Algorithms save a lot of time and effort.
- o If we want to access the STL algorithms, we must include the `<algorithm>` header file in our program.

STL algorithms can be categorized as:



- **Nonmutating algorithms:** Nonmutating algorithms are the algorithms that do not alter any value of a container object nor do they change the order of the elements in which they appear. These algorithms can be used for all the container objects, and they make use of the forward iterators.
- **Mutating algorithms:** Mutating algorithms are the algorithms that can be used to alter the value of a container. They can also be used to change the order of the elements in which they appear.
- **Sorting algorithms:** Sorting algorithms are the modifying algorithms used to sort the elements in a container.
- **Set algorithms:** Set algorithms are also known as sorted range algorithm. This algorithm is used to perform some function on a container that greatly improves the efficiency of a program.
- **Relational algorithms:** Relational algorithms are the algorithms used to work on the numerical data. They are mainly designed to perform the mathematical operations to all the elements in a container.

FUNCTION OBJECTS

A Function object is a function wrapped in a class so that it looks like an object. A function object extends the characteristics of a regular function by using the feature of an object oriented such as generic programming. Therefore, we can say that the function object is a smart pointer that has many advantages over the normal function.

Following are the advantages of function objects over a regular function:

- Function objects can have member functions as well as member attributes.
- Function objects can be initialized before their usage.
- Regular functions can have different types only when the signature differs. Function objects can have different types even when the signature is the same.
- Function objects are faster than the regular function.

A function object is also known as a '**functor**'. A function object is an object that contains atleast one definition of **operator()** function. It means that if we declare the object 'd' of a class in which **operator()** function is defined, we can use the object 'd' as a regular function.

Suppose 'd' is an object of a class, **operator()** function can be called as:

```
d();
```

which is same as:

```
d.operator() ( );
```

Let's see a simple example:

```
#include <iostream>
using namespace std;
class function_object
{
public:
    int operator()(int a, int b)
    {
        return a+b;
    }
};

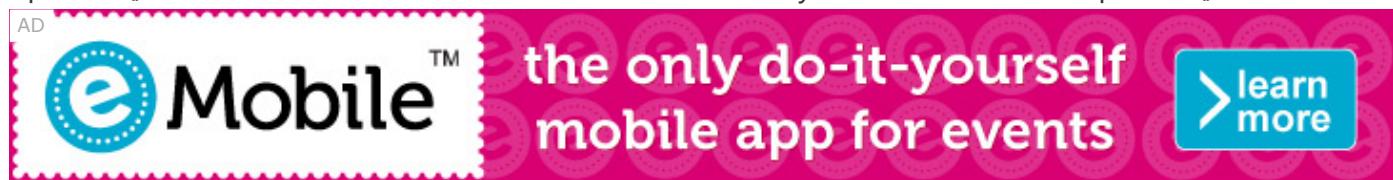
int main()
{
    function_object f;
    int result = f(5,5);
    cout<<"Addition of a and b is : "<<result;
```

```
    return 0;  
}
```

Output:

```
Addition of a and b is : 10
```

In the above example, 'f' is an object of a function_object class which contains the definition of operator() function. Therefore, 'f' can be used as an ordinary function to call the operator() function.



← Prev

Next →

AD

[For Videos Join Our Youtube Channel: Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

C++ Vector

A vector is a sequence container class that implements dynamic array, means size automatically changes when appending elements. A vector stores the elements in contiguous memory locations and allocates the memory as needed at run time.

Difference between vector and array

An array follows static approach, means its size cannot be changed during run time while vector implements dynamic array means it automatically resizes itself when appending elements.

Syntax

Consider a vector 'v1'. Syntax would be:

```
vector<object_type> v1;
```

Example

Let's see a simple example.

```
#include<iostream>
#include<vector>
using namespace std;
int main()
{
    vector<string> v1;
    v1.push_back("javaTpoint ");
    v1.push_back("tutorial");
    for(vector<string>::iterator itr=v1.begin();itr!=v1.end();++itr)
        cout<<*itr;
    return 0;
}
```

Output:

```
javaTpoint tutorial
```

In this example, vector class has been used to display the string.



C++ Vector Functions

Function	Description
at()	It provides a reference to an element.
back()	It gives a reference to the last element.
front()	It gives a reference to the first element.
swap()	It exchanges the elements between two vectors.
push_back()	It adds a new element at the end.
pop_back()	It removes a last element from the vector.
empty()	It determines whether the vector is empty or not.
insert()	It inserts new element at the specified position.
erase()	It deletes the specified element.
resize()	It modifies the size of the vector.
clear()	It removes all the elements from the vector.
size()	It determines a number of elements in the vector.
capacity()	It determines the current capacity of the vector.
assign()	It assigns new values to the vector.
operator=()	It assigns new values to the vector container.
operator[]()	It access a specified element.
end()	It refers to the past-lats-element in the vector.
emplace()	It inserts a new element just before the position pos.

<code>emplace_back()</code>	It inserts a new element at the end.
<code>rend()</code>	It points the element preceding the first element of the vector.
<code>rbegin()</code>	It points the last element of the vector.
<code>begin()</code>	It points the first element of the vector.
<code>max_size()</code>	It determines the maximum size that vector can hold.
<code>cend()</code>	It refers to the past-last-element in the vector.
<code>cbegin()</code>	It refers to the first element of the vector.
<code>crbegin()</code>	It refers to the last character of the vector.
<code>crend()</code>	It refers to the element preceding the first element of the vector.
<code>data()</code>	It writes the data of the vector into an array.
<code>shrink_to_fit()</code>	It reduces the capacity and makes it equal to the size of the vector.

[← Prev](#)[Next →](#)

AD

 For Videos Join Our YouTube Channel: [Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Initialize Vector in C++

A vector can store multiple data values like arrays, but they can only store object references and not primitive data types. They store an object's reference means that they point to the objects that contain the data, instead of storing them. Unlike an array, vectors need not be initialized with size. They have the flexibility to adjust according to the number of object references, which is possible because their storage is handled automatically by the container. The container will keep an internal copy of alloc, which is used to allocate storage for lifetime. Vectors can be located and traversed using iterators, so they are placed in contiguous storage. Vector has safety features also, which saves programs from crashing, unlike Array. We can give reserve space to vector, but not to arrays. An array is not a class, but a vector is a class. In vector, elements can be deleted, but not in arrays.

With the parent 'Collection class,' vector is sent in the form of a template class. The array is the lower level data structure with their specific properties. Vectors have functions & constructors; they are not index-based. They are the opposite of Arrays, which are index-based data structures. Here, the lowest address is provided to the first element, and the highest address is provided to the last element. Vector is used for insertion and deletion of an object, whereas Arrays used for frequent access of objects. Arrays are memory saving data structures, while Vector utilizes much more memory in exchange to manage storage and grow dynamically. Vector takes more time to access the elements, but this not the case with Arrays.

There are four ways of initializing a **vector in C++:**

- By entering the values one-by-one
- By using an overloaded constructor of the vector class
- By the help of arrays
- By using another initialized vector

By entering the values one-by-one -

All the elements in a vector can be inserted one-by-one using the vector class method 'push_back.'

Algorithm

Begin

Declare v of vector type.

Then we call push_back() function. This is done to insert values into vector v.

Then we print "Vector elements: \n".

" **for (int a: v)**

print all the elements of variable a."

Code -

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    vector<int> vec;
    vec.push_back(1);
    vec.push_back(2);
    vec.push_back(3);
    vec.push_back(4);
    vec.push_back(5);
    vec.push_back(6);
    vec.push_back(7);
    vec.push_back(8);
    vec.push_back(9);
    vec.push_back(101);
    for (int i = 0; i < vec.size(); i++)
    {
        cout << vec[i] << " ";
    }
    return 0;
}
```

Output

The screenshot shows a browser window with the URL [programiz.com/cpp-programming/online-compiler/](https://www.programiz.com/cpp-programming/online-compiler/). The main content area displays a C++ code editor with the file name 'main.cpp' containing the following code:

```

1 #include <iostream>
2 #include <vector>
3 using namespace std;
4 int main()
5 {
6     vector<int> vec;
7     vec.push_back(1);
8     vec.push_back(2);
9     vec.push_back(3);
10    vec.push_back(4);
11    vec.push_back(5);
12    vec.push_back(6);
13    vec.push_back(7);
14    vec.push_back(8);
15    vec.push_back(9);
16    vec.push_back(10);
17    for (int i = 0; i < vec.size(); i++)
18    {
19        cout << vec[i] << " ";
20    }
21    return 0;
22 }
23

```

Below the code editor is an 'Output' panel showing the numbers 1 through 10 separated by spaces. To the right of the output is a small video player window titled 'Meeting in "RCA-E31 Cloud Co..."' showing a person with the initials 'SB'. The bottom of the screen shows the Windows taskbar with various pinned icons.

Using an overloaded constructor -

When a vector has multiple elements with the same values, then we use this method.

By using an overloaded constructor of the vector class -

This method is mainly used when a vector is filled with multiple elements with the same value.

Algorithm

Begin

First, we initialize a variable say 's'.

Then we have to create a vector say 'v' with size's'.

Then we initialize vector v1.

Then initialize v2 by v1.

Then we print the elements.

End.

Code -

```

#include <iostream>
#include <vector>
using namespace std;
int main()

```

```
{  
int elements = 12;  
vector<int> vec(elements, 8);  
  
for (int i = 0; i < vec.size(); i++)  
{  
    cout << vec[i] << "\n";  
}  
  
return 0;  
}
```

Output

8 8 8 8 8 8 8 8 8 8 8 8 8 8

By the help of arrays -

We pass an array to the constructor of the vector class. The Array contains the elements which will fill the vector.

Algorithm -

Begin

First, we create a vector say v.

Then, we initialize the vector.

In the end, print the elements.

End

Code -

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    vector<int> vectr{9,8,7,6,5,4,3,2,1,0};
    for (int i = 0; i < vectr.size(); i++)
    {
        cout << vectr[i] << "\n";
    }
    return 0;
}
```

Output

```
9
8
7
6
5
4
3
2
1
0
```

Using another initialized vector -

Here, we have to pass the begin() and end() iterators of an initialized vector to a vector class constructor. Then we initialize a new vector and fill it with the old vector.

