Contents

[**Basic Concepts of Object Oriented Programming** 3](#_Toc510686230)

[Classes and Objects. 4](#_Toc510686231)

[Access Modifiers in C plus plus 4](#_Toc510686232)

[**Inheritance in C++** 6](#_Toc510686233)

[**Polymorphism in C++** 12](#_Toc510686234)

[**Encapsulation in C++** 13](#_Toc510686235)

[**Abstraction in C++** 13](#_Toc510686236)

[**Friend class and function in C++** 14](#_Toc510686237)

[**Local Classes in C++** 15](#_Toc510686238)

[**Nested Classes in C++** 15](#_Toc510686239)

[**Multiple Inheritance in C++** 16](#_Toc510686240)

[**Some interesting facts about static member functions in C++** 20](#_Toc510686241)

[**Static data members in C++** 21](#_Toc510686242)

[**Can a C++ class have an object of self type?** 24](#_Toc510686243)

[**Why is the size of an empty class not zero in C++?** 25](#_Toc510686244)

[**Why is the size of an empty class not zero in C++?** 28](#_Toc510686245)

[**Simulating final class in C++** 31](#_Toc510686246)

[**Structure vs class in C++** 33](#_Toc510686247)

[**Difference between C structures and C++ structures** 35](#_Toc510686248)

[**C/C++ Preprocessors** 37](#_Toc510686249)

[**What happen when we exceed valid range of built-in data types in C++?** 42](#_Toc510686250)

[**C++ string class and its applications** 45](#_Toc510686251)

[Pointers in C and C++ | Set 1 (Arithmetic and Array**)** 50](#_Toc510686252)

[**Multidimensional Pointer Arithmetic in C/C++** 51](#_Toc510686253)

[**References in C++** 52](#_Toc510686254)

[**What is Array Decay in C++? How can it be prevented?** 56](#_Toc510686255)

[**Namespace in C++ | Set 1 (Introduction)** 59](#_Toc510686256)

[**namespace in C++ | Set 2 (Extending namespace and Unnamed namespace)** 63](#_Toc510686257)

[**Namespace in C++ | Set 3 (Accessing, creating header, nesting and aliasing)** 66](#_Toc510686258)

[**Inline namespaces and usage of the “using” directive inside namespaces** 70](#_Toc510686259)

[**Default Arguments in C++** 74](#_Toc510686260)

[**Inline Functions in C++** 75](#_Toc510686261)

[**Return from void functions in C++** 80](#_Toc510686262)

[**Returning multiple values from a function using Tuple and Pair in C++** 83](#_Toc510686263)

[**Difference between “int fun()” and “int fun(void)” in C/C++?** 85](#_Toc510686264)

[**Functors in C++** 87](#_Toc510686265)

[**Const member functions in C++** 89](#_Toc510686266)

[**The C++ Standard Template Library (STL)** 91](#_Toc510686267)

[**Sort in C++ Standard Template Library (STL)** 92](#_Toc510686268)

[**Binary Search in C++ Standard Template Library (STL)** 93](#_Toc510686269)

[**std::bsearch in C++** 95](#_Toc510686270)

[**Pair in C++ Standard Template Library (STL)** 98](#_Toc510686271)

[**Vector in C++ STL** 104](#_Toc510686272)

[Vector 104](#_Toc510686273)

[**Ways to copy a vector in C++** 107](#_Toc510686274)

[**Modifiers for Vector in C++ STL** 114](#_Toc510686275)

[**Passing vector to a function in C++** 120](#_Toc510686276)

[**List in C++ Standard Template Library (STL)** 123](#_Toc510686277)

[List 123](#_Toc510686278)

[**List in C++ | Set 2 (Some Useful Functions)** 126](#_Toc510686279)

[**Forward List in C++ | Set 1 (Introduction and Important Functions)** 134](#_Toc510686280)

[**Forward List in C++ | Set 2 (Manipulating Functions)** 141](#_Toc510686281)

https://www.geeksforgeeks.org/c-plus-plus/

# **Basic Concepts of Object Oriented Programming**

The main aim of OOP is to bind together the data and the functions that operates on them so that no other part of code can access this data except that function.

Let us start by learning the different basic characteristics of an OO language

**Object**: Objects are basic run-time entities in an object oriented system, objects are instances of a class these are defined user defined data types.Object take up space in memory and have an associated address like a record in pascal or structure or union in C.When a program is executed the objects interact by sending messages to one another. Each object contains data and code to manipulate the data. Objects can interact without having to know details of each others data or code, it is sufficient to know the type of message accepted and type of response returned by the objects.

**Class**: Class is a blueprint of data and functions or methods. Class does not take any space. Class is a uiser defined data type like structures or unions in C.

**Encapsulation and Data abstraction:** Wrapping up(combing) of data and functions into a single unit is known as encapsulation. The data is not accessible to the outside world and only those functions which are wrapping in the class can access it. This insulation of the data from direct access by the program is called data hiding or information hiding.

Data abstraction refers to, providing only needed information to the outside world and hiding implementation details. For example, consider a class Complex with public functions as getReal() and getImag(). We may implement the class as an array of size 2 or as two variables. The advantage of abstractions is, we can change implementation at any point, users of Complex class wont’t be affected as out method interface remains same. Had our implementation be public, we would not have been able to change it.  
  
**Inheritance:** inheritance is the process by which objects of one class acquire the properties of objects of another class. It supports the concept of hierarchical classification. Inheritance provides re usability. This means that we can add additional features to an existing class without modifying it.  
  
**Polymorphism:** polymorphism means ability to take more than one form. An operation may exhibit different behaviors in different instances. The behavior depends upon the types of data used in the operation.  
C++ supports operator overloading and function overloading.  
Operator overloading is the process of making an operator to exhibit different behaviors in different instances is known as operator overloading.  
Function overloading is using a single function name to perform different types of tasks.  
Polymorphism is extensively used in implementing inheritance.

**Dynamic Binding:** In dynamic binding, the code to be executed in response to function call is decided at runtime. C++ has [virtual functions](https://www.geeksforgeeks.org/virtual-functions-and-runtime-polymorphism-in-c-set-1-introduction/) to support this.

**Message Passing:** Objects communicate with one another by sending and receiving information to each other. A message for an object is a request for execution of a procedure and therefore will invoke a function in the receiving object that generates the desired results. Message passing involves specifying the name of the object, the name of the function and the information to be sent.

# Classes and Objects.

* A Class is a user defined data-type which have data members and member functions.
* Data members are the data variables and member functions are the functions used to manipulate these variables and together these data members and member functions defines the properties and behavior of the objects in a Class.

When a class is defined no memory is allocated. When it is instantiated i.e an object is created only then memory is allocated.

Note that all the member functions defined inside the class definition are by default **inline**, but you can also make any non-class function inline by using keyword inline with them. Inline functions are actual functions, which are copied everywhere during compilation, like pre-processor macro, so the overhead of function calling is reduced.

# Access Modifiers in C plus plus

There are 3 access modifiers in C ++, private protected and public. If we do not specify any access modifiers for the members inside the class then by default the access modifier for the members will be **Private**.

Let us now look at each one these access modifiers in details:

* **Public**: All the class members declared under public will be available to everyone. The data members and member functions declared public can be accessed by other classes too. The public members of a class can be accessed from anywhere in the program using the direct member access operator (.) with the object of that class.

// C++ program to demonstrate public

// access modifier

#include<iostream>

using namespace std;

// class definition

class Circle

{

    public:

        double radius;

        double  compute\_area()

        {

            return 3.14\*radius\*radius;

        }

};

// main function

int main()

{

    Circle obj;

    // accessing public datamember outside class

    obj.radius = 5.5;

    cout << "Radius is:" << obj.radius << "\n";

    cout << "Area is:" << obj.compute\_area();

    return 0;

}

* **Private**: The class members declared as **private** can be accessed only by the functions inside the class. They are not allowed to be accessed directly by any object or function outside the class. Only the member functions or the [friend functions](https://www.geeksforgeeks.org/friend-class-function-cpp/) are allowed to access the private data members of a class.

// C++ program to demonstrate private

// access modifier

#include<iostream>

using namespace std;

class Circle

{

    // private data member

    private:

        double radius;

    // public member function

    public:

        double  compute\_area(double r)

        {   // member function can access private

            // data member radius

            radius = r;

            double area = 3.14\*radius\*radius;

            cout << "Radius is:" << radius << endl;

            cout << "Area is: " << area;

        }

};

// main function

int main()

{

    // creating object of the class

    Circle obj;

    // trying to access private data member

    // directly outside the class

    obj.compute\_area(1.5);

    return 0;

}

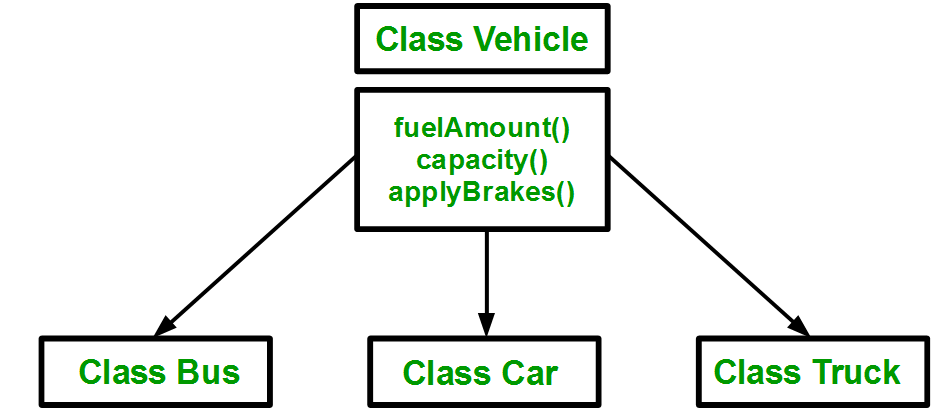
* **Protected**: Protected access modifier is similar to that of private access modifiers, the **difference is that the class member declared as Protected are inaccessible outside** the class but they can be accessed by any subclass(derived class) of that class.

