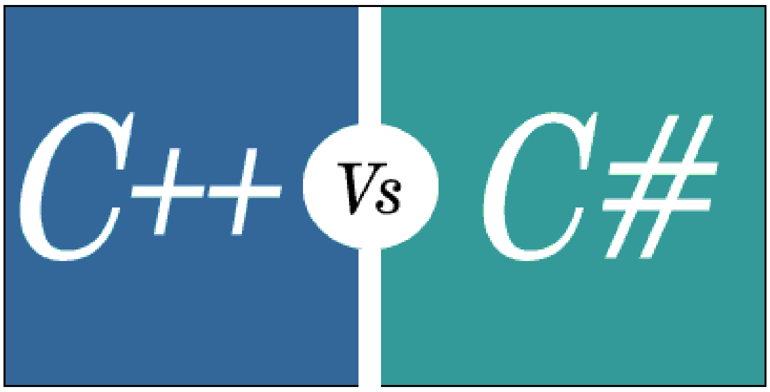
C++ vs C#

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



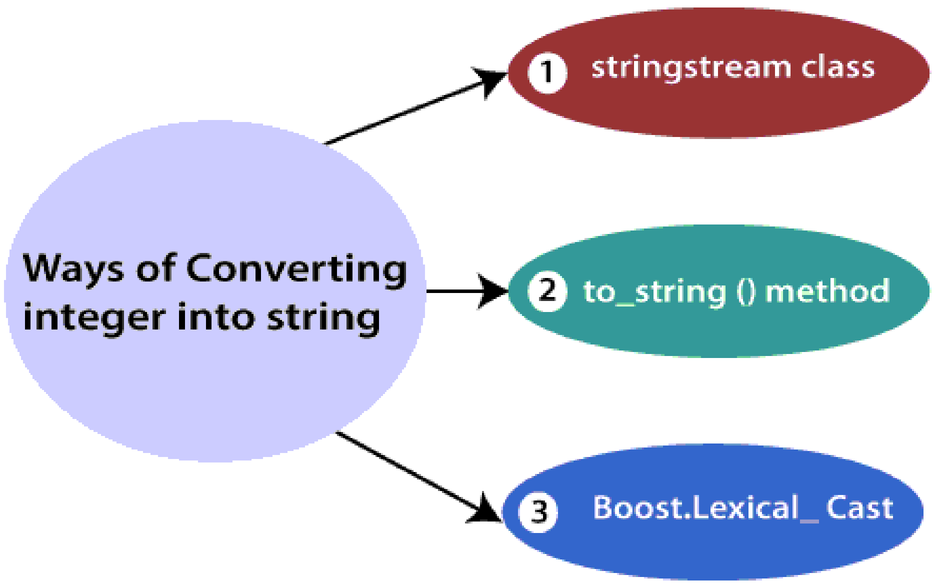
- **Type of language**
  C++ is a low-level language, while C# is a high-level language.
- **Lightweight language**
  C++ is a lightweight language as compared to C# language as the libraries of C# language need to be included before compilation due to which size of binaries in C# language is more than 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.

- **Multiple Inheritance**
  C++ language supports multiple inheritances, while C# language does not support the multiple inheritances.
- **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.
- **Default access specifier**
  In C++, the default access specifier is public while in C#, the default access specifier is private.
- **Object Oriented**
  C++ language is not a complete object-oriented language while C# language is a pure object-oriented programming language.
- **Bound checking**
  C++ language does not support the bound checking for arrays while C# language supports the bound checking for arrays.
- **For each loop**
  C++ language does not support the for each loop while C# language supports the for each loop.
- **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.
- **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.
- **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.

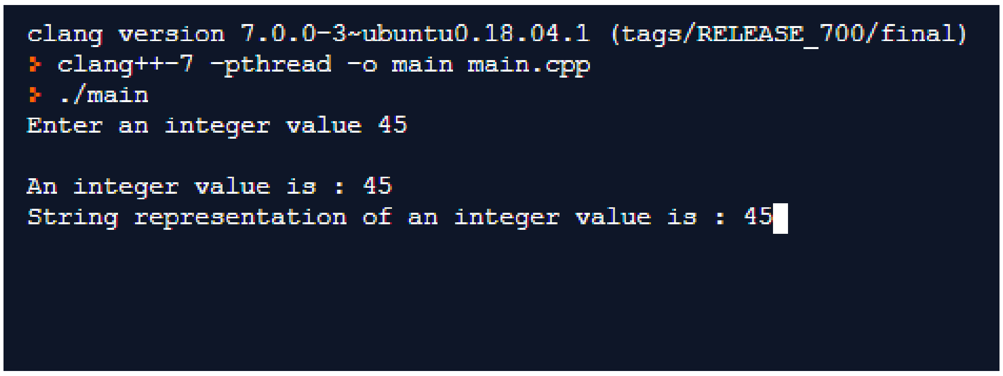
Let's understand the concept of operators through an example.