Algorithm -

Begin

First, we have to create a vector v1.

Then, we have to initialize vector v1 by an array.

Then we initialize vector v2 by v1.

We have to print the elements.

End.

Code -

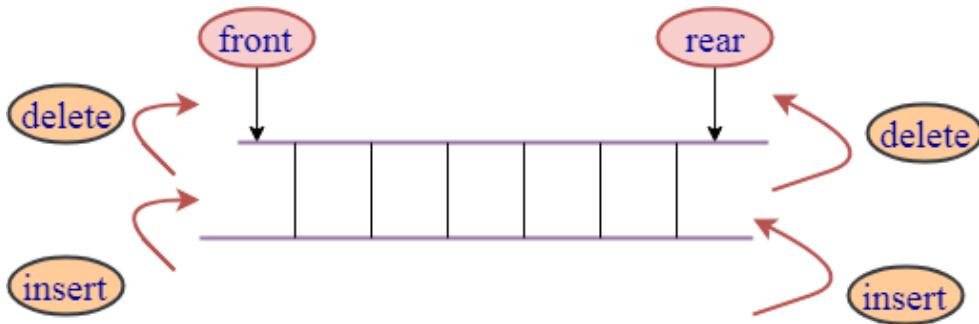
```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    vector<int> vec_1{1,2,3,4,5,6,7,8};
    vector<int> vec_2(vec_1.begin(), vec_1.end());
    for (int i = 0; i < vec_2.size(); i++)
    {
        cout << vec_2[i] << "\n";
    }
    return 0;
}
```

Output

```
1
2
3
4
5
6
7
8
```

C++ Deque

Deque stands for double ended queue. It generalizes the queue data structure i.e insertion and deletion can be performed from both the ends either front or back.



Syntax for creating a deque object:

```
deque<object_type> deque_name;
```

C++ Deque Functions

Method	Description
assign()	It assigns new content and replacing the old one.
emplace()	It adds a new element at a specified position.
emplace_back()	It adds a new element at the end.
emplace_front()	It adds a new element in the beginning of a deque.
insert()	It adds a new element just before the specified position.
push_back()	It adds a new element at the end of the container.
push_front()	It adds a new element at the beginning of the container.
pop_back()	It deletes the last element from the deque.
pop_front()	It deletes the first element from the deque.
swap()	It exchanges the contents of two deques.
clear()	It removes all the contents of the deque.

<code>empty()</code>	It checks whether the container is empty or not.
<code>erase()</code>	It removes the elements.
<code>max_size()</code>	It determines the maximum size of the deque.
<code>resize()</code>	It changes the size of the deque.
<code>shrink_to_fit()</code>	It reduces the memory to fit the size of the deque.
<code>size()</code>	It returns the number of elements.
<code>at()</code>	It access the element at position pos.
<code>operator[]()</code>	It access the element at position pos.
<code>operator=()</code>	It assigns new contents to the container.
<code>back()</code>	It access the last element.
<code>begin()</code>	It returns an iterator to the beginning of the deque.
<code>cbegin()</code>	It returns a constant iterator to the beginning of the deque.
<code>end()</code>	It returns an iterator to the end.
<code>cend()</code>	It returns a constant iterator to the end.
<code>rbegin()</code>	It returns a reverse iterator to the beginning.
<code>crbegin()</code>	It returns a constant reverse iterator to the beginning.
<code>rend()</code>	It returns a reverse iterator to the end.
<code>crend()</code>	It returns a constant reverse iterator to the end.
<code>front()</code>	It access the last element.

[← Prev](#)[Next →](#)

C++ List

- List is a contiguous container while vector is a non-contiguous container i.e list stores the elements on a contiguous memory and vector stores on a non-contiguous memory.
- Insertion and deletion in the middle of the vector is very costly as it takes lot of time in shifting all the elements. Linklist overcome this problem and it is implemented using list container.
- List supports a bidirectional and provides an efficient way for insertion and deletion operations.
- Traversal is slow in list as list elements are accessed sequentially while vector supports a random access.

Template for list

```
#include<iostream>
#include<list>
using namespace std;
int main()
{
    list<int> l;
```

It creates an empty list of integer type values.

List can also be initialised with the parameters.

```
#include<iostream>
#include<list>
using namespace std;
int main()
{
    list<int> l{1,2,3,4};
```

List can be initialised in two ways.

```
list<int> new_list{1,2,3,4};  
or  
list<int> new_list = {1,2,3,4};
```

C++ List Functions

Following are the member functions of the list:

Method	Description
insert()	It inserts the new element before the position pointed by the iterator.
push_back()	It adds a new element at the end of the vector.
push_front()	It adds a new element to the front.
pop_back()	It deletes the last element.
pop_front()	It deletes the first element.
empty()	It checks whether the list is empty or not.
size()	It finds the number of elements present in the list.
max_size()	It finds the maximum size of the list.
front()	It returns the first element of the list.
back()	It returns the last element of the list.
swap()	It swaps two list when the type of both the list are same.
reverse()	It reverses the elements of the list.
sort()	It sorts the elements of the list in an increasing order.
merge()	It merges the two sorted list.
splice()	It inserts a new list into the invoking list.
unique()	It removes all the duplicate elements from the list.
resize()	It changes the size of the list container.
assign()	It assigns a new element to the list container.

emplace()	It inserts a new element at a specified position.
emplace_back()	It inserts a new element at the end of the vector.
emplace_front()	It inserts a new element at the beginning of the list.

Next →

AD

Help Others, Please Share



C++ STL Set

Introduction to set

Sets are part of the **C++ STL (Standard Template Library)**. Sets are the associative containers that stores sorted key, in which each key is unique and it can be inserted or deleted but cannot be altered.

Syntax

```
template < class T,           // set::key_type/value_type
          class Compare = less<T>, // set::key_compare/value_compare
          class Alloc = allocator<T> // set::allocator_type
> class set;
```

Parameter

T: Type of element stored in the container set.

Compare: A comparison class that takes two arguments of the same type `bool` and returns a value. This argument is optional and the binary predicate `less<T>`, is the default value.

Alloc: Type of the allocator object which is used to define the storage allocation model.



Member Functions

Below is the list of all member functions of set:

Constructor/Destructor

Functions	Description
(constructor)	Construct set
(destructor)	Set destructor

operator=	Copy elements of the set to another set.
------------------	--

Iterators

Functions	Description
Begin	Returns an iterator pointing to the first element in the set.
cbegin	Returns a const iterator pointing to the first element in the set.
End	Returns an iterator pointing to the past-end.
Cend	Returns a constant iterator pointing to the past-end.
rbegin	Returns a reverse iterator pointing to the end.
Rend	Returns a reverse iterator pointing to the beginning.
crbegin	Returns a constant reverse iterator pointing to the end.
Crend	Returns a constant reverse iterator pointing to the beginning.

Capacity

Functions	Description
empty	Returns true if set is empty.
Size	Returns the number of elements in the set.
max_size	Returns the maximum size of the set.

Modifiers

Functions	Description
insert	Insert element in the set.
Erase	Erase elements from the set.
Swap	Exchange the content of the set.
Clear	Delete all the elements of the set.

<code>emplace</code>	Construct and insert the new elements into the set.
<code>emplace_hint</code>	Construct and insert new elements into the set by hint.

Observers

Functions	Description
<code>key_comp</code>	Return a copy of key comparison object.
<code>value_comp</code>	Return a copy of value comparison object.

Operations

Functions	Description
<code>Find</code>	Search for an element with given key.
<code>count</code>	Gets the number of elements matching with given key.
<code>lower_bound</code>	Returns an iterator to lower bound.
<code>upper_bound</code>	Returns an iterator to upper bound.
<code>equal_range</code>	Returns the range of elements matches with given key.

Allocator

Functions	Description
<code>get_allocator</code>	Returns an allocator object that is used to construct the set.

Non-Member Overloaded Functions

Functions	Description
<code>operator==</code>	Checks whether the two sets are equal or not.
<code>operator!=</code>	Checks whether the two sets are equal or not.
<code>operator<</code>	Checks whether the first set is less than other or not.

operator<=	Checks whether the first set is less than or equal to other or not.
operator>	Checks whether the first set is greater than other or not.
operator>=	Checks whether the first set is greater than equal to other or not.
swap()	Exchanges the element of two sets.

[Next →](#)

AD

 For Videos Join Our YouTube Channel: [Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share



Learn Latest Tutorials



C++ stack

In computer science we go for working on a large variety of programs. Each of them has their own domain and utility. Based on the purpose and environment of the program creation, we have a large number of data structures available to choose from. One of them is 'stack'. Before discussing about this data type let us take a look at its syntax.

Syntax

```
template<class T, class Container = deque<T>> class stack;
```

This data structure works on the LIFO technique, where LIFO stands for Last In First Out. The element which was first inserted will be extracted at the end and so on. There is an element called as 'top' which is the element at the upper most position. All the insertion and deletion operations are made at the top element itself in the stack.

Stacks in the application areas are implied as the container adaptors.

The containers should have a support for the following list of operations:

- empty
- size
- back
- push_back
- pop_back

Template Parameters

T: The argument specifies the type of the element which the container adaptor will be holding.