|  |
| --- |
| // C++ program to demonstrate  // protected access modifier  #include <bits/stdc++.h>  using namespace std;    // base class  class Parent  {      // protected data members      protected:      int id\_protected;    };    // sub class or derived class  class Child : public Parent  {          public:      void setId(int id)      {            // Child class is able to access the inherited          // protected data members of base class            id\_protected = id;        }        void displayId()      {          cout << "id\_protected is:" << id\_protected << endl;      }  };    // main function  int main() {        Child obj1;        // member function of derived class can      // access the protected data members of base class        obj1.setId(81);      obj1.displayId();      return 0;  } |

# **Inheritance in C++**

The capability of a class to derive properties and characteristics from another class is called **Inheritance**. Inheritance is one of the most important feature of Object Oriented Programming. It avoids duplication of data and increases reusability.   
**Sub Class:** The class that inherits properties from another class is called Sub class or Derived Class.  
**Super Class:**The class whose properties are inherited by sub class is called Base Class or Super class

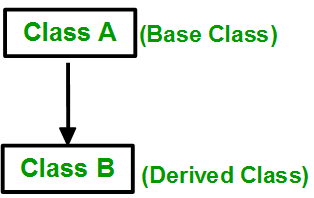
NOTE: Private member of base class will never get inherited into the sub class.



**Modes of Inheritance**

1. **Public mode**: If we derive a sub class from a public base class. Then the public member of the base class will become public in the derived class and protected members of the base class will become protected in derived class. Private members of the base class will never get inherited in sub class.
2. **Protected mode**: If we derive a sub class from a Protected base class. Then both public member and protected members of the base class will become protected in derived class. Private members of the base class will never get inherited in sub class.
3. **Private mode**: If we derive a sub class from a Private base class. Then both public member and protected members of the base class will become Private in derived class. Private members of the base class will never get inherited in sub clas

**Types of Inheritance in C++**

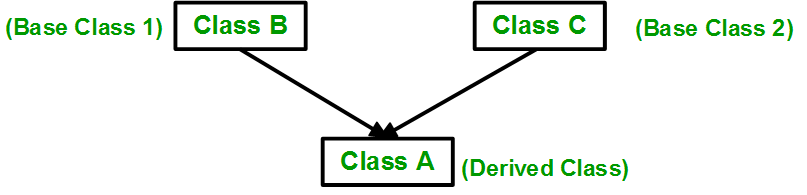
1. **Single Inheritance**: In single inheritance, a class is allowed to inherit from only one class. i.e. one sub class is inherited by one base class only.  
   **Syntax**:

class subclass\_name : access\_mode base\_class

{

//body of subclass

};

1. **Multiple Inheritance:** Multiple Inheritance is a feature of C++ where a class can inherit from more than one classes. i.e one **sub class** is inherited from more than one **base classes**.  
   **Syntax**:

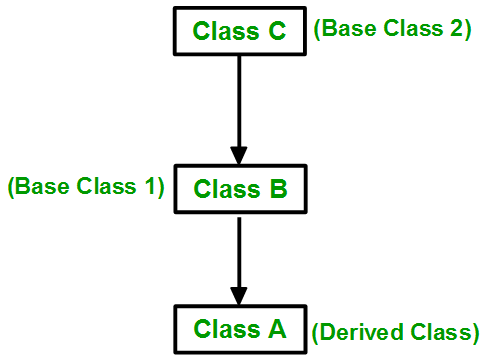
class subclass\_name : access\_mode base\_class1, access\_mode base\_class2, ....

{

//body of subclass

};

Here, the number of base classes will be separated by a comma (‘, ‘) and access mode for every base class must be specified.

1. **Multilevel Inheritance**: In this type of inheritance, a derived class is created from another derived class.

|  |
| --- |
| // C++ program to implement  // Multilevel Inheritance  #include <iostream>  using namespace std;    // base class  class Vehicle  {    public:      Vehicle()      {        cout << "This is a Vehicle" << endl;      }  };  class fourWheeler: public Vehicle  {  public:      fourWheeler()      {        cout<<"Objects with 4 wheels are vehicles"<<endl;      }  };  // sub class derived from two base classes  class Car: public fourWheeler{     public:       car()       {         cout<<"Car has 4 Wheels"<<endl;       }  };    // main function  int main()  {      //creating object of sub class will      //invoke the constructor of base classes      Car obj;      return 0;  } |

output:

This is a Vehicle

Objects with 4 wheels are vehicles

Car has 4 Wheels

1. **Hierarchical Inheritance**: In this type of inheritance, more than one sub class is inherited from a single base class. i.e. more than one derived class is created from a single base class.

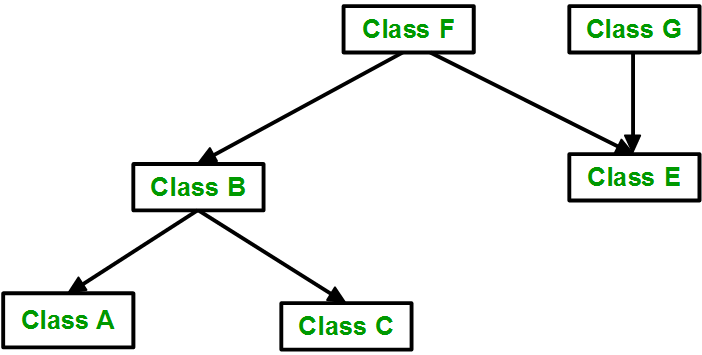
|  |
| --- |
| // C++ program to implement  // Hierarchical Inheritance  #include <iostream>  using namespace std;    // base class  class Vehicle  {    public:      Vehicle()      {        cout << "This is a Vehicle" << endl;      }  };      // first sub class  class Car: public Vehicle  {    };    // second sub class  class Bus: public Vehicle  {    };    // main function  int main()  {      // creating object of sub class will      // invoke the constructor of base class      Car obj1;      Bus obj2;      return 0;  } |

Run on IDE

Output:

This is a Vehicle

This is a Vehicle

5. **Hybrid (Virtual) Inheritance**: Hybrid Inheritance is implemented by combining more than one type of inheritance. For example: Combining Hierarchical inheritance and Multiple Inheritance.  
Below image shows the combination of hierarchical and multiple inheritance:

|  |
| --- |
| // C++ program for Hybrid Inheritance    #include <iostream>  using namespace std;    // base class  class Vehicle  {    public:      Vehicle()      {        cout << "This is a Vehicle" << endl;      }  };    //base class  class Fare  {      public:      Fare()      {          cout<<"Fare of Vehicle\n";      }  };    // first sub class  class Car: public Vehicle  {    };    // second sub class  class Bus: public Vehicle, public Fare  {    };    // main function  int main()  {      // creating object of sub class will      // invoke the constructor of base class      Bus obj2;      return 0;  } |

Run on IDE

Output:

This is a Vehicle

Fare of Vehicle

If the code is changed to class Bus: public Fare, public Vehicle, then the output will be

Fare of Vehicle

This is a Vehicle #include<iostream>

## Cool questions on Inheritance

1. using namespace std;

class P {

public:

   void print()  { cout <<" Inside P"; }

};

class Q : public P {

public:

   void print() { cout <<" Inside Q"; }

};

class R: public Q { };

int main(void)

{

  R r;

  r.print();

  return 0;

}

Output:

Inside Q.

The print function is not present in class R. So it is looked up in the inheritance hierarchy. print() is present in both classes P and Q, which of them should be called? The idea is, if there is multilevel inheritance, then function is linearly searched up in the inheritance hierarchy until a matching function is found.

1. #include<iostream>

using namespace std;

class Base {};

class Derived: public Base {};

int main()

{

    Base \*bp = new Derived;

    Derived \*dp = new Base;

}

Output: Compiler error in Derived \*dp = new Base;

# A Base class pointer/reference can point/refer to a derived class object, but the other way is not possible.

1. #include<iostream>

using namespace std;

class Base

{

public:

    void show()

    {

        cout<<" In Base ";

    }

};

class Derived: public Base

{

public:

    int x;

    void show()

    {

        cout<<"In Derived ";

    }

    Derived()

    {

        x = 10;

    }

};

int main(void)

{

    Base \*bp, b;

    Derived d;

    bp = &d;

    bp->show();

    cout << bp->x;

    return 0;

}

Output: Compiler Error in line " cout << bp->x". A b ase class pointer can point to a derived class object, but we can only access base class member or virtual functions using the base class point.

1. #include<iostream>

using namespace std;

class Base

{

public:

    int fun()  { cout << "Base::fun() called"; }

    int fun(int i)  { cout << "Base::fun(int i) called"; }

};

class Derived: public Base

{

public:

    int fun() {  cout << "Derived::fun() called"; }

};

int main()

{

    Derived d;

    d.fun(5);

    return 0;

}

Output; Compiler error. If a derived class writes its own method, then all functions of base class with same name become hidden, even if signaures of base class functions are different. In the above question, when fun() is rewritten in Derived, it hides both fun() and fun(int) of base class.

#include<iostream>

using namespace std;

class Base {

public:

    int fun()          {    cout << "Base::fun() called";     }

    int fun(int i)     {   cout << "Base::fun(int i) called";  }

};

class Derived: public Base  {

public:

    int fun()   {     cout << "Derived::fun() called";   }

};

int main()  {

    Derived d;

    d.Base::fun(5);

    return 0;

}

Output: Base::fun(int i) called

We can access base class functions using scope resolution operator even if they are made hidden by a derived class function.