- **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



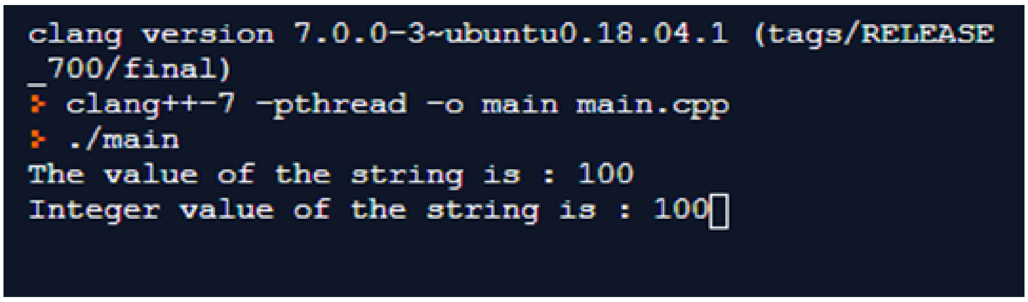
```
clang version 7.0.0-3~ubuntu0.18.04.1 (tags/RELEASE_700/final)
› clang++-7 -pthread -o main main.cpp
› ./main
Enter an integer value 45

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
```

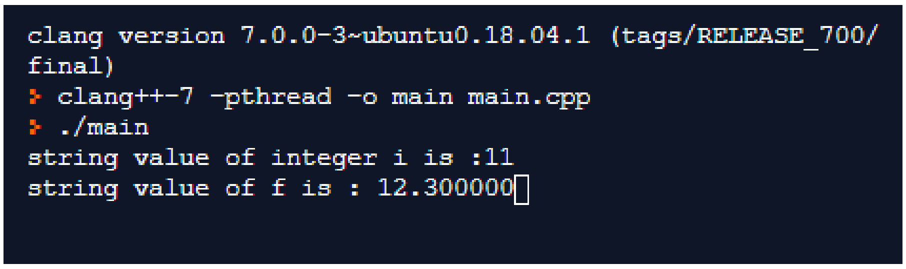
Conversion of an integer into a string by using to_string() method.

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
```

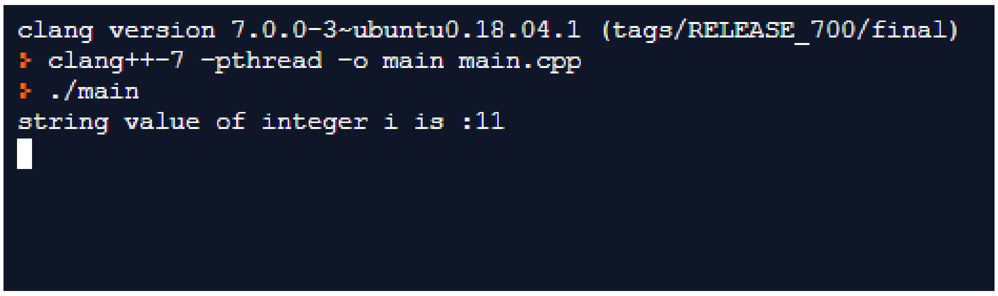
Conversion of an integer into a string by using a boost.lexical cast.

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";


}
```

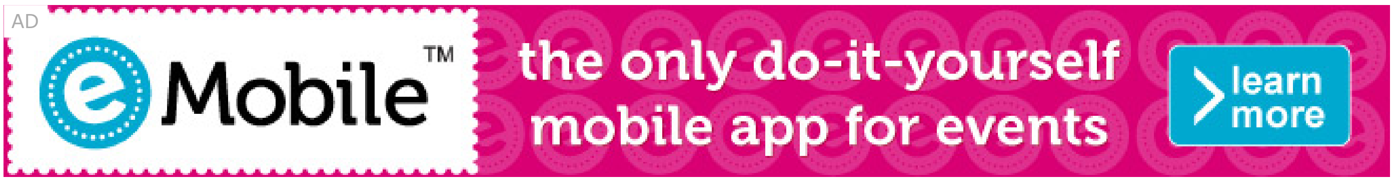
Output



```
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
```

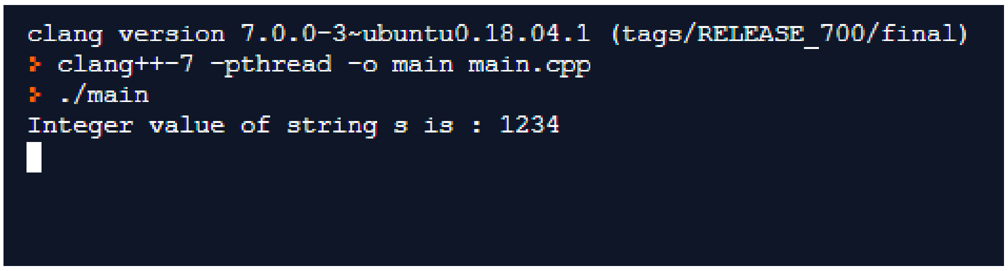
In the above example, we have converted the value of 'i' variable into a string value by using lexical_cast() function.

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.

← Prev Next →

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.

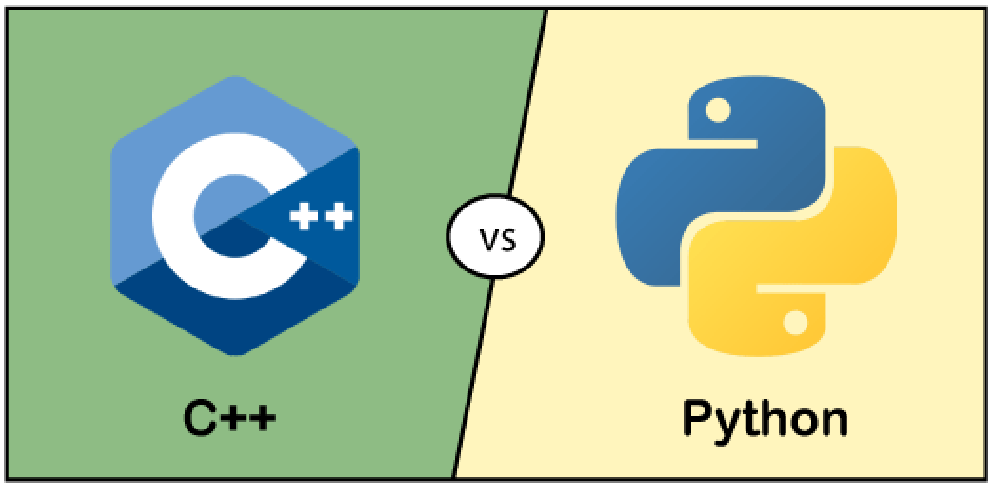
It was designed for system programming and embedded system, but later on, it was used in developing various applications such as desktop applications, video games, servers such as e-commerce, Web search or SQL servers and performance-critical applications such as telephone switches.