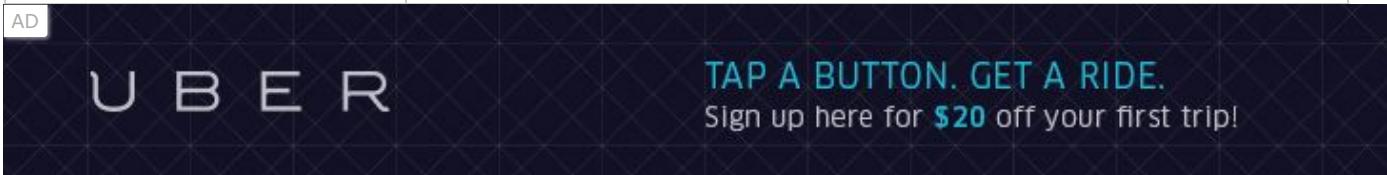
Container: The argument specifies an internal object of container where the elements of the stack are hold.

Member Types

Given below is a list of the stack member types with a short description of the same.

Member Types	Description
value_type	Element type is specified.

container_type	Underlying container type is specified.
size_type	It specifies the size range of the elements.



Functions

With the help of functions, an object or variable can be played with in the field of programming. Stacks provide a large number of functions that can be used or embedded in the programs. A list of the same is given below:

Function	Description
(constructor)	The function is used for the construction of a stack container.
<code>empty</code>	The function is used to test for the emptiness of a stack. If the stack is empty the function returns true else false.
<code>size</code>	The function returns the size of the stack container, which is a measure of the number of elements stored in the stack.
<code>top</code>	The function is used to access the top element of the stack. The element plays a very important role as all the insertion and deletion operations are performed at the top element.
<code>push</code>	The function is used for the insertion of a new element at the top of the stack.
<code>pop</code>	The function is used for the deletion of element, the element in the stack is deleted from the top.
<code>emplace</code>	The function is used for insertion of new elements in the stack above the current top element.
<code>swap</code>	The function is used for interchanging the contents of two containers in reference.
<code>relational operators</code>	The non member function specifies the relational operators that are needed for the stacks.

uses allocator<stack>	As the name suggests the non member function uses the allocator for the stacks.
--------------------------	---

Example: A simple program to show the use of basic stack functions.

```
#include <iostream>
#include <stack>
using namespace std;
void newstack(stack <int> ss)
{
    stack <int> sg = ss;
    while (!sg.empty())
    {
        cout << '\t' << sg.top();
        sg.pop();
    }
    cout << '\n';
}
int main ()
{
    stack <int> newst;
    newst.push(55);
    newst.push(44);
    newst.push(33);
    newst.push(22);
    newst.push(11);

    cout << "The stack newst is : ";
    newstack(newst);
    cout << "\n newst.size() : " << newst.size();
    cout << "\n newst.top() : " << newst.top();
    cout << "\n newst.pop() : ";
    newst.pop();
    newstack(newst);
    return 0;
}
```

Output:

```
The stack newst is :    11      22      33      44      55  
  
newst.size() : 5  
newst.top() : 11  
newst.pop() : 22      33      44      55
```

[← Prev](#)[Next →](#)

AD

 For Videos Join Our Youtube Channel: [Join Now](#)

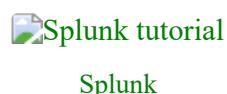
Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share



Learn Latest Tutorials



C++ queue

In computer science we go for working on a large variety of programs. Each of them has their own domain and utility. Based on the purpose and environment of the program creation, we have a large number of data structures available to choose from. One of them is 'queues'. Before discussing about this data type let us take a look at its syntax.

Syntax

```
template<class T, class Container = deque<T>> class queue;
```

This data structure works on the FIFO technique, where FIFO stands for First In First Out. The element which was first inserted will be extracted at the first and so on. There is an element called as 'front' which is the element at the front most position or say the first position, also there is an element called as 'rear' which is the element at the last position. In normal queues insertion of elements take at the rear end and the deletion is done from the front.

Queues in the application areas are implied as the container adaptors.

The containers should have a support for the following list of operations:

- empty
- size
- push_back
- pop_front
- front
- back

Template Parameters

T: The argument specifies the type of the element which the container adaptor will be holding.

Container: The argument specifies an internal object of container where the elements of the queues are held.

Member Types

Given below is a list of the queue member types with a short description of the same.

Member Types	Description
value_type	Element type is specified.
container_type	Underlying container type is specified.
size_type	It specifies the size range of the elements.
reference	It is a reference type of a container.
const_reference	It is a reference type of a constant container.



Functions

With the help of functions, an object or variable can be played with in the field of programming. Queues provide a large number of functions that can be used or embedded in the programs. A list of the same is given below:

Function	Description
(constructor)	The function is used for the construction of a queue container.
<code>empty</code>	The function is used to test for the emptiness of a queue. If the queue is empty the function returns true else false.
<code>size</code>	The function returns the size of the queue container, which is a measure of the number of elements stored in the queue.
<code>front</code>	The function is used to access the front element of the queue. The element plays a very important role as all the deletion operations are performed at the front element.
<code>back</code>	The function is used to access the rear element of the queue. The element plays a very important role as all the insertion operations are performed at the rear element.
<code>push</code>	The function is used for the insertion of a new element at the rear end of the queue.

pop	The function is used for the deletion of element; the element in the queue is deleted from the front end.
emplace	The function is used for insertion of new elements in the queue above the current rear element.
swap	The function is used for interchanging the contents of two containers in reference.
relational operators	The non member function specifies the relational operators that are needed for the queues.
uses allocator<queue>	As the name suggests the non member function uses the allocator for the queues.

Example: A simple program to show the use of basic queue functions.

```
#include <iostream>
#include <queue>
using namespace std;
void showsg(queue <int> sg)
{
    queue <int> ss = sg;
    while (!ss.empty())
    {
        cout << '\t' << ss.front();
        ss.pop();
    }
    cout << '\n';
}

int main()
{
    queue <int> fquiz;
    fquiz.push(10);
    fquiz.push(20);
    fquiz.push(30);

    cout << "The queue fquiz is : ";
}
```

```
showsg(fquiz);

cout << "\nfquiz.size() : " << fquiz.size();
cout << "\nfquiz.front() : " << fquiz.front();
cout << "\nfquiz.back() : " << fquiz.back();

cout << "\nfquiz.pop() : ";
fquiz.pop();
showsg(fquiz);

return 0;
}
```

Output:

```
The queue fquiz is :      10      20      30

fquiz.size() : 3
fquiz.front() : 10
fquiz.back() : 30
fquiz.pop() :      20      30
```

← Prev

Next →

AD

Priority Queue in C++

The priority queue in C++ is a derived container in STL that considers only the highest priority element. The queue follows the FIFO policy while priority queue pops the elements based on the priority, i.e., the highest priority element is popped first.

It is similar to the ordinary queue in certain aspects but differs in the following ways:

- o In a priority queue, every element in the queue is associated with some priority, but priority does not exist in a queue data structure.
- o The element with the highest priority in a priority queue will be removed first while queue follows the **FIFO(First-In-First-Out)** policy means the element which is inserted first will be deleted first.
- o If more than one element exists with the same priority, then the order of the element in a queue will be taken into consideration.

Note: The priority queue is the extended version of a normal queue except that the element with the highest priority will be removed first from the priority queue.

Syntax of Priority Queue

```
priority_queue<int> variable_name;
```

Let's understand the priority queue through a simple example.

Operation	Priority Queue	Return Value
Push(1)	1	
Push(4)	1 4	
Push(2)	1 4 2	
Pop()	1 2	4
Push(3)	1 2 3	
Pop()	1 4 2	3

In the above illustration, we have inserted the elements by using a push() function, and the insert operation is identical to the normal queue. But when we delete the element from the queue by using a pop() function, then the element with the highest priority will be deleted first.

Member Function of Priority Queue

Function	Description
push()	It inserts a new element in a priority queue.
pop()	It removes the top element from the queue, which has the highest priority.
top()	This function is used to address the topmost element of a priority queue.
size()	It determines the size of a priority queue.
empty()	It verifies whether the queue is empty or not. Based on the verification, it returns the status.
swap()	It swaps the elements of a priority queue with another queue having the same type and size.
emplace()	It inserts a new element at the top of the priority queue.

Let's create a simple program of priority queue.

```
#include <iostream>
#include<queue>
using namespace std;
int main()
{
    priority_queue<int> p; // variable declaration.
    p.push(10); // inserting 10 in a queue, top=10
    p.push(30); // inserting 30 in a queue, top=30
    p.push(20); // inserting 20 in a queue, top=20
    cout<<"Number of elements available in 'p' :"<<p.size()<<endl;
    while(!p.empty())
    {
        std::cout << p.top() << std::endl;
        p.pop();
    }
    return 0;
}
```

In the above code, we have created a priority queue in which we insert three elements, i.e., 10, 30, 20. After inserting the elements, we display all the elements of a priority queue by using a while loop.

Output

```
Number of elements available in 'p' :3
30
20
10 zzzzz/
```

Let's see another example of a priority queue.

```
#include <iostream>
#include<queue>
using namespace std;
int main()
{
    priority_queue<int> p; // priority queue declaration
    priority_queue<int> q; // priority queue declaration
    p.push(1); // inserting element '1' in p.
    p.push(2); // inserting element '2' in p.
    p.push(3); // inserting element '3' in p.
    p.push(4); // inserting element '4' in p.
    q.push(5); // inserting element '5' in q.
    q.push(6); // inserting element '6' in q.
    q.push(7); // inserting element '7' in q.
    q.push(8); // inserting element '8' in q.
    p.swap(q);
    std::cout << "Elements of p are :" << std::endl;
    while(!p.empty())
    {
        std::cout << p.top() << std::endl;
        p.pop();
    }
    std::cout << "Elements of q are :" << std::endl;
```

```

while(!q.empty())
{
    std::cout << q.top() << std::endl;
    q.pop();
}
return 0;
}

```

In the above code, we have declared two priority queues, i.e., p and q. We inserted four elements in 'p' priority queue and four in 'q' priority queue. After inserting the elements, we swap the elements of 'p' queue with 'q' queue by using a swap() function.