1. Output of following program?

|  |
| --- |
| #include <iostream>  #include<string>  using namespace std;    class Base  {  public:      virtual string print() const      {          return "This is Base class";      }  };    class Derived : public Base  {  public:      virtual string print() const      {          return "This is Derived class";      }  };    void describe(Base p)  {      cout << p.print() << endl;  }    int main()  {      Base b;      Derived d;      describe(b);      describe(d);      return 0;  } |

Output: This is Base class

This is Base class

Note that an object of Derived is passed in describe(d), but print of Base is called. The describe function accepts a parameter of Base type. This is a typical example of object slicing, when we assign an object of derived class to an object of base type, the derived class object is sliced off and all the data members inherited from base class are copied. Object slicing should be ovoided as there may be surprising results like above. As a side note, object slicing is not possible in Java. In Java, every non-primitive variable is actually a reference.

1. #include<iostream>

using namespace std;

class Base

{

public :

int x, y;

public:

Base(int i, int j){ x = i; y = j; }

};

class Derived : public Base

{

public:

Derived(int i, int j):x(i), y(j) {}

void print() {cout << x <<" "<< y; }

};

int main(void)

{

Derived q(10, 10);

q.print();

return 0;

}

Output: Compiler error. The base class members cannot be directly assigned using initializer list. We should call the base class constructor in order to initialize base class members. Following is error free program and prints "10 10" [sourcecode language="CPP" highlight="15"] #include using namespace std; class Base { public : int x, y; public: Base(int i, int j){ x = i; y = j; } }; class Derived : public Base { public: Derived(int i, int j): Base(i, j) {} void print() {cout << x <<" "<< y; } }; int main(void) { Derived q(10, 10); q.print(); return 0; } [/sourcecode]

# **Polymorphism in C++**

**In C++ polymorphism is mainly divided into two types:**

* Compile time Polymorphism
* Runtime Polymorphism

**Compile time polymorphism**: This type of polymorphism is achieved by function overloading or operator overloading.

[**Runtime polymorphism**](https://www.geeksforgeeks.org/virtual-functions-and-runtime-polymorphism-in-c-set-1-introduction/): This type of polymorphism is achieved by Function Overriding.

* **Function overriding** on the other hand occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be **overridden**.

|  |
| --- |
| // C++ program for function overriding    #include <bits/stdc++.h>  using namespace std;    // Base class  class Parent  {      public:      void print()      {          cout << "The Parent print function was called" << endl;      }  };    // Derived class  class Child : public Parent  {      public:        // definition of a member function already present in Parent      void print()      {          cout << "The child print function was called" << endl;      }    };    //main function  int main()  {      //object of parent class      Parent obj1;        //object of child class      Child obj2 = Child();          // obj1 will call the print function in Parent      obj1.print();        // obj2 will override the print function in Parent      // and call the print function in Child      obj2.print();      return 0;  } |

Output:

The Parent print function was called

The child print function was called

Predict the output

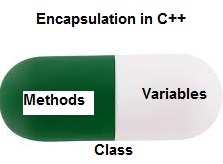
|  |
| --- |
| #include<iostream>  using namespace std;  class A  {      int i;  public:      A(int ii = 0) : i(ii) {}      void show() {  cout << i << endl;  }  };    class B  {      int x;  public:      B(int xx) : x(xx) {}      operator A() const {  return A(x); }  };    void g(A a)  {      a.show();  }    int main()  {      B b(10);      g(b);      g(20);      return 0;  } |

Output 10

20

Note that the class B has as conversion operator overloaded, so an object of B can be converted to that of A. Also, class A has a constructor which can be called with single integer argument, so an int can be converted to A.

# **Encapsulation in C++**

In normal terms **Encapsulation**is defined as wrapping up of data and information under a single unit. In Object Oriented Programming, Encapsulation is defined as binding together the data and the functions that manipulates them.  
Consider a real life example of encapsulation, in a company there are different sections like the accounts section, finance section, sales section etc. The finance section handles all the financial transactions and keep records of all the data related to finance. Similarly the sales section handles all the sales related activities and keep records of all the sales. Now there may arise a situation when for some reason an official from finance section needs all the data about sales in a particular month. In this case, he is not allowed to directly access the data of sales section. He will first have to contact some other officer in the sales section and then request him to give the particular data. This is what encapsulation is. Here the data of sales section and the employees that can manipulate them are wrapped under a single name “sales section”.  


Encapsulation also lead to data abstraction or hiding. As using encapsulation also hides the data. In the above example the data of any of the section like sales, finance or accounts is hidden from any other section.

# **Abstraction in C++**

Data abstraction is one of the most essential and important feature of object oriented programming in C++. Abstraction means displaying only essential information and hiding the details. Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.

Consider a real life example of a man driving a car. The man only knows that pressing the accelerators will increase the speed of car or applying brakes will stop the car but he does not know about how on pressing accelerator the speed is actually increasing, he does not know about the inner mechanism of

the car or the implementation of accelerator, brakes etc in the car. This is what abstraction is.  
 **Abstraction using Classes:** We can implement Abstraction in C++ using classes. Class helps us to group data members and member functions using available access specifiers. A Class can decide which data member will be visible to outside world and which is not.

**Abstraction in Header files:**One more type of abstraction in C++ can be header files. For example, consider the pow() method present in math.h header file. Whenever we need to calculate power of a number, we simply call the function pow() present in the math.h header file and pass the numbers as arguments without knowing the underlying algorithm according to which the function is actually calculating power of numbers.

Access specifiers are the main pillar of implementing abstraction in C++. We can use access specifiers to enforce restrictions on class members.

**Advantages of Data Abstraction**:

* Helps the user to avoid writing the low level code
* Avoids code duplication and increases reusability.
* Can change internal implementation of class independently without affecting the user.
* Helps to increase security of an application or program as only important details are provided to the user.

# **Friend class and function in C++**

**Friend Class** A friend class can access private and protected members of other class in which it is declared as friend. It is sometimes useful to allow a particular class to access private members of other class. For example a LinkedList class may be allowed to access private members of Node.

|  |
| --- |
| class Node  {  private:    int key;    Node \*next;    /\* Other members of Node Class \*/      friend class LinkedList; // Now class  LinkedList can                             // access private members of Node  } |

**Friend Function** Like friend class, a friend function can be given special grant to access private and protected members. A friend function can be:  
a) A method of another class  
b) A global function

class Node

{

private:

  int key;

  Node \*next;

  /\* Other members of Node Class \*/

  friend int LinkedList::search(); // Only search() of linkedList

                                  // can access internal members

};

Following are some important points about friend functions and classes:  
**1)** Friends should be used only for limited purpose. too many functions or external classes are declared as friends of a class with protected or private data, it lessens the value of encapsulation of separate classes in object-oriented programming.

**2)** Friendship is not mutual. If a class A is friend of B, then B doesn’t become friend of A automatically.

**3)** Friendship is not inherited (See [this](https://www.geeksforgeeks.org/g-fact-34/)for more details)

**4)** The concept of friends is not there in Java.

# **Local Classes in C++**

A class declared inside a function becomes local to that function and is called Local Class in C++. For example, in the following program, Test is a local class in fun().

|  |
| --- |
| #include<iostream>  using namespace std;  void fun()  {        class Test  // local to fun        {          /\* members of Test class \*/        };  }    int main()  {      return 0;  } |

<https://www.geeksforgeeks.org/local-class-in-c/>

**Following are some interesting facts about local classes.**  
***1)*** A local class type name can only be used in the enclosing function.

***2)*** All the methods of Local classes must be defined inside the class only.

***3)*** A Local class cannot contain static data members. It may contain static functions though

***4)*** Member methods of local class can only access static and enum variables of the enclosing function. Non-static variables of the enclosing function are not accessible inside local classes.

***5)*** Local classes can access global types, variables and functions. Also, local classes can access other local classes of same function..

# **Nested Classes in C++**

A nested class is a class which is declared in another enclosing class. A nested class is a member and as such has the same access rights as any other member. The members of an enclosing class have no special access to members of a nested class; the usual access rules shall be obeyed. Also, except by using explicit pointers, references, and object names, declarations in a nested class shall not use nonstatic data members or non-static member functions from the enclosing class. This restriction applies in all constructs including the operands of the sizeof operator.

References: <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1905.pdf>

For example, program 1 compiles without any error and program 2 fails in compilation.

**Program 1**

|  |
| --- |
| #include<iostream>    using namespace std;     /\* start of Enclosing class declaration \*/  class Enclosing {       int x;       /\* start of Nested class declaration \*/     class Nested {        int y;        void NestedFun(Enclosing \*e) {          cout<<e->x;  // works fine: nested class can access                       // private members of Enclosing class        }     }; // declaration Nested class ends here  }; // declaration Enclosing class ends here    int main()  {    } |

Run on IDE

**Program 2**

|  |
| --- |
| #include<iostream>    using namespace std;     /\* start of Enclosing class declaration \*/  class Enclosing {       int x;       /\* start of Nested class declaration \*/     class Nested {        int y;     }; // declaration Nested class ends here       void EnclosingFun(Nested \*n) {          cout<<n->y;  // Compiler Error: y is private in Nested     }  }; // declaration Enclosing class ends here    int main()  {    } |

# **Multiple Inheritance in C++**

Multiple Inheritance is a feature of C++ where a class can inherit from more than one classes.

The constructors of inherited classes are called in the same order in which they are inherited. For example, in the following program, B’s constructor is called before A’s constructor.

|  |
| --- |
| #include<iostream>  using namespace std;    class A  {  public:    A()  { cout << "A's constructor called" << endl; }  };    class B  {  public:    B()  { cout << "B's constructor called" << endl; }  };    class C: public B, public A  // Note the order  {  public:    C()  { cout << "C's constructor called" << endl; }  };    int main()  {      C c;      return 0;  } |

Run on IDE

Output:

B's constructor called

A's constructor called

C's constructor called

The destructors are called in reverse order of constructors.