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	`struct structure_name{`
	`type struct_member 1;`
	`type struct_member 2;`
	`type struct_member 3;`
	`.`
	`type struct_memberN;`
	`};`
		`class class_name{`
		`data member;`
		`member function;`
		`};`

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.

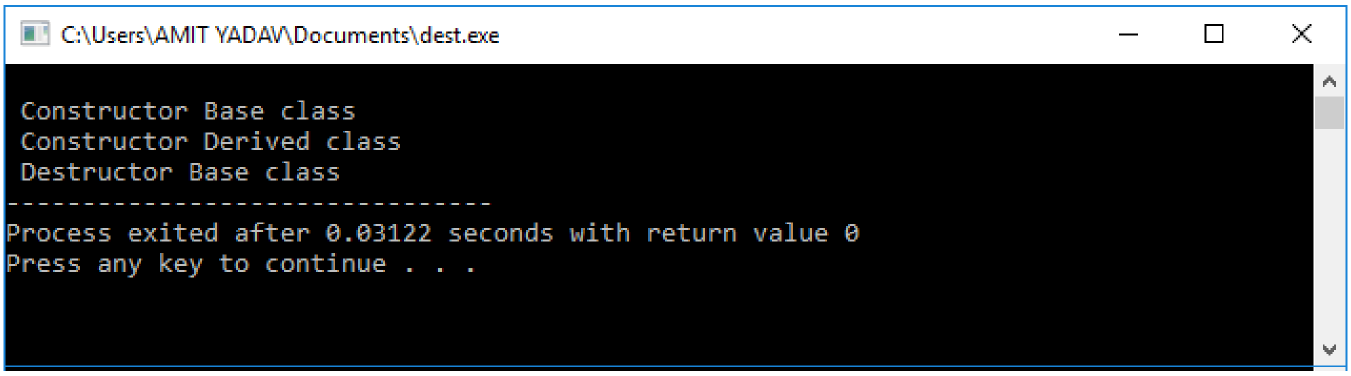
Why we use virtual destructor in C++?

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:



C:\Users\AMIT YADAV\Documents\dest.exe

```
 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.

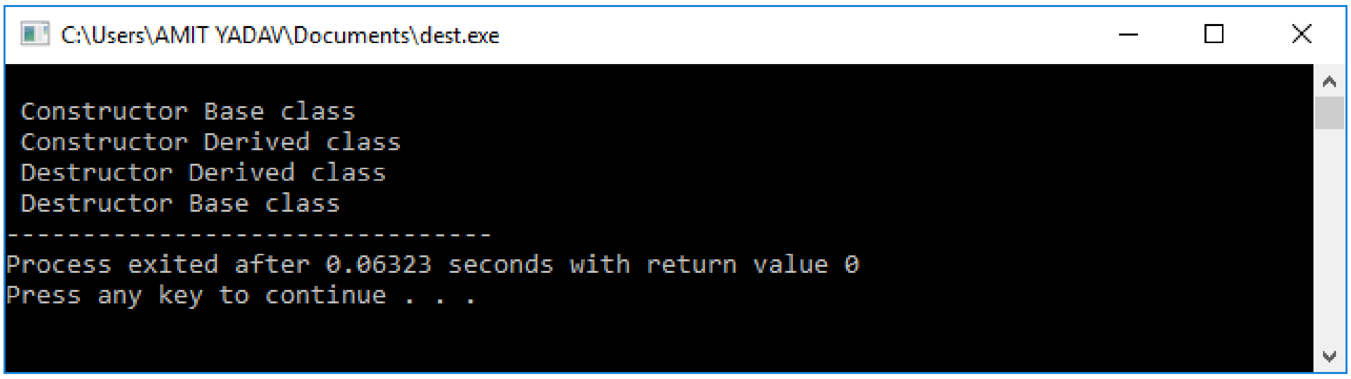
> Note: If the base class destructor does not use a virtual keyword, only the base class destructor will be called or deleted its occupied space because the pointer object is pointing to the base class. So it does not call the derived class destructor to free the memory used by the derived class, which 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 . . .
```

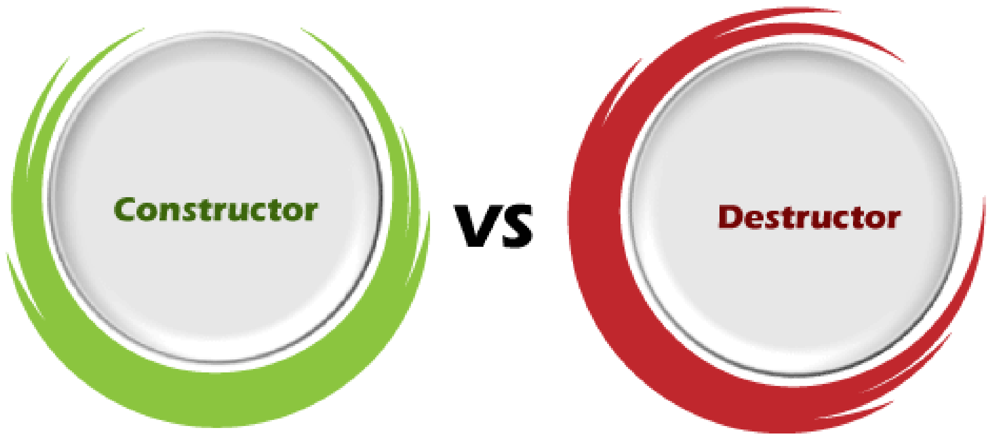
In the above program, we have used a virtual destructor inside the base class that calls the derived class' destructor before calling the base class' destructor and release the spaces or resolve the memory leak issue in the program.