Output

Elements of p are :

8
7
6
5

Elements of q are :

4
3
2
1

← Prev

Next →

AD

 [For Videos Join Our YouTube Channel: Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share



Learn Latest Tutorials

 Splunk tutorial Splunk	 SPSS tutorial SPSS	 Swagger tutorial	 T-SQL tutorial Transact-SQL	 Tumblr tutorial Tumblr	 React tutorial ReactJS
---	---	--	--	--	---

C++ map function

Maps are part of the C++ STL (Standard Template Library). Maps are the associative containers that store sorted key-value pair, in which each key is unique and it can be inserted or deleted but cannot be altered. Values associated with keys can be changed.

For example: A map of Employees where employee ID is the key and name is the value can be represented as:

Keys	Values
101	Nikita
102	Robin
103	Deep
104	John

Syntax

```
template < class Key,                                     // map::key_type
          class T,                                         // map::mapped_type
          class Compare = less<Key>,                      // map::key_compare
          class Alloc = allocator<pair<const Key,T>> // map::allocator_type
        > class map;
```

Parameter

key: The key data type to be stored in the map.

type: The data type of value to be stored in the map.

compare: A comparison class that takes two arguments of the same type bool and returns a value. This argument is optional and the binary predicate less<"key"> is the default value.

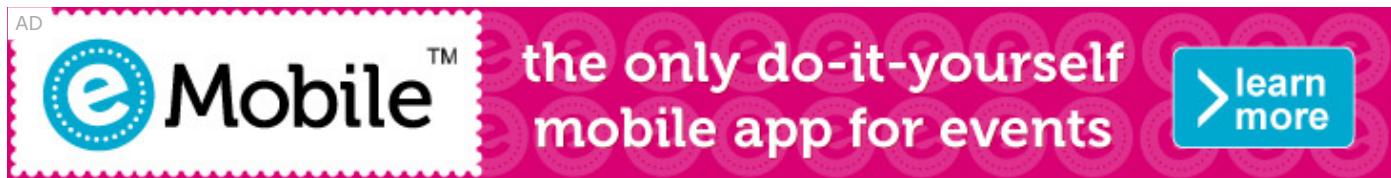
alloc: Type of the allocator object. This argument is optional and the default value is allocator .

Creating a map

Maps can easily be created using the following statement:

```
typedef pair<const Key, T> value_type;
```

The above form will use to create a map with key of type **Key type** and value of type **value type**. One important thing is that key of a map and corresponding values are always inserted as a pair, you cannot insert only key or just a value in a map.



Example 1

```
#include <string.h>
#include <iostream>
#include <map>
#include <utility>
using namespace std;
int main()
{
    map<int, string> Employees;
    // 1) Assignment using array index notation
    Employees[101] = "Nikita";
    Employees[105] = "John";
    Employees[103] = "Dolly";
    Employees[104] = "Deep";
    Employees[102] = "Aman";
    cout << "Employees[104] = " << Employees[104] << endl << endl;
    cout << "Map size: " << Employees.size() << endl;
    cout << endl << "Natural Order:" << endl;
    for( map<int,string>::iterator ii=Employees.begin(); ii!=Employees.end(); ++ii)
    {
        cout << (*ii).first << ":" << (*ii).second << endl;
    }
    cout << endl << "Reverse Order:" << endl;
    for( map<int,string>::reverse_iterator ii=Employees.rbegin(); ii!=Employees.rend(); ++ii)
    {
        cout << (*ii).first << ":" << (*ii).second << endl;
    }
}
```

```
    }  
}
```

Output:

```
Employees[104]=Deep
```

```
Map size: 5
```

```
Natural Order:
```

```
101: Nikita  
102: Aman  
103: Dolly  
104: Deep  
105: John
```

```
Reverse Order:
```

```
105: John  
104: Deep  
103: Dolly  
102: Aman  
101: Nikita
```

Member Functions

Below is the list of all member functions of map:

Constructor/Destructor

Functions	Description
constructors	Construct map
destructors	Map destructor
operator=	Copy elements of the map to another map.

Iterators

Functions	Description
<code>begin</code>	Returns an iterator pointing to the first element in the map.
<code>cbegin</code>	Returns a const iterator pointing to the first element in the map.
<code>end</code>	Returns an iterator pointing to the past-end.
<code>cend</code>	Returns a constant iterator pointing to the past-end.
<code>rbegin</code>	Returns a reverse iterator pointing to the end.
<code>rend</code>	Returns a reverse iterator pointing to the beginning.
<code>crbegin</code>	Returns a constant reverse iterator pointing to the end.
<code>crend</code>	Returns a constant reverse iterator pointing to the beginning.

Capacity

Functions	Description
<code>empty</code>	Returns true if map is empty.
<code>size</code>	Returns the number of elements in the map.
<code>max_size</code>	Returns the maximum size of the map.

Element Access

Functions	Description
<code>operator[]</code>	Retrieve the element with given key.
<code>at</code>	Retrieve the element with given key.

Modifiers

Functions	Description
<code>insert</code>	Insert element in the map.

<code>erase</code>	Erase elements from the map.
<code>swap</code>	Exchange the content of the map.
<code>clear</code>	Delete all the elements of the map.
<code>emplace</code>	Construct and insert the new elements into the map.
<code>emplace_hint</code>	Construct and insert new elements into the map by hint.

Observers

Functions	Description
<code>key_comp</code>	Return a copy of key comparison object.
<code>value_comp</code>	Return a copy of value comparison object.

Operations

Functions	Description
<code>find</code>	Search for an element with given key.
<code>count</code>	Gets the number of elements matching with given key.
<code>lower_bound</code>	Returns an iterator to lower bound.
<code>upper_bound</code>	Returns an iterator to upper bound.
<code>equal_range</code>	Returns the range of elements matches with given key.

Allocator

Functions	Description
<code>get_allocator</code>	Returns an allocator object that is used to construct the map.

Non-Member Overloaded Functions

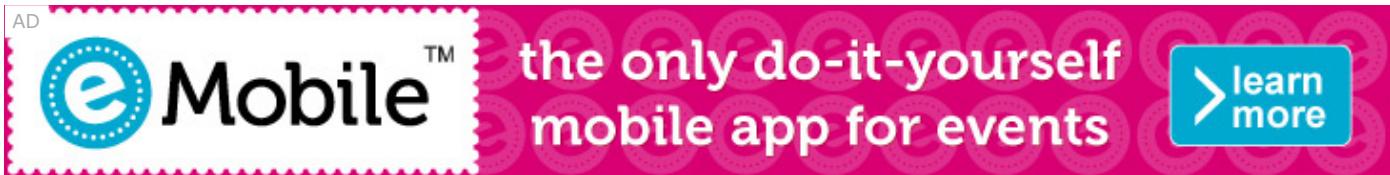
Functions	Description

operator==	Checks whether the two maps are equal or not.
operator!=	Checks whether the two maps are equal or not.
operator<	Checks whether the first map is less than other or not.
operator<=	Checks whether the first map is less than or equal to other or not.
operator>	Checks whether the first map is greater than other or not.
operator>=	Checks whether the first map is greater than equal to other or not.
swap()	Exchanges the element of two maps.

[← Prev](#)[Next →](#)

AD

Help Others, Please Share



C++ multimap

Multimaps are part of the **C++ STL (Standard Template Library)**. Multimaps are the associative containers like map that stores sorted key-value pair, but unlike maps which store only unique keys, **multimap can have duplicate keys**. By default it uses < operator to compare the keys.

For example: A multimap of Employees where employee age is the key and name is the value can be represented as:

Keys	Values
23	Nikita
28	Robin
25	Deep
25	Aman

Multimap employee has duplicate keys age.

Syntax

```
template < class Key,                                // multimap::key_type
          class T,                                     // multimap::mapped_type
          class Compare = less<Key>,                  // multimap::key_compare
          class Alloc = allocator<pair<const Key,T>> // multimap::allocator_type
        > class multimap;
```

Parameter

key: The key data type to be stored in the multimap.

type: The data type of value to be stored in the multimap.

compare: A comparison class that takes two arguments of the same type bool and returns a value. This argument is optional and the binary predicate less<"key"> is the default value.

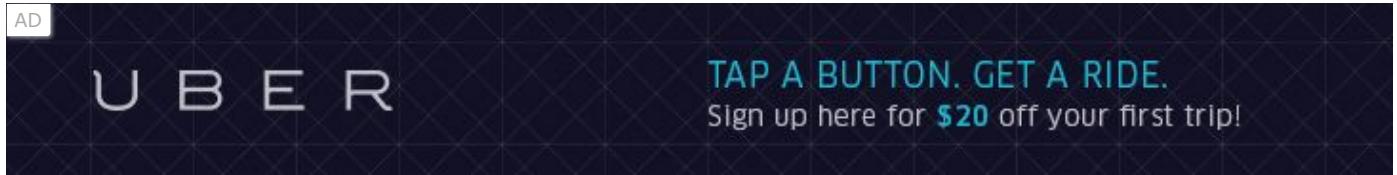
alloc: Type of the allocator object. This argument is optional and the default value is allocator .

Creating a multimap

Multimaps can easily be created using the following statement:

```
typedef pair<const Key, T> value_type;
```

The above form will use to create a multimap with key of type **Key_type** and value of type **value_type**. One important thing is that key of a multimap and corresponding values are always inserted as a pair, you cannot insert only key or just a value in a multimap.