**The diamond problem**  
The diamond problem occurs when two superclasses of a class have a common base class. For example, in the following diagram, the TA class gets two copies of all attributes of Person class, this causes ambiguities.

https://cdncontribute.geeksforgeeks.org/wp-content/uploads/diamondproblem.png

For example, consider the following program.

|  |
| --- |
| #include<iostream>  using namespace std;  class Person {     // Data members of person  public:      Person(int x)  { cout << "Person::Person(int ) called" << endl;   }  };    class Faculty : public Person {     // data members of Faculty  public:      Faculty(int x):Person(x)   {         cout<<"Faculty::Faculty(int ) called"<< endl;      }  };    class Student : public Person {     // data members of Student  public:      Student(int x):Person(x) {          cout<<"Student::Student(int ) called"<< endl;      }  };    class TA : public Faculty, public Student  {  public:      TA(int x):Student(x), Faculty(x)   {          cout<<"TA::TA(int ) called"<< endl;      }  };    int main()  {      TA ta1(30);  } |

Run on IDE

Person::Person(int ) called

Faculty::Faculty(int ) called

Person::Person(int ) called

Student::Student(int ) called

TA::TA(int ) called

In the above program, constructor of ‘Person’ is called two times. Destructor of ‘Person’ will also be called two times when object ‘ta1’ is destructed. So object ‘ta1’ has two copies of all members of ‘Person’, this causes ambiguities. The solution to this problem is ‘virtual’ keyword. We make the classes ‘Faculty’ and ‘Student’ as virtual base classes to avoid two copies of ‘Person’ in ‘TA’ class. For example, consider the following program.

|  |
| --- |
| #include<iostream>  using namespace std;  class Person {  public:      Person(int x)  { cout << "Person::Person(int ) called" << endl;   }      Person()     { cout << "Person::Person() called" << endl;   }  };    class Faculty : virtual public Person {  public:      Faculty(int x):Person(x)   {         cout<<"Faculty::Faculty(int ) called"<< endl;      }  };    class Student : virtual public Person {  public:      Student(int x):Person(x) {          cout<<"Student::Student(int ) called"<< endl;      }  };    class TA : public Faculty, public Student  {  public:      TA(int x):Student(x), Faculty(x)   {          cout<<"TA::TA(int ) called"<< endl;      }  };    int main()  {      TA ta1(30);  } |

Run on IDE

Output:

Person::Person() called

Faculty::Faculty(int ) called

Student::Student(int ) called

TA::TA(int ) called

In the above program, constructor of ‘Person’ is called once. One important thing to note in the above output is, the default constructor of ‘Person’ is called. When we use ‘virtual’ keyword, the default constructor of grandparent class is called by default even if the parent classes explicitly call parameterized constructor.

**How to call the parameterized constructor of the ‘Person’ class?**The constructor has to be called in ‘TA’ class. For example, see the following program.

|  |
| --- |
| #include<iostream>  using namespace std;  class Person {  public:      Person(int x)  { cout << "Person::Person(int ) called" << endl;   }      Person()     { cout << "Person::Person() called" << endl;   }  };    class Faculty : virtual public Person {  public:      Faculty(int x):Person(x)   {         cout<<"Faculty::Faculty(int ) called"<< endl;      }  };    class Student : virtual public Person {  public:      Student(int x):Person(x) {          cout<<"Student::Student(int ) called"<< endl;      }  };    class TA : public Faculty, public Student  {  public:      TA(int x):Student(x), Faculty(x), Person(x)   {          cout<<"TA::TA(int ) called"<< endl;      }  };    int main()  {      TA ta1(30);  } |

Run on IDE

Output:

Person::Person(int ) called

Faculty::Faculty(int ) called

Student::Student(int ) called

TA::TA(int ) called

In general, it is not allowed to call the grandparent’s constructor directly, it has to be called through parent class. It is allowed only when ‘virtual’ keyword is used.

# **Some interesting facts about static member functions in C++**

**1)** static member functions do not have [this pointer](http://publib.boulder.ibm.com/infocenter/comphelp/v8v101/index.jsp?topic=%2Fcom.ibm.xlcpp8a.doc%2Flanguage%2Fref%2Fcplr035.htm).  
For example following program fails in compilation with error “`this’ is unavailable for static member functions “

|  |
| --- |
| #include<iostream>  class Test {     static Test \* fun() {       return this; // compiler error     }  };  int main()  {     getchar();     return 0;  } |

**2)**A static member function cannot be virtual (See [this](https://www.geeksforgeeks.org/?p=8413)G-Fact)

**3)**Member function declarations with the same name and the name parameter-type-list cannot be overloaded if any of them is a static member function declaration.  
For example, following program fails in compilation with error “‘void Test::fun()’ and `static void Test::fun()’ cannot be overloaded ”

|  |
| --- |
| #include<iostream>  class Test {     static void fun() {}     void fun() {} // compiler error  };    int main()  {     getchar();     return 0;  } |

**4)** A static member function can not be declared const, volatile, or const volatile.  
For example, following program fails in compilation with error “static member function `static void Test::fun()’ cannot have `const’ method qualifier ”

|  |
| --- |
| #include<iostream>  class Test {     static void fun() const { // compiler error       return;     }  };    int main()  {     getchar();     return 0;  }  When you apply the const qualifier to a nonstatic member function, it affects the this pointer. For a const-qualified member function of class C, the this pointer is of type C const\*, whereas for a member function that is not const-qualified, the this pointer is of type C\*.  A static member function does not have a this pointer (such a function is not called on a particular instance of a class), so const qualification of a static member function doesn't make any sense. |

References:  
<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1905.pdf>

# **Static data members in C++**

Static data members can be accessed by non static functions. Static functions can access only static members (i.e only static data and static member functions)

Predict the output of following C++ program:

|  |
| --- |
| #include <iostream>  using namespace std;  class A  {  public:      A() { cout << "A's Constructor Called " << endl;  }  };    class B  {      static A a;  public:      B() { cout << "B's Constructor Called " << endl; }  };  int main()  {      B b;      return 0;  } |

Output:

B's Constructor Called

The above program calls only B’s constructor, it doesn’t call A’s constructor. The reason for this is simple, static members are only declared in class declaration, not defined. They must be explicitly defined outside the class using scope resolution operator.  
If we try to access static member ‘a’ without explicit definition of it, we will get compilation error. For example, following program fails in compilation.

|  |
| --- |
| #include <iostream>  using namespace std;    class A  {      int x;  public:      A() { cout << "A's constructor called " << endl;  }  };    class B  {      static A a;  public:      B() { cout << "B's constructor called " << endl; }      static A getA() { return a; }  };    int main()  {      B b;      A a = b.getA();      return 0;  } |

Run on IDE

Output:

Compiler Error: undefined reference to `B::a'

If we add definition of a, the program will works fine and will call A’s constructor. See the following program.

|  |
| --- |
| #include <iostream>  using namespace std;    class A  {      int x;  public:      A() { cout << "A's constructor called " << endl;  }  };    class B  {      static A a;  public:      B() { cout << "B's constructor called " << endl; }      static A getA() { return a; }  };    A B::a;  // definition of a    int main()  {      B b1, b2, b3;      A a = b1.getA();        return 0;  } |

Run on IDE

Output:

A's constructor called

B's constructor called

B's constructor called

B's constructor called

Note that the above program calls B’s constructor 3 times for 3 objects (b1, b2 and b3), but calls A’s constructor only once. The reason is, static members are shared among all objects. That is why they are also known as class members or class fields. Also, static members can be accessed without any object, see the below program where static member ‘a’ is accessed without any object.

|  |
| --- |
| #include <iostream>  using namespace std;    class A  {      int x;  public:      A() { cout << "A's constructor called " << endl;  }  };    class B  {      static A a;  public:      B() { cout << "B's constructor called " << endl; }      static A getA() { return a; }  };    A B::a;  // definition of a    int main()  {      // static member 'a' is accessed without any object of B      A a = B::getA();        return 0;  } |

Output:

A's constructor called

# **Can a C++ class have an object of self type?**

A class declaration can contain static object of self type, it can also have pointer to self type, but it cannot have a non-static object of self type.

For example, following program works fine.

|  |
| --- |
| // A class can have a static member of self type  #include<iostream>    using namespace std;    class Test {    static Test self;  // works fine      /\* other stuff in class\*/    };    int main()  {    Test t;    getchar();    return 0;  } |

And following program also works fine.

|  |
| --- |
| // A class can have a pointer to self type  #include<iostream>  using namespace std;    class Test {    Test \* self; //works fine      /\* other stuff in class\*/  };    int main()  {    Test t;    getchar();    return 0;  } |

But following program generates compilation error “field `self’ has incomplete type”

|  |
| --- |
| // A class cannot have non-static object(s) of self type.  #include<iostream>  using namespace std;    class Test {    Test self; // Error    /\* other stuff in class\*/  };    int main()  {    Test t;    getchar();    return 0;  } |

If a non-static object is member then declaration of class is incomplete and compiler has no way to find out size of the objects of the class.  
Static variables do not contribute to the size of objects. So no problem in calculating size with static variables of self type.  
For a compiler, all pointers have a fixed size irrespective of the data type they are pointing to, so no problem with this also.