← Prev

Next →

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

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
};
```

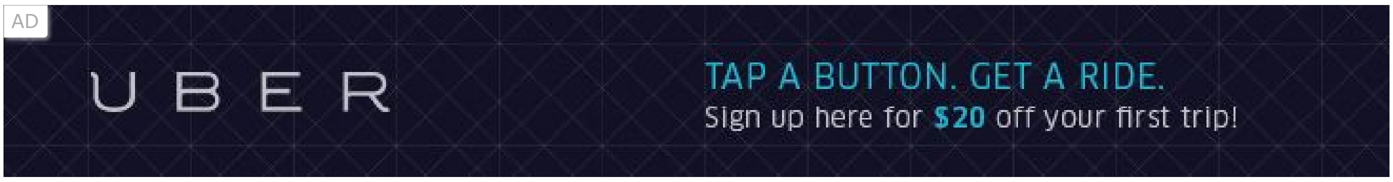
Here, we use the tilde symbol for defining the destructor in C++ programming.

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;
}
```


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

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	Constructor has four types: ◦ Default constructor ◦ Copy constructor ◦ Parameterized constructor ◦ Dynamic constructor	Destructors have no classes.
Declaration	The following declaration is used for creating a constructor: `class class_name` `{` `……….` `public` `class_name ([parameter list])` `{` `………………….` `}` `};`	The following declaration is used for creating a destructor: `class class_name` `{` `…………….;` `…………….;` `public:` `~xyz();` `{` `…………` `};`

← 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

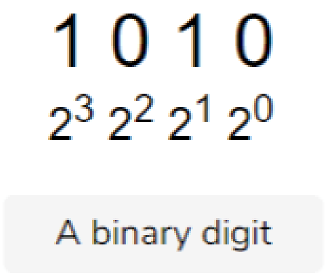
A bit is defined as the basic unit which stores the data in digital notation.

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.



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++.

- **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.

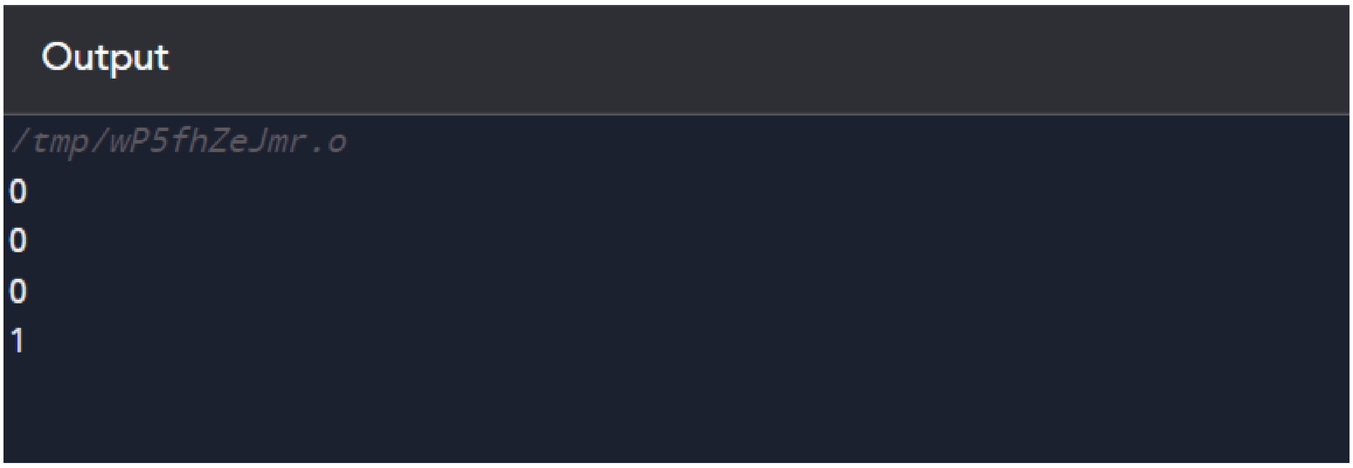
A	B	A&&B
0	0	0
0	1	0
1	0	0
1	1	1

In the last row, A and B are high, resulting in a high output.

C++ Program

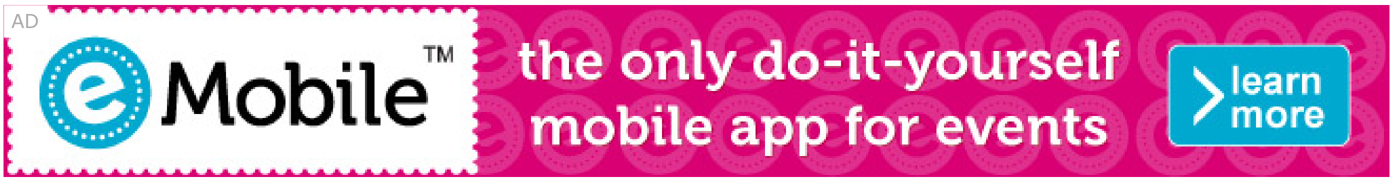
```
#include <iostream>
using namespace std;

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
```