Example

```
#include <iostream>
#include <map>
#include <string>

using namespace std;

int main()
{
    multimap<string, string> m = {
        {"India", "New Delhi"},
        {"India", "Hyderabad"},
        {"United Kingdom", "London"},
        {"United States", "Washington D.C"}
    };

    cout << "Size of map m: " << m.size() << endl;
    cout << "Elements in m: " << endl;

    for (multimap<string, string>::iterator it = m.begin(); it != m.end(); ++it)
    {
        cout << " [" << (*it).first << ", " << (*it).second << "]" << endl;
    }
}
```

```

return 0;
}

```

Output:

```

Size of map m: 4
Elements in m:
[India, New Delhi]
[India, Hyderabad]
[United Kingdom, London]
[United States, Washington D.C]

```

Member Functions

Below is the list of all member functions of multimap:

Constructor/Destructor

Functions	Description
constructor	Construct multimap
destructor	Multimap destructor
operator=	Copy elements of the multimap to another multimap.

Iterators

Functions	Description
begin	Returns an iterator pointing to the first element in the multimap.
cbegin	Returns a const_iterator pointing to the first element in the multimap.
end	Returns an iterator pointing to the past-end.
cend	Returns a constant iterator pointing to the past-end.
rbegin	Returns a reverse iterator pointing to the end.

<code>rend</code>	Returns a reverse iterator pointing to the beginning.
<code>crbegin</code>	Returns a constant reverse iterator pointing to the end.
<code>crend</code>	Returns a constant reverse iterator pointing to the beginning.

Capacity

Functions	Description
<code>empty</code>	Return true if multimap is empty.
<code>size</code>	Returns the number of elements in the multimap.
<code>max_size</code>	Returns the maximum size of the multimap.

Modifiers

Functions	Description
<code>insert</code>	Insert element in the multimap.
<code>erase</code>	Erase elements from the multimap.
<code>swap</code>	Exchange the content of the multimap.
<code>clear</code>	Delete all the elements of the multimap.
<code>emplace</code>	Construct and insert the new elements into the multimap.
<code>emplace_hint</code>	Construct and insert new elements into the multimap by hint.

Observers

Functions	Description
<code>key_comp</code>	Return a copy of key comparison object.
<code>value_comp</code>	Return a copy of value comparison object.

Operations

Functions	Description
find	Search for an element with given key.
count	Gets the number of elements matching with given key.
lower_bound	Returns an iterator to lower bound.
upper_bound	Returns an iterator to upper bound.
equal_range()	Returns the range of elements matches with given key.

Allocator

Functions	Description
get_allocator	Returns an allocator object that is used to construct the multimap.

Non-Member Overloaded Functions

Functions	Description
operator==	Checks whether the two multimaps are equal or not.
operator!=	Checks whether the two multimaps are equal or not.
operator<	Checks whether the first multimap is less than other or not.
operator<=	Checks whether the first multimap is less than or equal to other or not.
operator>	Checks whether the first multimap is greater than other or not.
operator>=	Checks whether the first multimap is greater than equal to other or not.
swap()	Exchanges the element of two multimaps.

← Prev

Next →

C++ Algorithm Functions

The library defines a large number of functions that are specially suited to be used on a large number of elements at a time or say a range. Now let's straightway take a look at these functions.

Non-modifying sequence operations:

Function	Description
<code>all_of</code>	The following function tests a condition to all the elements of the range.
<code>any_of</code>	The following function tests a condition to some or any of the elements of the range
<code>none_of</code>	The following function checks if none of the elements follow the condition or not.
<code>for_each</code>	The function applies an operation to all the elements of the range.
<code>find</code>	The function finds a value in the range.
<code>find_if</code>	The function finds for an element in the range.
<code>find_if_not</code>	The function finds an element in the range but in the opposite way as the above one.
<code>find_end</code>	The function is used to return the last element of the range.
<code>find_first_of</code>	The function finds for the element that satisfies a condition and occurs at the first.
<code>adjacent_find</code>	The function makes a search for finding the equal and adjacent elements in a range.
<code>count</code>	The function returns the count of a value in the range.
<code>count_if</code>	The function returns the count of values that satisfies a condition.
<code>mismatch</code>	The function returns the value in sequence which is the first mismatch.
<code>equal</code>	The function is used to check if the two ranges have all elements equal.
<code>is_permutation</code>	The function checks whether the range in reference is a permutation of some other range.

<code>search</code>	The function searches for the subsequence in a range.
<code>search_n</code>	The function searches the range for the occurrence of an element.

Modifying sequence operations

Function	Description
<code>copy</code>	The function copies the range of elements.
<code>copy_n</code>	The function copies n elements of the range
<code>copy_if</code>	The function copies the elements of the range if a certain condition is fulfilled.
<code>copy_backward</code>	The function copies the elements in a backward order
<code>move</code>	The function moves the ranges of elements.
<code>move_backward</code>	The function moves the range of elements in the backward order
<code>swap</code>	The function swaps the value of two objects.
<code>swap_ranges</code>	The function swaps the value of two ranges.
<code>iter_swap</code>	The function swaps the values of two iterators under reference.
<code>transform</code>	The function transforms all the values in a range.
<code>replace</code>	The function replaces the values in the range with a specific value.
<code>replace_if</code>	The function replaces the value of the range if a certain condition is fulfilled.
<code>replace_copy</code>	The function copies the range of values by replacing with an element.
<code>replace_copy_if</code>	The function copies the range of values by replacing with an element if a certain condition is fulfilled.
<code>fill</code>	The function fills the values in the range with a value.
<code>fill_n</code>	The function fills the values in the sequence.
<code>generate</code>	The function is used for the generation of values of the range.
<code>generate_n</code>	The function is used for the generation of values of the sequence.
<code>remove</code>	The function removes the values from the range.

<code>remove_if</code>	The function removes the values of the range if a condition is fulfilled.
<code>remove_copy</code>	The function copies the values of the range by removing them.
<code>remove_copy_if</code>	The function copies the values of the range by removing them if a condition is fulfilled.
<code>unique</code>	The function identifies the unique element of the range.
<code>unique_copy</code>	The function copies the unique elements of the range.
<code>reverse</code>	The function reverses the range.
<code>reverse_copy</code>	The function copies the range by reversing values.
<code>rotate</code>	The function rotates the elements of the range in left direction.
<code>rotate_copy</code>	The function copies the elements of the range which is rotated left.
<code>random_shuffle</code>	The function shuffles the range randomly.
<code>shuffle</code>	The function shuffles the range randomly with the help of a generator.

Partitions

Function	Description
<code>is_partitioned</code>	The function is used to deduce whether the range is partitioned or not.
<code>partition</code>	The function is used to partition the range.
<code>stable_partition</code>	The function partitions the range in two stable halves.
<code>partition_copy</code>	The function copies the range after partition.
<code>partition_point</code>	The function returns the partition point for a range.

AD



VIDEO STREAMING FOR THE AWAKENED MIND

GET STARTED NOW

Sorting

Function	Description

<code>sort</code>	The function sorts all the elements in a range.
<code>stable_sort</code>	The function sorts the elements in the range maintaining the relative equivalent order.
<code>partial_sort</code>	The function partially sorts the elements of the range.
<code>partial_sort_copy</code>	The function copies the elements of the range after sorting it.
<code>is_sorted</code>	The function checks whether the range is sorted or not.
<code>is_sorted_until</code>	The function checks till which element a range is sorted.
<code>nth_element</code>	The functions sorts the elements in the range.

Binary search

Function	Description
<code>lower_bound</code>	Returns the lower bound element of the range.
<code>upper_bound</code>	Returns the upper bound element of the range.
<code>equal_range</code>	The function returns the subrange for the equal elements.
<code>binary_search</code>	The function tests if the values in the range exists in a sorted sequence or not.

Merge

Function	Description
<code>merge</code>	The function merges two ranges that are in a sorted order.
<code>inplace_merge</code>	The function merges two consecutive ranges that are sorted.
<code>includes</code>	The function searches whether the sorted range includes another range or not.
<code>set_union</code>	The function returns the union of two ranges that is sorted.
<code>set_intersection</code>	The function returns the intersection of two ranges that is sorted.
<code>set_difference</code>	The function returns the difference of two ranges that is sorted.

set_symmetric_difference	The function returns the symmetric difference of two ranges that is sorted.
--------------------------	---

Heap

Function	Description
push_heap	The function pushes new elements in the heap.
pop_heap	The function pops new elements in the heap.
make_heap	The function is used for the creation of a heap.
sort_heap	The function sorts the heap.
is_heap	The function checks whether the range is a heap.
is_heap_until	The function checks till which position a range is a heap.

Min/Max

Function	Description
min	Returns the smallest element of the range.
max	Returns the largest element of the range.
minmax	Returns the smallest and largest element of the range.
min_element	Returns the smallest element of the range.
max_element	Returns the largest element of the range.
minmax_element	Returns the smallest and largest element of the range.

Other functions

Function	Description
lexicographical_comapre	The function performs the lexicographical less-than comparison.
next_permutation	The function is used for the transformation of range into the next permutation.

perv_permutation

The function is used for the transformation of range into the previous permutation.

Next →

AD

 [Youtube For Videos Join Our Youtube Channel: Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share



Learn Latest Tutorials

 [Splunk tutorial](#)

Splunk

 [SPSS tutorial](#)

SPSS

 [Swagger tutorial](#)

Swagger

 [T-SQL tutorial](#)

Transact-SQL

 [Tumblr tutorial](#)

 [React tutorial](#)

 [Regex tutorial](#)

C++ Iterators

Iterators are just like pointers used to access the container elements.

Important Points:

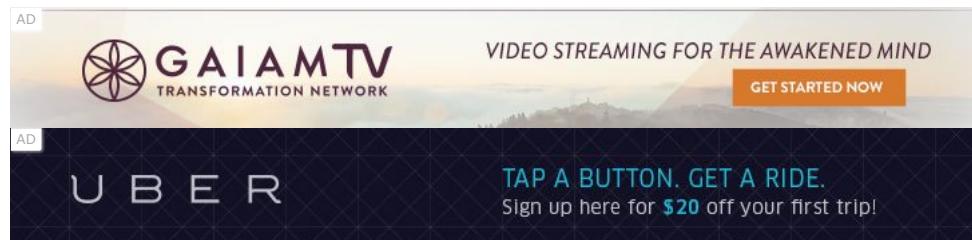
- Iterators are used to traverse from one element to another element, a process is known as **iterating through the container**.
- The main advantage of an iterator is to provide a common interface for all the containers type.
- Iterators make the **algorithm independent** of the type of the container used.
- Iterators provide a generic approach to navigate through the elements of a container.