# **Why is the size of an empty class not zero in C++?**

Predict the output of following program?

|  |
| --- |
| #include<iostream>  using namespace std;    class Empty {};    int main()  {    cout << sizeof(Empty);    return 0;  } |

Run on IDE

Output:

1

Size of an empty class is not zero. It is 1 byte generally. It is nonzero to ensure that the two different objects will have different addresses. See the following example.

|  |
| --- |
| #include<iostream>  using namespace std;    class Empty { };    int main()  {      Empty a, b;        if (&a == &b)        cout << "impossible " << endl;      else        cout << "Fine " << endl;       return 0;  } |

Run on IDE

Output:

Fine

For the same reason (different objects should have different addresses), “new” always returns pointers to distinct objects. See the following example.

|  |
| --- |
| #include<iostream>  using namespace std;    class Empty { };    int main()  {      Empty\* p1 = new Empty;      Empty\* p2 = new Empty;        if (p1 == p2)          cout << "impossible " << endl;      else          cout << "Fine " << endl;        return 0;  } |

Output: Fine

Now guess the output of following program (This is tricky)

|  |
| --- |
| #include<iostream>  using namespace std;    class Empty { };    class Derived: Empty { int a; };    int main()  {      cout << sizeof(Derived);      return 0;  } |

Output (with GCC compiler. See [this](http://ideone.com/JFoX8)):

4

Note that the output is not greater than 4. There is an interesting rule that says that an empty base class need not be represented by a separate byte. So compilers are free to make optimization in case of empty base classes. As an excercise, try the following program on your compiler.

|  |
| --- |
| #include <iostream>  using namespace std;    class Empty  {};  class Derived1 : public Empty  {};  class Derived2 : virtual public Empty  {};  class Derived3 : public Empty  {      char c;  };  class Derived4 : virtual public Empty  {      char c;  };  class Dummy  {      char c;  };    int main()  {      cout << "sizeof(Empty) " << sizeof(Empty) << endl;      cout << "sizeof(Derived1) " << sizeof(Derived1) << endl;      cout << "sizeof(Derived2) " << sizeof(Derived2) << endl;      cout << "sizeof(Derived3) " << sizeof(Derived3) << endl;      cout << "sizeof(Derived4) " << sizeof(Derived4) << endl;      cout << "sizeof(Dummy) " << sizeof(Dummy) << endl;        return 0;  }  Output:  sizeof(Empty) 1  sizeof(Derived1) 1  sizeof(Derived2) 8  sizeof(Derived3) 1  sizeof(Derived4) 16  sizeof(Dummy) 1 |

**Source:**  
<http://www2.research.att.com/~bs/bs_faq2.html>

# **Simulating final class in C++**

Ever wondered how can you design a class in C++ which can’t be inherited. Java and C# programming languages have this feature built-in. You can use [final](http://en.wikipedia.org/wiki/Final_%28Java%29#Final_classes)keyword in java, [sealed](http://msdn.microsoft.com/en-us/library/88c54tsw%28v=vs.71%29.aspx)in C# to make a class non-extendable.

Below is a mechanism using which we can achieve the same behavior in C++. It makes use of private constructor, virtual inheritance and friend class.

In the following code, we make the Final class non-inheritable. When a class Derived tries to inherit from it, we get compilation error.

An extra class MakeFinal (whose default constructor is private) is used for our purpose. Constructor of Final can call private constructor of MakeFinal as Final is a friend of MakeFinal .

Note that MakeFinal is also a virtual base class. The reason for this is to call the constructor of MakeFinal through the constructor of Derived, not Final (The constructor of a virtual base class is not called by the class that inherits from it, instead the constructor is called by the constructor of the concrete class).

|  |
| --- |
| /\* A program with compilation error to demonstrate that Final class cannot     be inherited \*/  #include<iostream>  using namespace std;    class Final;  // The class to be made final  class MakeFinal // used to make the Final class final  {  private:      MakeFinal() { cout << "MakFinal constructor" << endl; }  friend class Final;  };    class Final : virtual MakeFinal  {  public:      Final() { cout << "Final constructor" << endl; }  };    class Derived : public Final // Compiler error  {  public:      Derived() { cout << "Derived constructor" << endl; }  };    int main(int argc, char \*argv[])  {      Derived d;      return 0;  } |

Run on IDE

Output: Compiler Error

In constructor 'Derived::Derived()':

error: 'MakeFinal::MakeFinal()' is private

In the above example, Derived‘s constructor directly invokes MakeFinal’s constructor, and the constructor of MakeFinal is private, therefore we get the compilation error.

You can create the object of Final class as it is friend class of MakeFinal and has access to its constructor. For example, the following program works fine.

|  |
| --- |
| /\* A program without any compilation error to demonstrate that instances of     the Final class can be created \*/  #include<iostream>  using namespace std;    class Final;    class MakeFinal  {  private:      MakeFinal() { cout << "MakeFinal constructor" << endl; }      friend class Final;  };    class Final : virtual MakeFinal  {  public:      Final() { cout << "Final constructor" << endl; }  };    int main(int argc, char \*argv[])  {      Final f;      return 0;  } |

Run on IDE

Output: Compiles and runs fine

MakeFinal constructor

Final constructor

# **Structure vs class in C++**

In C++, a structure is same as class except the following differences:

1) Members of a class are private by default and members of struct are public by default.  
For example program 1 fails in compilation and program 2 works fine.

|  |
| --- |
| // Program 1  #include <stdio.h>    class Test {      int x; // x is private  };  int main()  {    Test t;    t.x = 20; // compiler error because x is private    getchar();    return 0;  } |

Run on IDE

|  |
| --- |
| // Program 2  #include <stdio.h>    struct Test {      int x; // x is public  };  int main()  {    Test t;    t.x = 20; // works fine because x is public    getchar();    return 0;  } |

Run on IDE

2) When deriving a struct from a class/struct, default access-specifier for a base class/struct is public. And when deriving a class, default access specifier is private.  
For example program 3 fails in compilation and program 4 works fine.

|  |
| --- |
| // Program 3  #include <stdio.h>    class Base {  public:      int x;  };  class Derived : Base { }; // is equilalent to class Derived : private Base {}  int main()  {    Derived d;    d.x = 20; // compiler error becuase inheritance is private    getchar();    return 0;  } |

Run on IDE

|  |
| --- |
| // Program 4  #include <stdio.h>    class Base {  public:      int x;  };    struct Derived : Base { }; // is equilalent to struct Derived : public Base {}    int main()  {    Derived d;    d.x = 20; // works fine becuase inheritance is public    getchar();    return 0;  } |

# **Difference between C structures and C++ structures**

In C++, struct and class are exactly the same things, except for that struct defaults to public visibility and class defaults to private visibility.  
**Some important differences between the C and C++ structures:**

1. **Member functions inside structure**: Structures in C cannot have member functions inside structure but Structures in C++ can have member functions along with data members.
2. **Direct Initialization:** We cannot directly initialize structure data members in C but we can do it in C++.
   * C

#include<iostream>

struct Record

{

    int x = 7; //Compiler error

};

* + C++

|  |
| --- |
| // CPP program to initialize data member in c++  #include<iostream>  using namespace std;  struct Record  {      int x = 7;  }; |

1. **Using struct keyword:** In C, we need to use struct to declare a struct variable. In C++, struct is not necessary. For example, let there be a structure for Record. In C, we must use “struct Record” for Record variables. In C++, we need not use struct and using ‘Record‘ only would work.
2. **Static Members:** C structures cannot have static members but is allowed in C++.
   * C
   * C++

|  |
| --- |
| // C program with structure static member  struct Record  {      static int x;  };    // Driver program  int main()  {      return 0;  }  /\* 6:5: error: expected specifier-qualifier-list     before 'static'       static int x;       ^\*/ |

This will generate an error in C but no error in C++.

1. **sizeof operator:**This operator will generate **0** for an empty structure in C whereas **1** for an empty structure in C++.

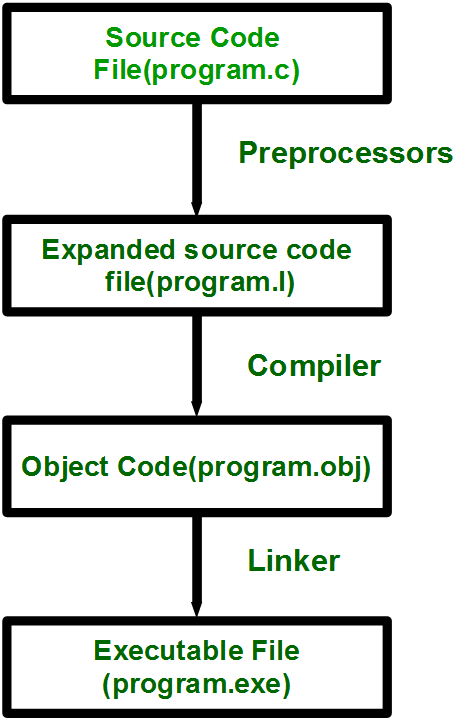
|  |
| --- |
| // C program to illustrate empty structure  #include<stdio.h>  //empty structure  struct Record  {  };  //Driver program  int main()  {      struct Record s;      printf("%d\n",sizeof(s));      return 0;  } |

Output in C: 0

Output in C++: 1

1. **Data Hiding:** C structures does not allow concept of Data hiding but is permitted in C++ as C++ is an object oriented language whereas C is not.
2. **Access Modifiers:** C structures does not have access modifiers as these modifiers are not suppoted by the language. C++ structures can have this concept as it is inbuilt in the language.

# **C/C++ Preprocessors**

As the name suggests Preprocessors are programs that processes our source code before compilation. There are a number of steps involved between writing a program and executing a program in C / C++. Let us have a look at these steps before we actually start learning about Preprocessors.  
[](http://cdncontribute.geeksforgeeks.org/wp-content/uploads/preprocessors.png)

You can see the intermediate steps in the above diagram. The source code written by programmers is stored in the file program.c. This file is then processed by preprocessors and an expanded source code file is generated named program. This expanded file is compiled by the compiler and an object code file is generated named program.obj . Finally the linker links this object code file to the object code of the library functions to generate the executable file program.exe .