- **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

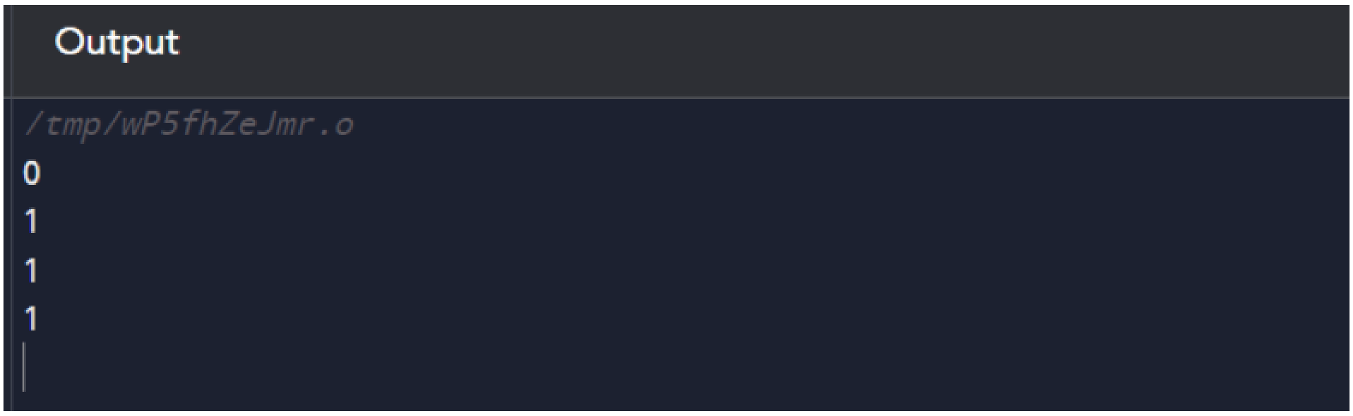
Here we can see the first row. Both inputs A and B are low, which results in 0(a low output).

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
```

- **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 !.

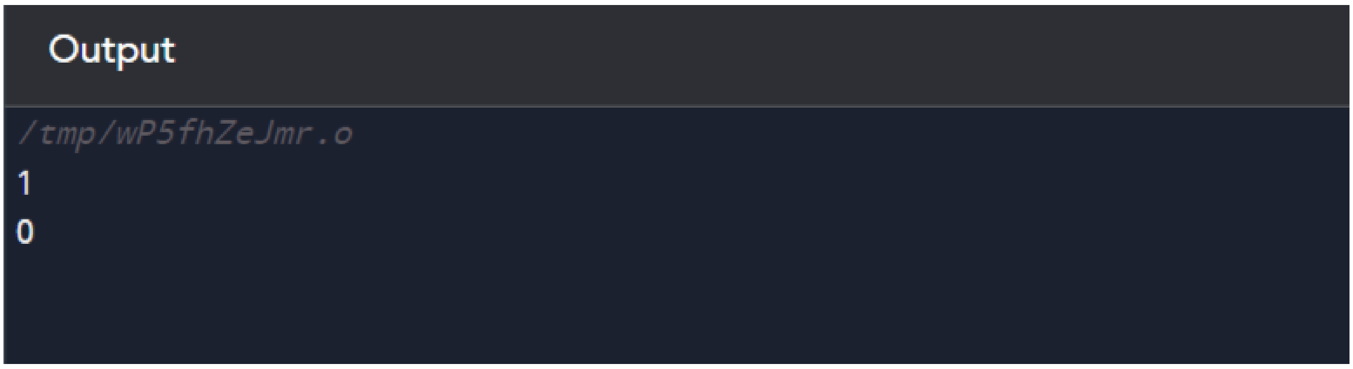
Let us look at the truth table of the NOT operator.

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:



```
/tmp/wP5fhZeJmr.o
1
0
```

- **Left shift operator**

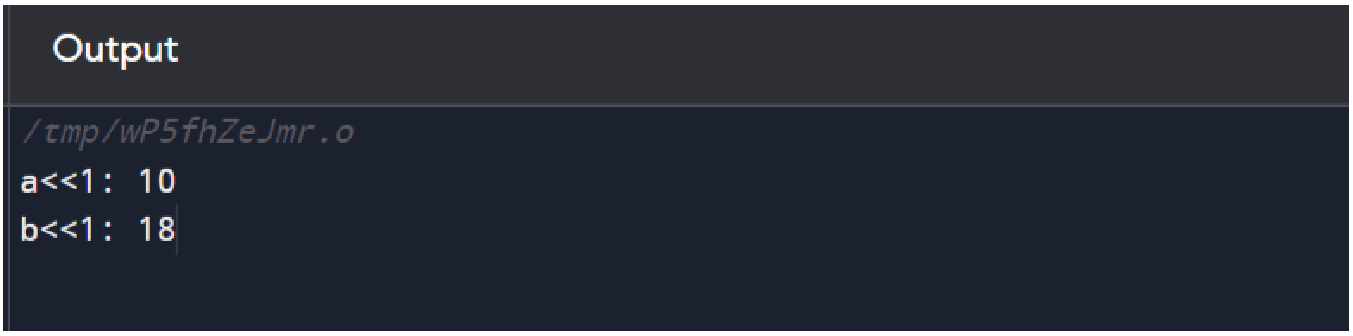
The left shift operator takes an operand and the value of the left operand is moved left by the number of bits specified by the right operand.

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
```