Syntax

```
<ContainerType> :: iterator;
<ContainerType> :: const_iterator;
```

Operations Performed on the Iterators:

- **Operator (*)** : The '*' operator returns the element of the current position pointed by the iterator.
- **Operator (++)** : The '++' operator increments the iterator by one. Therefore, an iterator points to the next element of the container.
- **Operator (==) and Operator (!=)** : Both these operators determine whether the two iterators point to the same position or not.
- **Operator (=)** : The '=' operator assigns the iterator.

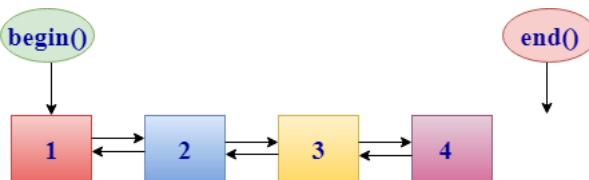


Difference b/w Iterators & Pointers

Iterators can be smart pointers which allow to iterate over the complex data structures. A Container provides its iterator type. Therefore, we can say that the iterators have the common interface with different container type.

The container classes provide two basic member functions that allow to iterate or move through the elements of a container:

- **begin()**: The begin() function returns an iterator pointing to the first element of the container.
- **end()**: The end() function returns an iterator pointing to the past-the-last element of the container.



Let's see a simple example:

```
#include <iostream>
#include<iterator>
#include<vector>
using namespace std;
int main()
{
```

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

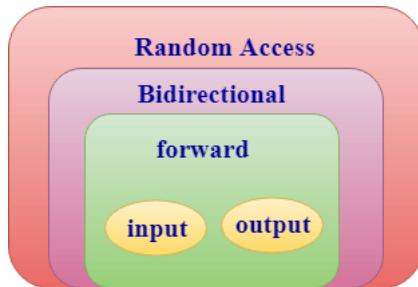
Output:

```
1 2 3 4 5
```

Iterator Categories

An iterator can be categorized in the following ways:

- Input Iterator
- Output Iterator
- Forward Iterator
- Bidirectional Iterator
- Random Access Iterator



Input Iterator: An input iterator is an iterator used to access the elements from the container, but it does not modify the value of a container.

Operators used for an input iterator are:

- Increment operator(++)
- Equal operator(==)
- Not equal operator(!=)
- Dereference operator(*)

Output Iterator: An output iterator is an iterator used to modify the value of a container, but it does not read the value from a container.

Therefore, we can say that an output iterator is a **write-only iterator**.

Operators used for an output iterator are:

- Increment operator(++)
- Assignment operator(=)

Forward Iterator: A forward iterator is an iterator used to read and write to a container. It is a multi-pass iterator.

Operators used for a Forward iterator are:

- Increment operator(++)
- Assignment operator(=)
- Equal operator(=)

- Not equal operator(\neq)

Bidirectional iterator: A bidirectional iterator is an iterator supports all the features of a forward iterator plus it adds one more feature, i.e., decrement operator($--$). We can move backward by decrementing an iterator.

Operators used for a Bidirectional iterator are:

- Increment operator($++$)
- Assignment operator($=$)
- Equal operator($=$)
- Not equal operator(\neq)
- Decrement operator($--$)

Random Access Iterator: A Random Access iterator is an iterator provides random access of an element at an arbitrary location. It has all the features of a bidirectional iterator plus it adds one more feature, i.e., pointer addition and pointer subtraction to provide random access to an element.

Providers Of Iterators

Iterator categories	Provider
Input iterator	istream
Output iterator	ostream
Forward iterator	
Bidirectional iterator	List, set, multiset, map, multimap
Random access iterator	Vector, deque, array

Iterators and their Characteristics

Iterator	Access method	Direction of movement	I/O capability
Input	Linear	Forward only	Read-only
Output	Linear	Forward only	Write-only
Forward	Linear	Forward only	Read/Write
Bidirectional	Linear	Forward & backward	Read/Write
Random	Random	Forward & backward	Read/Write

Disadvantages of iterator

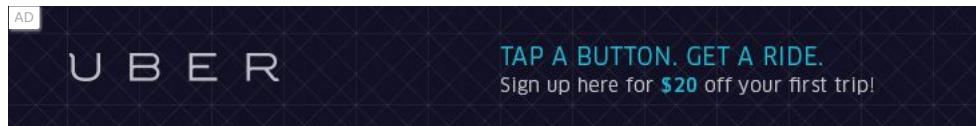
- If we want to move from one data structure to another at the same time, iterators won't work.
- If we want to update the structure which is being iterated, an iterator won't allow us to do because of the way it stores the position.
- If we want to backtrack while processing through a list, the iterator will not work in this case.

Advantages of iterator

Following are the advantages of an iterator:

- **Ease in programming:** It is convenient to use iterators rather than using a subscript operator[] to access the elements of a container. If we use subscript operator[] to access the elements, then we need to keep the track of the number of elements added at the runtime, but this would not happen in the case of an iterator.

Let's see a simple example:



```
#include <iostream>
#include<vector>
#include<iterator>
using namespace std;
int main()
{
    vector<int> v{1,2,3,4,5};
    vector<int>::iterator itr;
    for(int i=0;i<5;i++)
        // Traversal without using an iterator.
    {
        cout<<v[i]<<" ";
    }
    cout<<'\n';
    for(itr=v.begin();itr!=v.end();itr++)
        // Traversal by using an iterator.
    {
        cout<<*itr<<" ";
    }
    v.push_back(10);
    cout<<'\n';
    for(int i=0;i<6;i++)
    {
        cout<<v[i]<<" ";
    }
    cout<<'\n';
    for(itr=v.begin();itr!=v.end();itr++)
    {
        cout<<*itr<<" ";
    }
    return 0;
}
```

Output:

```
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5 10
1 2 3 4 5 10
```

In the above example, we observe that if we traverse the elements of a vector without using an iterator, then we need to keep track of the number of elements added in the container.

- **Code Reusability:** A code can be reused if we use iterators. In the above example, if we replace vector with the list, and then the subscript operator[] would not work to access the elements as the list does not support the random access. However, we use iterators to access the elements, then we can also access the list elements.
- **Dynamic Processing:** C++ iterators provide the facility to add or delete the data dynamically.

Let's see a simple example:

```
#include <iostream>
#include<vector>
```

```
#include<iterator>
using namespace std;
int main()
{
    vector<int> v{1,2,3,4,5}; // vector declaration
    vector<int>::iterator itr;
    v.insert(v.begin() + 1, 10);
    for(itr=v.begin();itr!=v.end();itr++)
    {
        cout<<*itr<<" ";
    }
    return 0;
}
```

Output:

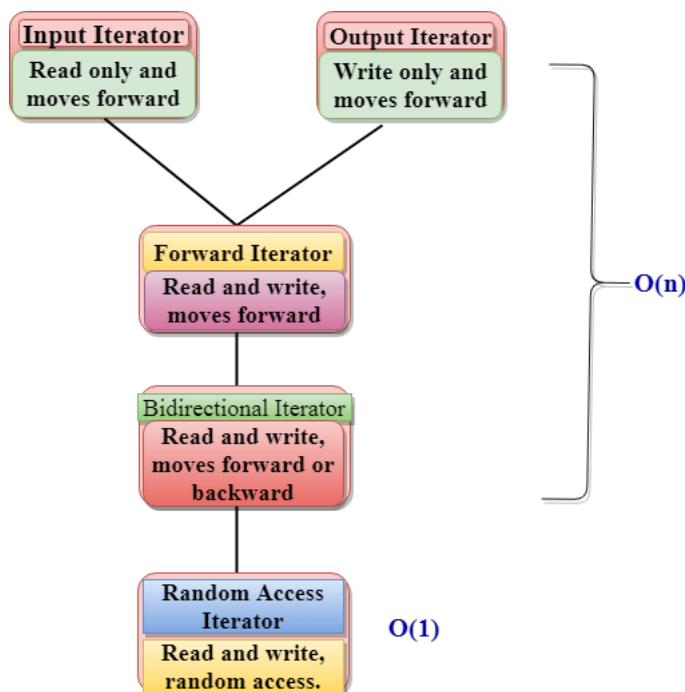
```
1 10 2 3 4 5
```

In the above example, we insert a new element at the second position by using `insert()` function and all other elements are shifted by one.



Difference b/w Random Access Iterator and Other Iterators

The most important difference between the Random access iterator and other iterators is that **random access iterator requires '1' step to access an element while other iterators require 'n' steps.**



C++ Bidirectional iterator

- A Bidirectional iterator supports all the features of a forward iterator, and it also supports the two **decrement operators** (prefix and postfix).
- Bidirectional iterators are the iterators used to access the elements in both the directions, i.e., **towards the end and towards the beginning**.
- A **random access iterator** is also a valid bidirectional iterator.
- Many containers implement the bidirectional iterator such as list, set, multiset, map, multimap.
- C++ provides two non-const iterators that move in both the directions are iterator and reverse iterator.
- C++ Bidirectional iterator has the same features like the forward iterator, with the only difference is that the bidirectional iterator can also be decremented.

Properties Of Bidirectional Iterator

Suppose x and y are the two iterators:

Property	Expressions
A Bidirectional iterator is a default-constructible, copy-assignable and destructible.	<code>A x;</code> <code>A y(x);</code> <code>y=x;</code>
It can be compared by using equality or inequality operator.	<code>x==y</code> <code>x!=y</code>
It can be dereferenced means we can retrieve the value by using a dereference operator(*) .	<code>*x</code>
A mutable iterator can be dereferenced as an lvalue.	<code>*x = t</code>
A Bidirectional iterator can be incremented.	<code>x++</code> <code>+ +x</code>
A Bidirectional iterator can also be decremented.	<code>x--</code> <code>--x</code>

In the above table, '**A**' is of **bidirectional type**, **x** and **y** are the objects of an iterator type, and '**t**' is an object pointed by the iterator.