Preprocessor programs provides preprocessors directives which tell the compiler to preprocess the source code before compiling. All of these preprocessor directive begins with a ‘#’ (hash) symbol. This (‘#’) symbol at the beginning of a statement in a C/C++ program indicates that it is a pre-processor directive. We can place these pre processor directives anywhere in our program. Examples of some preprocessor directives are: #include , #define, #ifndef etc. In C we use something like #define SIDE 6. Macros are nothing but preprocessor text replacements of the word side with 6. Not a great idea. But in C++ it must be written as const int side = 6. This is a definitin and not a preprocessor direcive. So it undergoes compiler check as per language rules. Also,the keyword inline replaces Macros with code.

**There are 4 main types of preprocessor directives:**

1. Macros
2. File Inclusion
3. Conditional Compilation
4. Other directives

Let us now learn about each of these directives in details.

* **Macros**: Macros are piece of code in a program which is given some name. Whenever this name is encountered by the compiler the compiler replaces the name with the actual piece of code. The ‘#define’ directive is used to define a macro. Let us now understand macro definition with the help of a program:

|  |
| --- |
| #include<iostream>  //macro definition  #define LIMIT 5  int main()  {      for(int i=0; i < LIMIT; i++)      {          std::cout<<i<<"\n";      }        return 0;  } |

Output:

0

1

2

3

4

* In the above program, when the compiler executes the word LIMIT it replaces it with 5. The word ‘LIMIT’ in macro definition is called macro template and ‘5’ is macro expansion.  
  **Note**: There is no semi-colon(‘;’) at the end of macro definition. Macro definitions do not need a semi-colon to end.
* **Macros with arguments**: We can also pass arguments to macros. Macros defined with arguments works similarly as functions. Let us understand this with a program:

|  |
| --- |
| #include<iostream>  //macro with parameter  #define AREA(l,b) (l\*b)  int main()  {      int l1 = 10, l2=5, area;        area = AREA(l1,l2);        std::cout<<"Area of rectangle is: "<<area;        return 0;  } |

Output:

Area of rectangle is: 50

We can see from the above program that whenever the compiler finds AREA(l,b) in the program it replaces it with the statement (l\*b) . Not only this, the values passed to the macro template AREA(l,b) will also be replaced in the statement (l\*b). Therefore AREA(10,5) will be equal to 10\*5.

**File Inclusion**: This type of preprocessor directive tells the compiler to include a file in the source code program. There are some header files with their name starting with 'c', for eg #include<cstdlib>. This means that use the standard library from C programming language. There are two types of files which can be included by the user in the program: :

* 1. **Header File or Standard files**: These files contains definition of pre-defined functions like printf(), scanf() etc. These files must be included for working with these functions. Different function are declared in different header files. For example standard I/O funuctions are in ‘iostream’ file whereas functions which perform string operations are in ‘string’ file.  
     **Syntax**:

#include< file\_name >

where file\_name is the name of file to be included. The ‘<‘ and ‘>’ brackets tells the compiler to look for the file in standard directory.

* 1. **user defined files**: When a program becomes very large, it is good practice to divide it into smaller files and include whenever needed. These types of files are user defined files. These files can be included as:

#include"filename"

* **Conditional Compilation**: Conditional Compilation directives are type of directives which helps to compile a specific portion of the program or to skip compilation of some specific part of the program based on some conditions. This can be done with the help of two preprocessing commands ‘**ifdef**‘ and ‘**endif**‘.  
  **Syntax**:

ifdef macro\_name

statement1;

statement2;

statement3;

.

.

.

statementN;

endif

If the macro with name as ‘macroname‘ is defined then the block of statements will execute normally but if it is not defined, the compiler will simply skip this block of statements.

* **Other directives**: Apart from the above directives there are two more directives which are not commonly used. These are:
  1. **#undef Directive**: The #undef directive is used to undefine an existing macro. This directive works as:

#undef LIMIT

Using this statement will undefine the existing macro LIMIT. After this statement every “#ifdef LIMIT” statement will evaluate to false.

* 1. **#pragma Directive**: This directive is a special purpose directive and is used to turn on or off some features. This type of directives are compiler-specific i.e., they vary from compiler to compiler. Some of the #pragma directives are discussed below:
     + **#pragma startup** and **#pragma exit**: These directives helps us to specify the functions that are needed to run before program startup( before the control passes to main()) and just before program exit (just before the control returns from main()).  
       **Note:** Below program will not work with GCC compilers.  
       Look at the below program:

|  |
| --- |
| #include<stdio.h>    void func1();  void func2();    #pragma startup func1  #pragma exit func2    void func1()  {      printf("Inside func1()\n");  }    void func2()  {      printf("Inside func2()\n");  }    int main()  {      printf("Inside main()\n");        return 0;  } |

Run on IDE

Output:

Inside func1()

Inside main()

Inside func2()

The above code will produce the output as given below when run on GCC compilers:

Inside main()

This happens because GCC does not supports #pragma startup or exit. However you can use the below code for a similar output on GCC compilers.

|  |
| --- |
| #include<stdio.h>  void func1();  void func2();    void \_\_attribute\_\_((constructor)) func1();  void \_\_attribute\_\_((destructor)) func2();    void func1()  {      printf("Inside func1()\n");  }    void func2()  {      printf("Inside func2()\n");  }    int main()  {      printf("Inside main()\n");        return 0;  } |

* + - **#pragma warn Directive:** This directive is used to hide the warning message which are displayed during compilation.  
      We can hide the warnings as shown below:
      * **#pragma warn -rvl**: This directive hides those warning which are raised when a function which is supposed to return a value does not returns a value.
      * **#pragma warn -par**: This directive hides those warning which are raised when a function does not uses the parameters passed to it.
      * **#pragma warn -rch**: This directive hides those warning which are raised when a code is unreachable. For example: any code written after the returnstatement in a function is unreachable.

# **What happen when we exceed valid range of built-in data types in C++?**

Consider the below programs.  
1) Program to show what happens when we cross range of ‘char’ :

|  |
| --- |
| // C++ program to demonstrate the problem with 'char'  #include <iostream>  using namespace std;    int main()  {      for (char a = 0; a <= 225; a++)          cout << a;      return 0;  } |

Run on IDE

Will this code print ‘a’ till it becomes 226? Well the answer is indefinite loop, because here ‘a’ is declared as a char and its valid range is -128 to +127. When ‘a’ become 128 through a++, the range is exceeded and as a result the first number from negative side of the range (i.e. -128) gets assigned to a. Hence the condition “a <= 225” is satisfied and control remains within the loop.

2) Program to show what happens when we cross range of ‘bool’ :

|  |
| --- |
| // C++ program to demonstrate  // the problem with 'bool'  #include <iostream>  using namespace std;   int main()  {      // declaring Boolean variable with true value      bool a = true;        for (a = 1; a <= 5; a++)          cout << a;      return 0;  } |

Run on IDE

This code will print ‘1’ infinite time because here ‘a’ is declared as ‘bool’ and it’s valid range is 0 to 1. And for a Boolean variable anything else than 0 is 1 (or true). When ‘a’ tries to become 2 (through a++), 1 gets assigned to ‘a’. The condition a<=5 is satisfied and the control remains with in the loop. See [this](https://www.geeksforgeeks.org/bool-data-type-in-c/)for Bool data type.

3) Program to show what happens when we cross range of ‘short’ :  
Note that short is short for short int. They are synonymous. short, short int, signed short, and signed short int are all the same data-type.

|  |
| --- |
| // C++ program to demonstrate the problem with 'short'  #include <iostream>  using namespace std;  int main()  {      // declaring short variable      short a;        for (a = 32767; a < 32770; a++)          cout << a << "\n";        return 0;  } |

Run on IDE

Will this code print ‘a’ till it becomes 32770? Well the answer is indefinite loop, because here ‘a’ is declared as a short and its valid range is -32767 to +32767. When ‘a’ tries to become 32768 through a++, the range is exceeded and as a result the first number from negative side of the range(i.e. -32767) gets assigned to a. Hence the condition “a < 32770" is satisfied and control remains within the loop.

4) Program to show what happens when we cross range of ‘unsigned short’ :

|  |
| --- |
| // C++ program to demonstrate the problem with 'unsigned short'  #include <iostream>  using namespace std;    int main()  {      unsigned short a;        for (a = 65532; a < 65536; a++)          cout << a << "\n";        return 0;  } |

Will this code print ‘a’ till it becomes 65536? Well the answer is indefinite loop, because here ‘a’ is declared as a short and its valid range is 0 to +65535. When ‘a’ tries to become 65536 through a++, the range is exceeded and as a result the first number from the range(i.e. 0) gets assigned to a. Hence the condition “a < 65536” is satisfied and control remains within the loop.