- **Right shift operator**

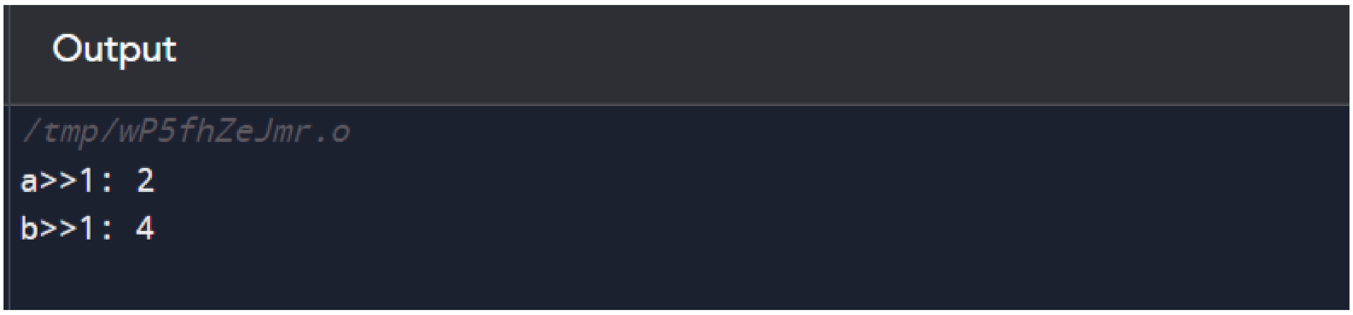
The right shift operator takes an operand and the value of the right operand is moved right by the number of bits specified by the right operand.

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";

    // The result is 00000100
    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 -

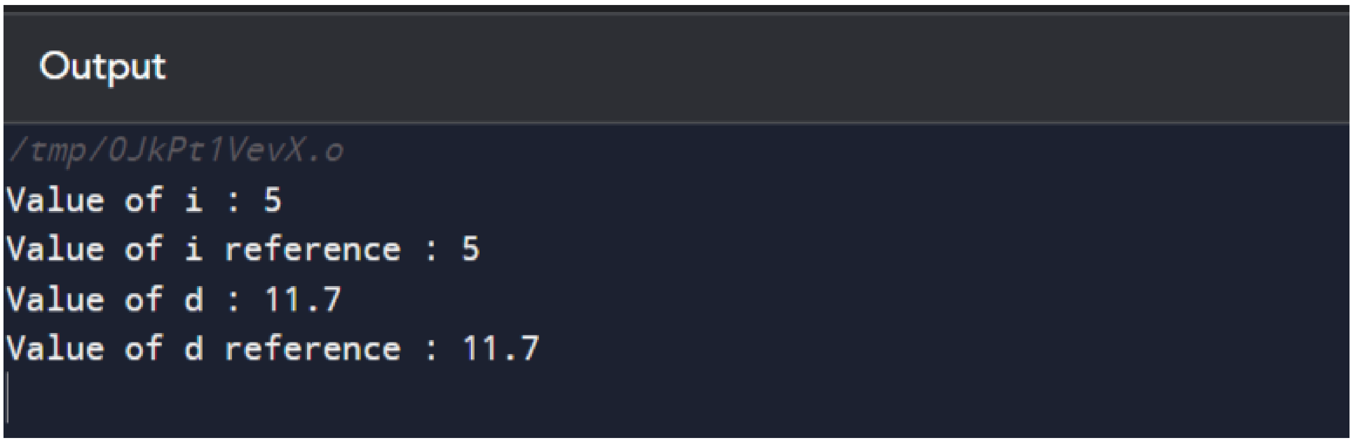
int& x = i // Here x will be called as the integer variable initialised to r

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 cant 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.

AD

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**.

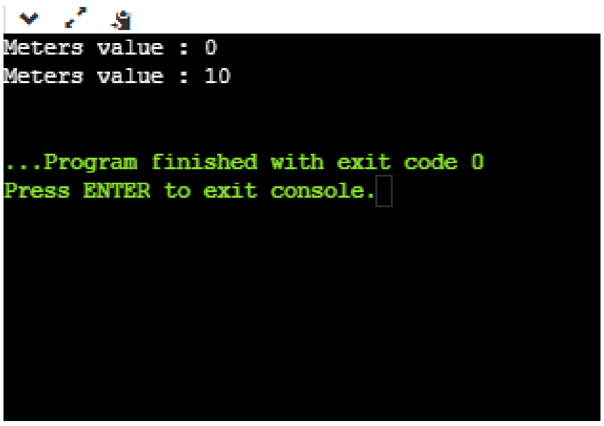
Let's understand the friend function through an example.

```
#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.

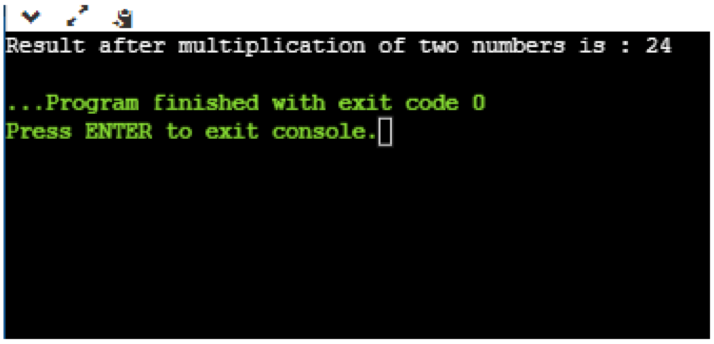
Let's understand through an example.