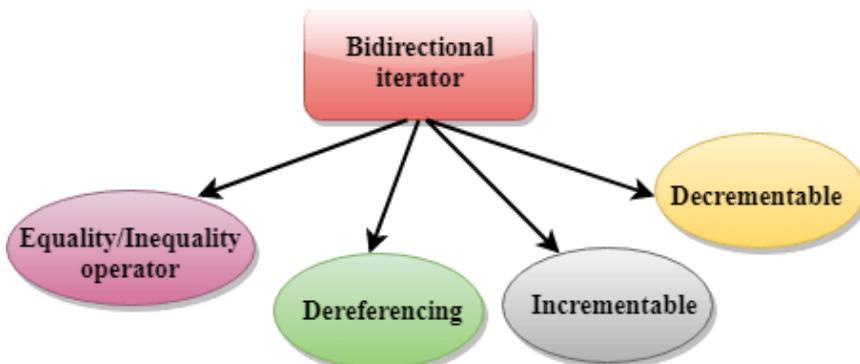
Let's see a simple example:

```
#include <iostream>
#include<iterator>
#include<vector>
using namespace std;
int main()
{
    vector<int> v{1,2,3,4,5};           // vector declaration
    vector<int> ::iterator itr;        // iterator declaration
    vector<int> :: reverse_iterator ritr; // reverse iterator declaration
    for(itr = v.begin();itr!=v.end();itr++)
    {
        cout<<*itr<< " ";
    }
    cout<<'\n';
    for(ritr = v.rbegin();ritr!= v.rend();ritr++)
    {
        cout<<*ritr<< " ";
    }
    return 0;
}
```

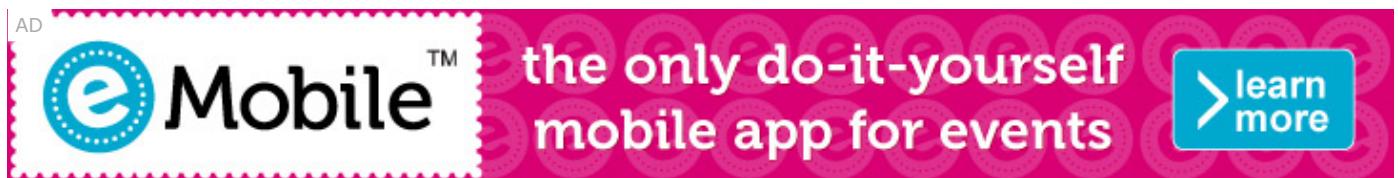
Output:

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

Features of the Bidirectional iterator



- **Equality/Inequality operator:** A bidirectional iterator can be compared by using an **equality or inequality operator**. The two iterators are equal only when both the iterators point to the same position.



Suppose 'A' and 'B' are the two iterators:

```
A==B;  
A!=B;
```

- **Dereferencing:** A bidirectional iterator can also be dereferenced both as an **lvalue** and **rvalue**.

Suppose 'A' is an iterator and 't' is an integer variable:

```
*A = t;  
t = *A
```

- **Incrementable:** A bidirectional iterator can be incremented by using an **operator++()** function.

```
A++;  
++A;
```

- **Decrementable:** A bidirectional iterator can also be decremented by using an **operator--()** function.

```
A--;  
--A;
```

Limitations Of Bidirectional Iterator:

- **Relational operator:** An equality or inequality operator can be used with the bidirectional iterator, but the other iterators cannot be applied on the bidirectional iterator.

Suppose 'A' and 'B' are the two iterators:

```
A==B;      // valid  
A<=B;     // invalid
```

- **Arithmetic operator:** An arithmetic operator cannot be used with the bidirectional iterator as it accesses the data sequentially.

```
A+2;      // invalid  
A+1;      // invalid
```

- **Offset dereference operator:** A Bidirectional iterator does not support the offset dereference operator or subscript operator [] for the random access of an element.

← Prev

Next →

AD

 [For Videos Join Our Youtube Channel: Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

C++ Forward Iterator

- **Forward Iterator** is a combination of Bidirectional and Random Access iterator. Therefore, we can say that the forward iterator can be used to read and write to a container.
- **Forward iterators** are used to read the contents from the beginning to the end of a container.
- **Forward iterator** use only increments operator (++) to move through all the elements of a container. Therefore, we can say that the forward iterator can only move forward.
- A Forward iterator is a multi-pass iterator.

Operations Performed on the Forward Iterator:

Properties	Valid Expressions
It is default constructible.	A x;
It is a copy-constructible.	A x(y);
It is a copy-assignable.	y = x;
It can be compared either by using an equality or inequality operator.	a==b; a!=b;
It can be incremented.	a++; ++a;
It can be dereferenced as an rvalue.	*a;
It can also be dereferenced as an lvalue.	*a = t;

Where 'A' is a forward iterator type, and x and y are the objects of a forward iterator type, and t is an object pointed by the iterator type object.

Let's see a simple example:

```
#include <iostream>
#include <fstream>
#include <iterator>
#include <vector>
```

```

using namespace std;

template<class ForwardIterator> // function template
void display(ForwardIterator first, ForwardIterator last) // display function
{
    while(first!=last)
{
    cout<<*first<< " ";
    first++;
}
}

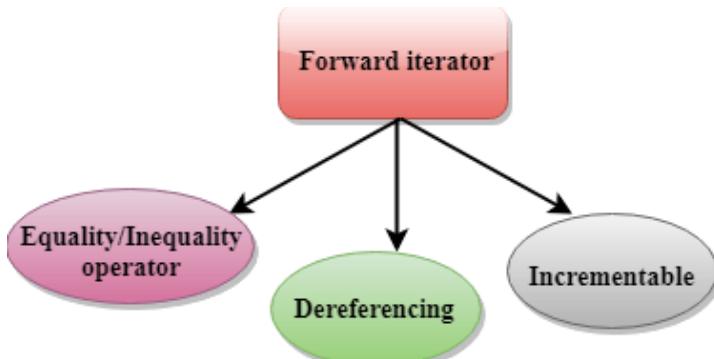
int main()
{
vector<int> a; // declaration of vector.
for(int i=1;i<=10;i++)
{
    a.push_back(i);
}
display(a.begin(),a.end()); // calling display() function.

return 0;
}

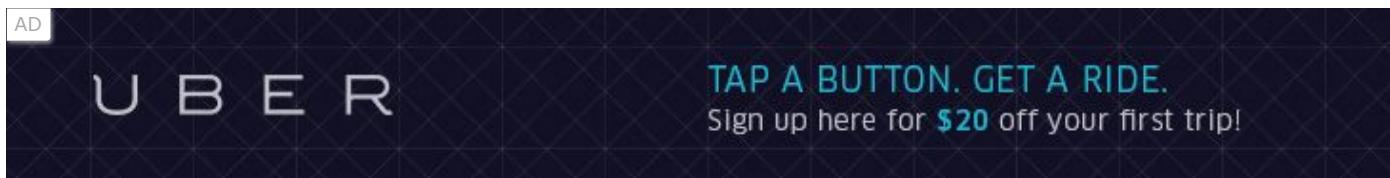
```

Output:

```
1 2 3 4 5 6 7 8 9 10
```

Features of the Forward Iterator:

- **Equality/Inequality operator:** A forward iterator can be compared by using equality or an inequality operator.



Suppose 'A' and 'B' are the two iterators:

```
A==B;      // equality operator
A!=B;      // inequality operator
```

- **Dereferencing:** We can dereference the forward iterator as an rvalue as well as an lvalue. Therefore, we can access the output iterator and can also assign the value to the output iterator.

Suppose 'A' is an iterator and 't' is an integer variable:

```
*A = t;
t = *A;
```

- **Incrementable:** A forward iterator can be incremented but cannot be decremented.

Suppose 'A' is an iterator:

```
A++;
++A;
```

Limitations of the Forward Iterator:

- **Decrementable:** A forward iterator cannot be decremented as it moves only in the forward direction.

Suppose 'A' is an iterator:

```
A--;
// invalid
```

- **Relational Operators:** A forward iterator can be used with the equality operator, but no other relational operators can be applied on the forward iterator.

Suppose 'A' and 'B' are the two iterators:

```
A==B;    // valid  
A>=B;   // invalid
```

- **Arithmetic Operators:** Arithmetic operators cannot be used with the forward iterator.

```
A+2;    // invalid  
A+3;    // invalid
```

- **Random Access:** A forward iterator does not provide the random access of an element. It can only iterate through the elements of a container.

← Prev

Next →

AD

 [For Videos Join Our Youtube Channel: Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

C++ Input Iterator

- Input Iterator is an iterator used to read the values from the container.
- Dereferencing an input iterator allows us to retrieve the value from the container.
- It does not alter the value of a container.
- It is a one-way iterator.
- It can be incremented, but cannot be decremented.
- Operators which can be used for an input iterator are increment operator(++) , decrement operator(--), dereference operator(*) , not equal operator(!=) and equal operator(==).
- An input Iterator is produced by the **Istream**.
- A Forward iterator, bidirectional iterator, and random access iterator are all valid input iterators.

Property	Valid Expressions
An input iterator is a copy-constructible, copy-assignable and destructible.	X b(a); b = a;
It can be compared by using a equality or inequality operator.	a == b; a != b;
It can be dereferenced.	*a;
It can be incremented.	++a;

Where 'X' is of input iterator type while 'a' and 'b' are the objects of an iterator type.