# **C++ string class and its applications**

In C++ we can store string by one of the two ways –

1. [C style strings](https://www.geeksforgeeks.org/storage-for-strings-in-c/)
2. string class (discussed in this post)

In this post, second method is discussed. string class is part of C++ library that supports a lot much functionality over C style strings.  
C++ string class internally uses char array to store character but all memory management, allocation and null termination is handled by string class itself that is why it is easy to use. The length of c++ string can be changed at runtime because of dynamic allocation of memory similar to vectors. As string class is a container class, we can iterate over all its characters using an iterator similar to other containers like vector, set and maps, but generally we use a simple for loop for iterating over the characters and index them using [] operator.  
C++ string class has a lot of functions to handle string easily. Most useful of them are demonstrated in below code.

|  |
| --- |
| // C++ program to demonstrate various function string class  #include <bits/stdc++.h>  using namespace std;    int main()  {      // various constructor of string class        // initialization by raw string      string str1("first string");        // initialization by another string      string str2(str1);        // initialization by character with number of occurence      string str3(5, '#');        // initialization by part of another string      string str4(str1, 6, 6); // from 6th index (second parameter)                               // 6 characters (third parameter)        // initialization by part of another string : iteartor version      string str5(str2.begin(), str2.begin() + 5);        cout << str1 << endl;      cout << str2 << endl;      cout << str3 << endl;      cout << str4 << endl;      cout << str5 << endl;        //  assignment operator      string str6 = str4;        // clear function deletes all character from string      str4.clear();        // both size() and length() return length of string and they work as synonyms      int len = str6.length(); // Same as "len = str6.size();"        cout << "Length of string is : " << len << endl;        // a particular character can be accessed using at / [] operator      char ch = str6.at(2); //  Same as "ch = str6[2];"        cout << "third character of string is : " << ch << endl;        // front return first character and back returns last charcter of string        char ch\_f = str6.front();  // Same as "ch\_f = str6[0];"      char ch\_b = str6.back();   // Same as below                                 // "ch\_b = str6[str6.length() - 1];"        cout << "First char is : " << ch\_f << ", Last char is : "           << ch\_b << endl;        // c\_str returns null terminated char array version of string      const char\* charstr = str6.c\_str();      printf("%s\n", charstr);        // append add the argument string at the end      str6.append(" extension");      //  same as str6 += " extension"        // another version of appends, which appends part of other      // string      str4.append(str6, 0, 6);  // at 0th position 6 character        cout << str6 << endl;      cout << str4 << endl;        //  find returns index where pattern is found.      //  If pattern is not there it returns predefined      //  constant npos whose value is -1        if (str6.find(str4) != string::npos)          cout << "str4 found in str6 at " << str6.find(str4)               << " pos" << endl;      else          cout << "str4 not found in str6" << endl;        //  substr(a, b) function returns a substring of b length      //  starting from index a      cout << str6.substr(7, 3) << endl;        //  if second argument is not passed, string till end is      // taken as substring      cout << str6.substr(7) << endl;        //  erase(a, b) deletes b character at index a      str6.erase(7, 4);      cout << str6 << endl;        //  iterator version of erase      str6.erase(str6.begin() + 5, str6.end() - 3);      cout << str6 << endl;        str6 = "This is a examples";        //  replace(a, b, str)  replaces b character from a index by str      str6.replace(2, 7, "ese are test");        cout << str6 << endl;        return 0;  } |

Run on IDE

Output :

first string

first string

#####

string

first

Length of string is : 6

third character of string is : r

First char is : s, Last char is : g

string

string extension

string

str4 found in str6 at 0 pos

ext

extension

string nsion

strinion

These are test examples

As seen in above code, we can get length of string by size() as well as length() but length() is preferred for strings. We can concat a string to another string by += or by append(), but += is slightly slower than append() because each time + is called a new string (creation of new buffer) is made which is returned that is a bit overhead in case of many append operation.

**Applications :**  
On basis of above string function some application are written below :

|  |
| --- |
| // C++ program to demonstrate uses of some string function  #include <bits/stdc++.h>  using namespace std;    // this function returns floating point part of a number-string  string returnFloatingPart(string str)  {      int pos = str.find(".");      if (pos == string::npos)          return "";      else          return str.substr(pos + 1);  }    // this function checks whether string contains all digit or not  bool containsOnlyDigit(string str)  {      int l = str.length();      for (int i = 0; i < l; i++)      {          if (str.at(i) < '0' || str.at(i) > '9')              return false;      }      //  if we reach here all character are digits      return true;  }    // this function replaces all single space by %20  // Used in URLS  string replaceBlankWith20(string str)  {      string replaceby = "%20";      int n = 0;        // loop till all space are replaced      while ((n = str.find(" ", n)) != string::npos )      {          str.replace(n, 1, replaceby);          n += replaceby.length();      }      return str;  }    // driver function to check above methods  int main()  {      string fnum = "23.342";      cout << "Floating part is : " << returnFloatingPart(fnum)           << endl;        string num = "3452";      if (containsOnlyDigit(num))          cout << "string contains only digit" << endl;        string urlex = "google com in";      cout << replaceBlankWith20(urlex) << endl;        return 0;  } |

Run on IDE

Output :

Floating part is : 342

string contains only digit

google%20com%20in

# Pointers in C and C++ | Set 1 (Arithmetic and Array**)**

**Pointer Expressions and Pointer Arithmetic**

A limited set of arithmetic operations can be performed on pointers. A pointer may be:

* incremented ( ++ )
* decremented ( — )
* an integer may be added to a pointer ( + or += )
* an integer may be subtracted from a pointer ( – or -= )

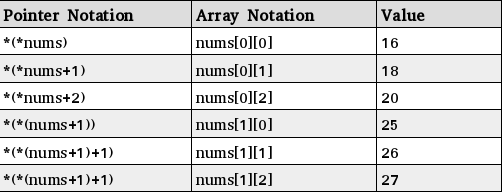
Pointer arithmetic is meaningless unless performed on an array.  
Note : Pointers contain addresses. Adding two addresses makes no sense, because there is no idea what it would point to. Subtracting two addresses lets you compute the offset between these two addresses.

**Pointers and Multidimensional Arrays**

Consider pointer notation for the two-dimensional numeric arrays. consider the following declaration

int nums[2][3] = { {16, 18, 20}, {25, 26, 27} };

**In general, nums[i][j] is equivalent to \*(\*(nums+i)+j)**



# **Multidimensional Pointer Arithmetic in C/C++**

In C/C++, arrays and pointers have similar semantics, except on type information.

As an example, given a 3D array

int buffer[5][7][6];

An element at location [2][1][2] can be accessed as “buffer[2][1][2]” or \*( \*( \*(buffer + 2) + 1) + 2).

Observe the following declaration

T \*p; // p is a pointer to an object of type T

When a pointer p is pointing to an object of type T, the expression \*p is of type T. For example buffer is of type array of 5 two dimensional arrays. The type of the expression \*buffer is “array of arrays (i.e. two dimensional array)”.

Based on the above concept translating the expression \*( \*( \*(buffer + 2) + 1) + 2) step-by-step makes it more clear.

1. buffer – An array of 5 two dimensional arrays, i.e. its type is “array of 5 two dimensional arrays”.
2. buffer + 2 – displacement for 3rd element in the array of 5 two dimensional arrays.
3. \*(buffer + 2) – dereferencing, i.e. its type is now two dimensional array.
4. \*(buffer + 2) + 1 – displacement to access 2nd element in the array of 7 one dimensional arrays.
5. \*( \*(buffer + 2) + 1) – dereferencing (accessing), now the type of expression “\*( \*(buffer + 2) + 1)” is an array of integers.
6. \*( \*(buffer + 2) + 1) + 2 – displacement to get element at 3rd position in the single dimension array of integers.
7. \*( \*( \*(buffer + 2) + 1) + 2) – accessing the element at 3rd position (the overall expression type is int now).

The compiler calculates an “offset” to access array element. The “offset” is calculated based on dimensions of the array. In the above case, offset = 2 \* (7 \* 6) + 1 \* (6) + 2. Those in blue colour are dimensions, note that the higher dimension is not used in offset calculation. During compile time the compiler is aware of dimensions of array. Using offset we can access the element as shown below,

element\_data = \*( (int \*)buffer + offset );

It is not always possible to declare dimensions of array at compile time. Sometimes we need to interpret a buffer as multidimensional array object. For instance, when we are processing 3D image whose dimensions are determined at run-time, usual array subscript rules can’t be used. It is due to lack of fixed dimensions during compile time. Consider the following example,

int \*base;

Where base is pointing large image buffer that represents 3D image of dimension l x b x h where l, b and h are variables. If we want to access an element at location (2, 3, 4) we need to calculate offset of the element as

offset = 2 \* (b x h) + 3 \* (h) + 4 and the element located at base + offset.

Generalizing further, given start address (say base) of an array of size [**l x b x h**] dimensions, we can access the element at an arbitrary location (**a, b, c**) in the following way,

data = \*(base + a \* (***b x h***) + b \* (***h***) + c); // Note that we haven’t used the higher dimension **l**.

The same concept can be applied to any number of dimensions. We don’t need the higher dimension to calculate offset of any element in the multidimensional array. It is the reason behind omitting the higher dimension when we pass multidimensional arrays to functions. The higher dimension is needed only when the programmer iterating over limited number of elements of higher dimension.

A C/C++ puzzle, predict the output of following program

|  |
| --- |
| int main()  {      char arr[5][7][6];      char (\*p)[5][7][6] = &arr;        /\* Hint: &arr - is of type const pointer to an array of         5 two dimensional arrays of size [7][6] \*/        printf("%d\n", (&arr + 1) - &arr);      printf("%d\n", (char \*)(&arr + 1) - (char \*)&arr);      printf("%d\n", (unsigned)(arr + 1) - (unsigned)arr);      printf("%d\n", (unsigned)(p + 1) - (unsigned)p);        return 0;  } |

Run on IDE

**Output:**

1

210

42

210

# **References in C++**

When a variable is declared as reference, it becomes an alternative name for an existing variable. A variable can be declared as reference by putting ‘&’ in the declaration.

|  |
| --- |
| #include<iostream>  using namespace std;    int main()  {    int x = 10;      // ref is a reference to x.    int& ref = x;      // Value of x is now changed to 20    ref = 20;    cout << "x = " << x << endl ;      // Value of x is now changed to 30    x = 30;    cout << "ref = " << ref << endl ;      return 0;  } |

Run on IDE

Output:

x = 20

ref = 30

Following is one more example that uses references to swap two variables.

|  |
| --- |
| #include<iostream>  using namespace std;    void swap (int& first, int& second)  {      int temp = first;      first = second;      second = temp;  }    int main()  {      int a = 2, b = 3;      swap( a, b );      cout << a << " " << b;      return 0;  } |

Run on IDE

Output:

3 2

**References vs Pointers**  
Both references and pointers can be used to change local variables of one function inside another function. Both of them can also be used to save copying of big objects when passed as arguments to functions or returned from functions, to get efficiency gain.  
Despite above similarities, there are following differences between references and pointers.