```
// 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.
```

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.

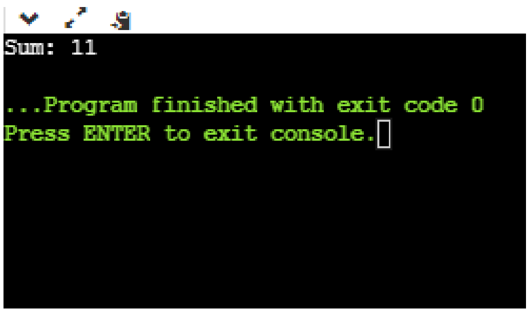
Let's understand through an example.

```
// 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():

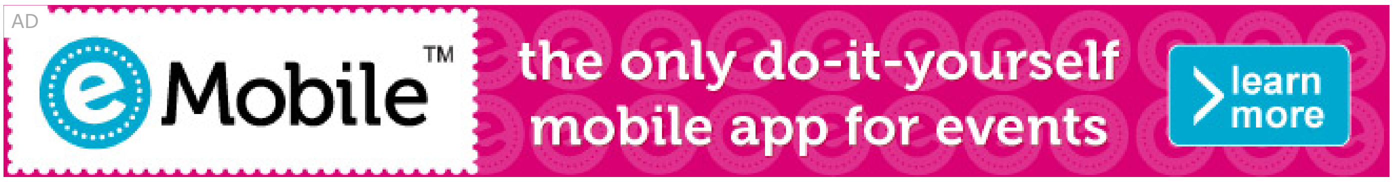
It is used to close the graphics function.

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 twnkl = 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);
    fillellipse(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.

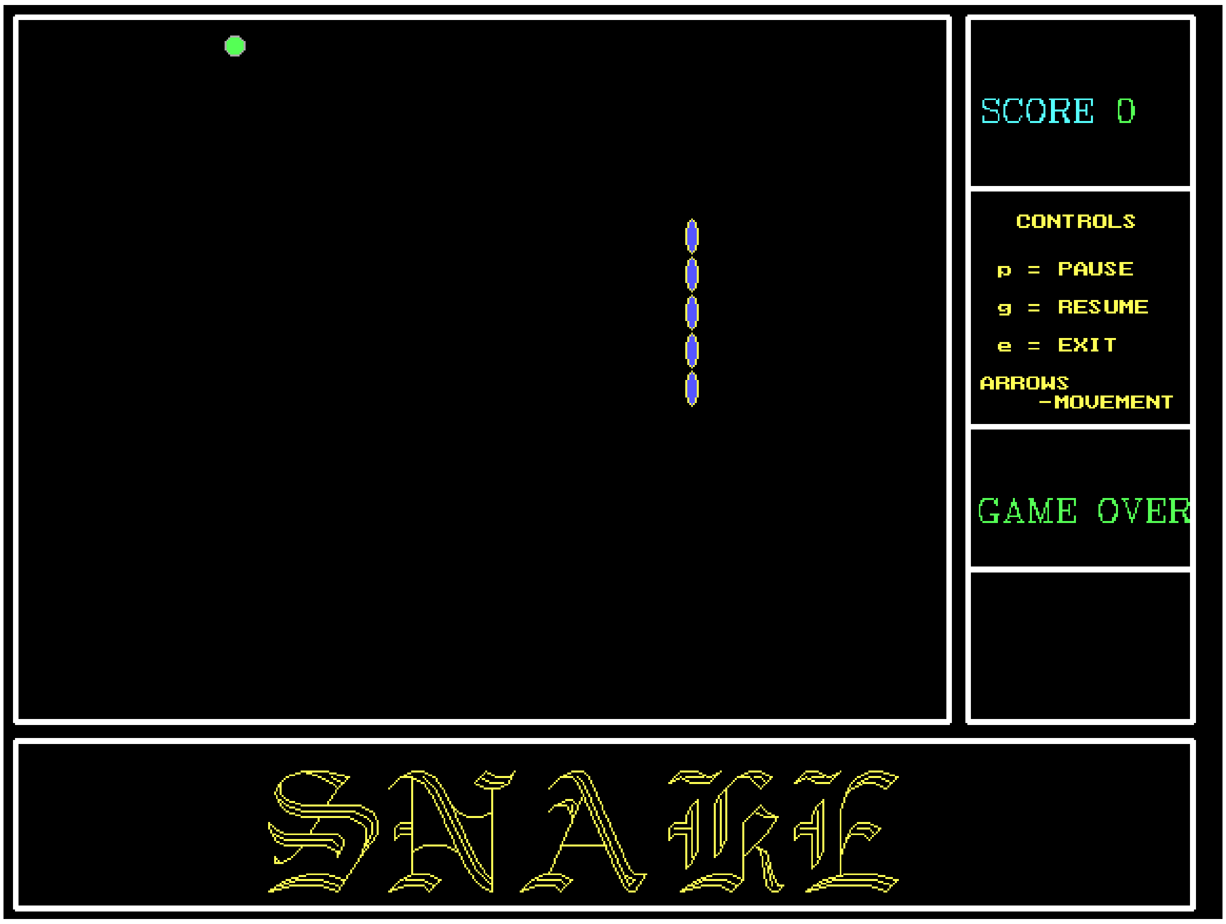
The basic functionalities of this game are given below.