Features of Input iterator:

- **Equality/Inequality operator:** An input iterator can be compared by using an equality or inequality operator. The two iterators are equal only when both the iterators point to the same location otherwise not. Suppose 'A' and 'B' are the two iterators:

AD


VIDEO STREAMING FOR THE AWAKENED MIND
[GET STARTED NOW](#)

A == B; // equality operator

```
A!=B; // inequality operator
```

Let's see a simple example:

```
#include <iostream>
#include<vector>
#include<iterator>
using namespace std;
int main()
{
    vector<int> v{1,2,3,4,5};
    vector<int>::iterator itr,itr1;
    itr=v.begin();
    itr1=v.begin()+1;
    if(itr==itr1)
        std::cout << "Both the iterators are equal" << std::endl;
    if(itr!=itr1)
        std::cout << "Both the iterators are not equal" << std::endl;
    return 0;
}
```

Output:

```
Both the iterators are not equal
```

In the above example, `itr` and `itr1` are the two iterators. Both these iterators are of vector type. The '`itr`' is an iterator object pointing to the first position of the vector and '`itr1`' is an iterator object pointing to the second position of the vector. Therefore, both the iterators point to the same location, so the condition `itr1!=itr` returns true value and prints "**Both the iterators are not equal**".

- **Dereferencing an iterator:** We can dereference an iterator by using a dereference operator(`*`). Suppose 'A' is an iterator:

```
*A // Dereferencing 'A' iterator by using *.
```

Let's see a simple example:

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

int main()

vector<int> v{11,22,33,44};
vector<int>::iterator it;
it = v.begin();
cout<<*it;
return 0;
```

Output:

```
11
```

In the above example, 'it' is an iterator object pointing to the first element of a vector 'v'. A dereferencing an iterator *it returns the value pointed by the iterator 'it'.

- **Swappable:** The two iterators pointing two different locations can be swapped.

Let's see a simple example:

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

int main()
{
    vector<int> v{11,22,33,44};
    vector<int>::iterator it,it1,temp;
    it = v.begin();
    it1 = v.begin()+1;
    temp=it;
    it=it1;
    it1=temp;
    cout<<*it<<" ";
```

```
cout<<*it1;  
return 0;  
}
```

Output:

```
22 11
```

In the above example, 'it' and 'it1' iterators are swapped by using an object of a third iterator, i.e., **temp**.

← Prev

Next →

AD

 YouTube For Videos Join Our Youtube Channel: [Join Now](#)

Feedback

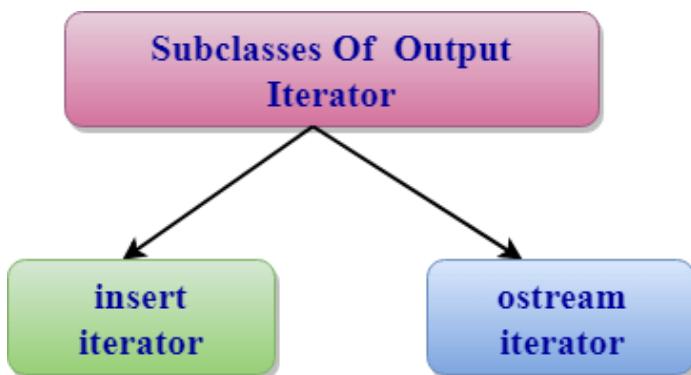
- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share



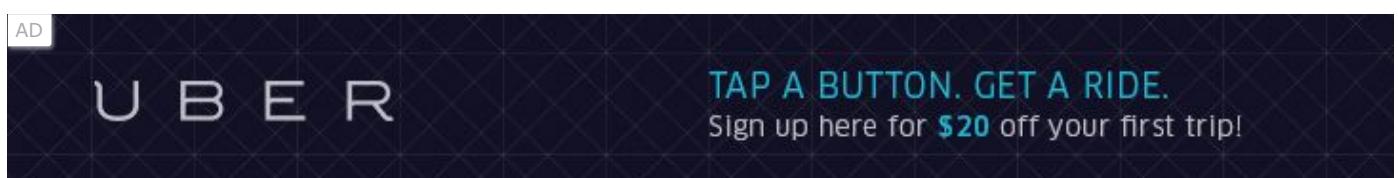
C++ Output Iterator

- Output Iterator is an iterator used to modify the value in the container.
- Dereferencing an output iterator allows us to alter the value of the container.
- It does not allow us to read the value from the container.
- It is a one-way and write-only iterator.
- It can be incremented, but cannot be decremented.
- Operators that can be used for an output iterator **are increment operator(++)**, **decrement operator(--)** and **assignment operator(=)**.
- **There are two main subclasses of an Output Iterator are:**
 - **insert iterator**
 - **ostream iterator**



Insert Iterator

- An insert iterator is an iterator used to insert the element in a specified position.
- An assignment operator on the `insert_iterator` inserts the new element at the current position.



Syntax

```
template<class Container, class Iterator>
insert_iterator<container> inserter(Container &x,Iterator it);
```

Parameters

x: It is the container on which the new element is to be inserted.

it: It is an iterator object pointing to the position which is to be modified.

Let's see a simple example:

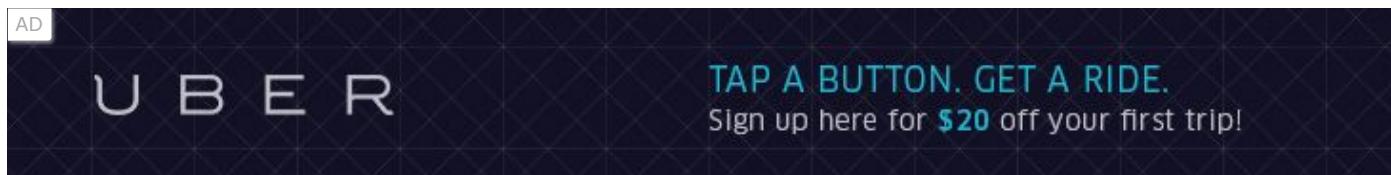
```
#include <iostream>    // std::cout
#include <iterator>    // std::front_inserter
#include <vector>      // std::list
#include <algorithm>   // std::copy
using namespace std;

int main () {
    vector<int> v1,v2;
    for (int i=1; i<=5; i++)
    {
        v1.push_back(i);
        v2.push_back(i+2);
    }
    vector<int>::iterator it = v1.begin();
    advance (it,3);
    copy (v2.begin(),v2.end(),inserter(v1,it));
    cout<<"Elements of v1 are :";
    for ( it = v1.begin(); it!= v1.end(); ++it )
        cout << ' ' << *it;
    cout << '\n';
    return 0;
}
```

Output:

```
Elements of v1 are : 1 2 3 3 4 5 6 7 4 5
```

In the above example, `insert_iterator` is applied on the `copy` algorithm to insert the elements of the vector `v2` into the vector `v1` at a specified position pointed by `it`.



Ostream iterator

- An ostream iterators are the output iterators used to write to the output stream such as cout successively.
- An ostream iterator is created using a basic_ostream object.
- When an assigment operator is used on the ostream iterator, it inserts a new element into the output stream.

Syntax

```
template<class T, class charT=char, class traits=char_traits<charT>>
class ostream_iterator;
```

Member functions of Ostream Iterator class

```
Ostream_iterator<T, charT, traits>& operator=(const T& value);
Ostream_iterator<T, charT, traits>& operator*();
Ostream_iterator<T, charT, traits>& operator++();
Ostream_iterator<T, charT, traits>& operator++(int);
```

Parameters

- **T:** It is the type of elements to be inserted into the container.
- **charT:** The type of elements that ostream can handle, for example, char ostream.
- **traits:** These are the character traits that the stream handles for the elements.

Let's see a simple example:

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

```
int main()
{
    vector<int> v;
    for(int i=1;i<=5;i++)
    {
        v.push_back(i*10);
    }
    ostream_iterator<int> out(cout, ",");
    copy(v.begin(),v.end(),out);
    return 0;
}
```

Output:

```
10,20,30,40,50
```

In the above example, out is an object of the ostream_iterator used to add the delimiter ',' between the vector elements.

Let's see another simple example of ostream iterator:

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

int main()
{
    ostream_iterator<int> out(cout, ",");
    *out = 5;
    out++;
    *out = 10;
    out++;
    *out = 15;
    return 0;
}
```

Output:

```
5,10,15,
```

Features Of Output Iterator

- **Equality/Inequality Operator:** Output iterators cannot be compared either by using equality or inequality operator. Suppose X and Y are the two iterators:

```
X==Y; invalid
```

```
X!=Y; invalid
```

- **Dereferencing:** An output iterator can be dereferenced as an lvalue.

```
*X=7;
```

- **Incrementable:** An output iterator can be incremented by using operator`++()` function.

```
X++;
```

```
++X;
```

Limitations Of Output Iterator

- **Assigning but no accessing:** We can assign an output iterator as an lvalue, but we cannot access them as an rvalue.

Suppose 'A' is an output iterator type and 'x' is a integer variable:

```
*A = x;           // valid  
x = *A;          // invalid
```

- **It cannot be decremented:** We can increment the output iterator by using operator`++()` function, but we cannot decrement the output iterator.

Suppose 'A' is an output iterator type:

```
A++;      // not valid  
++A;      // not valid
```

- **Multi-pass algorithm:** An output iterator cannot be used as a multi-pass algorithm. Since an output iterator is unidirectional and can move only forward. Therefore, it cannot be used to move through the container multiple times
- **Relational Operators:** An output iterator cannot be compared by using any of the relational operators.

Suppose 'A' and 'B' are the two iterators:

```
A == B;    // not valid  
A != B;    // not valid
```

- **Arithmetic Operators:** An output iterator cannot be used with the arithmetic operators. Therefore, we can say that the output iterator only moves forward in a sequential manner.

Suppose 'A' is an output iterator:

```
A + 2;    // invalid  
A + 5;    // invalid
```

← Prev

Next →

AD