- 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:


```
SCORE 0

CONTROLS
p = PAUSE
g = RESUME
e = EXIT
ARROWS
   -MOVEMENT

GAME OVER

SNAKE
```

← 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);A
   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:

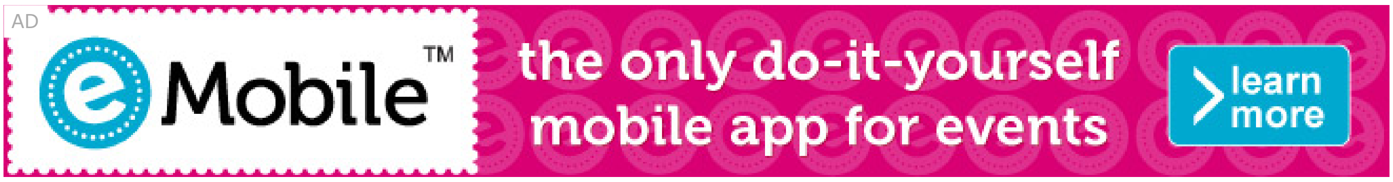
- An inline function can be used when the performance is required.
- It can be used over the macros.
- We can use the inline function outside the class so that we can hide the internal implementation of the function.

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 →

Youtube 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 preceeding 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;
}
```

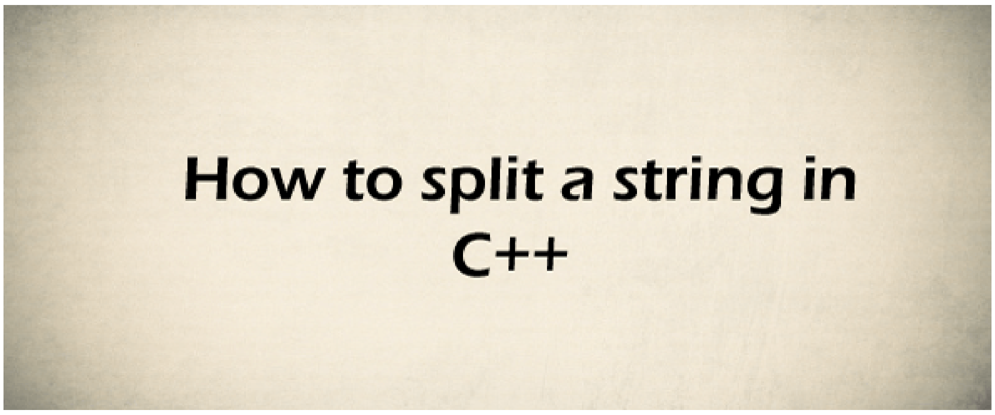
Differences between the virtual function and pure virtual function

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 += 1;
            startIndex = endIndex + 1;
        }
        i++;
        }
}

int main()
{
    string str = "Program to split strings using custom split function.";
    char seperator = ' '; // space
    split(str, seperator);
    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).

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.

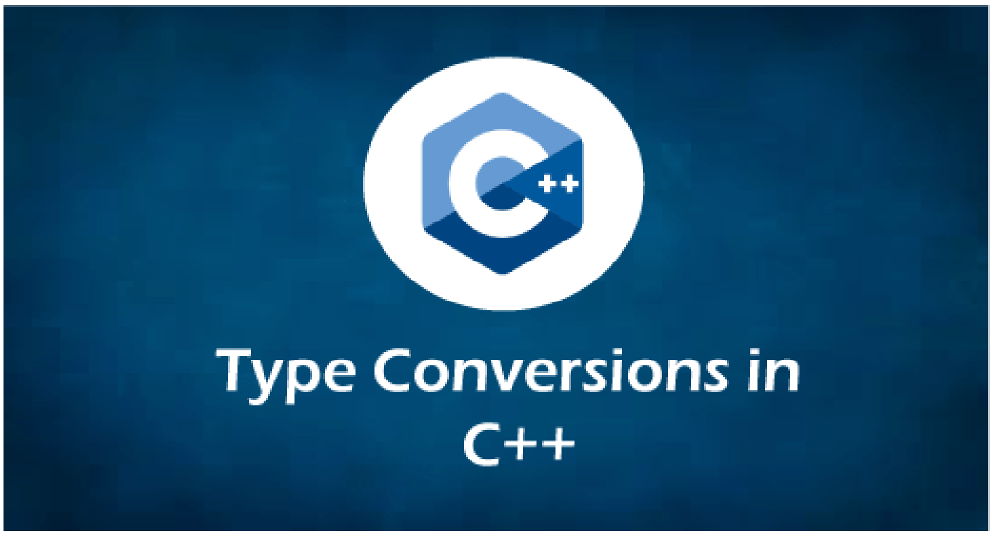
Disadvantage of the range-based for loop

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 →

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
```

← Prev Next →

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 (n1 and n2), 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 n1 and n2

Step 2: Store the smallest common multiple of n1 and n2 into the max variable.

Step 3: Validate whether the max variable is divisible by n1 and n2, 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

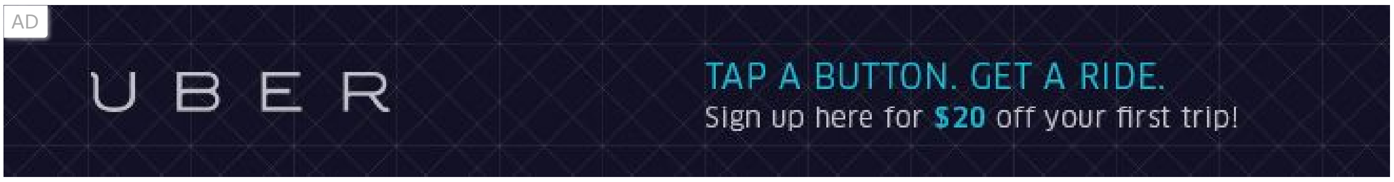
Program5.cpp

```
#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

AD


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

```
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 ty

     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 " << intdata
    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(\"" << strdata1 << "\") is " << i
    cout << " The conversion of string to an integer value using atoi(\"" << strdata2 << "\") is " << i
    cout << " The conversion of string to an integer value using atoi(\"" << strdata3 << "\") 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 →