A pointer can be declared as void but a reference can never be void  
For example

int a = 10;

void\* aa = &a;. //it is valid

void &ar = a; // it is not valid

Thanks to Shweta Bansal for adding this point.

References are less powerful than pointers  
1) Once a reference is created, it cannot be later made to reference another object; it cannot be reseated. This is often done with pointers.  
2) References cannot be NULL. Pointers are often made NULL to indicate that they are not pointing to any valid thing.  
3) A reference must be initialized when declared. There is no such restriction with pointers

Due to the above limitations, references in C++ cannot be used for implementing data structures like Linked List, Tree, etc. In Java, references don’t have above restrictions, and can be used to implement all data structures. References being more powerful in Java, is the main reason Java doesn’t need pointers.

References are safer and easier to use:  
1) Safer: Since references must be initialized, wild references like [wild pointers](https://www.geeksforgeeks.org/archives/4979) are unlikely to exist. It is still possible to have references that don’t refer to a valid location (See questions 5 and 6 in the below exercise )  
2) Easier to use: References don’t need dereferencing operator to access the value. They can be used like normal variables. ‘&’ operator is needed only at the time of declaration. Also, members of an object reference can be accessed with dot operator (‘.’), unlike pointers where arrow operator (->) is needed to access members.

Together with the above reasons, there are few places like copy constructor argument where pointer cannot be used. Reference must be used pass the argument in copy constructor. Similarly references must be used for overloading some operators like ++.  
  
**Exercise:**  
Predict the output of following programs. If there are compilation errors, then fix them.

**Question 1**

|  |
| --- |
| #include<iostream>  using namespace std;  int &fun()  {      static int x = 10;      return x;  }  int main()  {      fun() = 30;      cout << fun();      return 0;  } |

//O/p 30  
**Question 2**

|  |
| --- |
| #include<iostream>  using namespace std;  int fun(int &x)  {      return x;  }  int main()  {      cout << fun(10);      return 0;  } |

//error. Cannot directly assign const 10  
**Question 3**

|  |
| --- |
| #include<iostream>  using namespace std;  void swap(char \* &str1, char \* &str2)  {    char \*temp = str1;    str1 = str2;    str2 = temp;  }    int main()  {    char \*str1 = "GEEKS";    char \*str2 = "FOR GEEKS";    swap(str1, str2);    cout<<"str1 is "<<str1<<endl;    cout<<"str2 is "<<str2<<endl;    return 0;  } |

//swap is done  
**Question 4**

|  |
| --- |
| #include<iostream>  using namespace std;  int main()  {     int x = 10;     int \*ptr = &x;     int &\*ptr1 = ptr;  }//error: cannot declare pointer to 'int&'  int &\*ptr1 = ptr; |

**Question 5**

|  |
| --- |
| #include<iostream>  using namespace std;  int main()  {     int \*ptr = NULL;     int &ref = \*ptr;     cout << ref;  } |

**Question 6**

|  |
| --- |
| #include<iostream>  using namespace std;  int &fun()  {      int x = 10;      return x;  }  int main()  {      fun() = 30;      cout << fun();      return 0;  } |

# **What is Array Decay in C++? How can it be prevented?**

**What is Array Decay?**  
The loss of type and dimensions of an array is known as decay of an array.This generally occurs when we pass the array into function by value or pointer. What it does is, it sends first address to the array which is a pointer, hence the size of array is not the original one, but the one occupied by the pointer in the memory.

|  |
| --- |
| // C++ code to demonstrate array decay  #include<iostream>  using namespace std;    // Driver function to show Array decay  // Passing array by value  void aDecay(int \*p)  {      // Printing size of pointer      cout << "Modified size of array is by "              " passing by value: ";      cout << sizeof(p) << endl;  }    // Function to show that array decay happens  // even if we use pointer  void pDecay(int (\*p)[7])  {      // Printing size of array      cout << "Modified size of array by "              "passing by pointer: ";      cout << sizeof(p) << endl;  }    int main()  {      int a[7] = {1, 2, 3, 4, 5, 6, 7,};        // Printing original size of array      cout << "Actual size of array is: ";      cout << sizeof(a) <<endl;        // Calling function by value      aDecay(a);        // Calling function by pointer      pDecay(&a);        return 0;  } |

Run on IDE

Output:

Actual size of array is: 28

Modified size of array by passing by value: 8

Modified size of array by passing by pointer: 8

In the above code, the actual array has 7 int elements and hence has 28 size. But by calling by value and pointer, array decays into pointer and prints the size of 1 pointer i.e. 8 (4 in 32 bit).

**How to prevent Array Decay?**  
A typical solution to handle decay is to pass size of array also as a parameter and not use sizeof on array parameters (See [this](https://www.geeksforgeeks.org/using-sizof-operator-with-array-paratmeters/) for details)

Another way to prevent array decay is to send the array into functions by reference. This prevents conversion of array into a pointer, hence prevents the decay.

|  |
| --- |
| // C++ code to demonstrate prevention of  // decay of array  #include<iostream>  using namespace std;    // A function that prevents Array decay  // by passing array by reference  void fun(int (&p)[7])  {      // Printing size of array      cout << "Modified size of array by "              "passing by reference: ";      cout << sizeof(p) << endl;  }    int main()  {      int a[7] = {1, 2, 3, 4, 5, 6, 7,};        // Printing original size of array      cout << "Actual size of array is: ";      cout << sizeof(a) <<endl;        // Calling function by reference      fun(a);        return 0;  } |

Run on IDE

Output:

Actual size of array is: 28

Modified size of array by passing by reference: 28

In the above code, passing array by reference solves the problem of decay of array. Sizes in both cases is 28.

# **Namespace in C++ | Set 1 (Introduction)**

Consider following C++ program.

|  |
| --- |
| // A program to demonstrate need of namespace  int main()  {      int value;      value = 0;      double value; // Error here      value = 0.0;  } |

Run on IDE

Output :

Compiler Error:

'value' has a previous declaration as 'int value'

In each scope, a name can only represent one entity. So, there cannot be two variables with the same name in the same scope. Using namespaces, we can create two variables or member functions having the same name.

|  |
| --- |
| // Here we can see that more than one variables  // are being used without reporting any error.  // That is because they are declared in the  // different namespaces and scopes.  #include <iostream>  using namespace std;    // Variable created inside namespace  namespace first  {      int val = 500;  }    // Global variable  int val = 100;    int main()  {      // Local variable      int val = 200;        // These variables can be accessed from      // outside the namespace using the scope      // operator ::      cout << first::val << '\n';        return 0;  } |

Run on IDE

**Output:**

500

**Definition and Creation:**

Namespaces allow us to group named entities that otherwise would have *global scope* into narrower scopes, giving them *namespace scope*. This allows organizing the elements of programs into different logical scopes referred to by names.

* Namespace is a feature added in C++ and not present in C.
* A namespace is a declarative region that provides a scope to the identifiers (names of the types, function, variables etc) inside it.
* Multiple namespace blocks with the same name are allowed. All declarations within those blocks are declared in the named scope.

A namespace definition begins with the keyword **namespace** followed by the namespace name as follows:

namespace namespace\_name

{

int x, y; // code declarations where

// x and y are declared in

// namespace\_name's scope

}

* Namespace declarations appear only at global scope.
* Namespace declarations can be nested within another namespace.
* Namespace declarations don’t have access specifiers. (Public or private)
* No need to give semicolon after the closing brace of definition of namespace.
* We can split the definition of namespace over several units.

|  |
| --- |
| // Creating namespaces  #include <iostream>  using namespace std;  namespace ns1  {      int value()    { return 5; }  }  namespace ns2  {      const double x = 100;      double value() {  return 2\*x; }  }    int main()  {      // Access value function within ns1      cout << ns1::value() << '\n';        // Access value function within ns2      cout << ns2::value() << '\n';        // Access variable x directly      cout << ns2::x << '\n';        return 0;  } |

**Output:**

5

200

100

**Classes and Namespace:**

Following is a simple way to create classes in a name space

|  |
| --- |
| // A C++ program to demonstrate use of class  // in a namespace  #include <iostream>  using namespace std;    namespace ns  {      // A Class in a namespace      class geek      {      public:          void display()          {              cout << "ns::geek::display()\n";          }      };  }    int main()  {      // Creating Object of student Class      ns::geek obj;        obj.display();        return 0;  } |

Run on IDE

Output:

ns::geek::display()

**Class can also be declared inside namespace and defined outside namespace** using following syntax

|  |
| --- |
| // A C++ program to demonstrate use of class  // in a namespace  #include <iostream>  using namespace std;    namespace ns  {      // Only declaring class here      class geek;  }    // Defining class outside  class ns::geek  {  public:      void display()      {          cout << "ns::geek::display()\n";      }  };    int main()  {      //Creating Object of student Class      ns::geek obj;      obj.display();      return 0;  } |

Run on IDE

Output:

ns::geek::display()

We can **define methods also outside the namespace**. Following is an example code.

|  |
| --- |
| // A C++ code to demonstrate that we can define  // methods outside namespace.  #include <iostream>  using namespace std;    // Creating a namespace  namespace ns  {      void display();      class geek      {      public:         void display();      };  }    // Defining methods of namespace  void ns::geek::display()  {      cout << "ns::geek::display()\n";  }  void ns::display()  {      cout << "ns::display()\n";  }    // Driver code  int main()  {      ns::geek obj;      ns::display();      obj.display();      return 0;  } |

Run on IDE

**Output:**

ns::display()

ns::geek::display()

# **namespace in C++ | Set 2 (Extending namespace and Unnamed namespace)**

We have introduced namespaces in below set 1.

[Namespace in C++ | Set 1 (Introduction)](https://www.geeksforgeeks.org/namespace-in-c/)

It is also possible to create more than one namespaces in the global space. This can be done in two ways.

* **namespaces having different names**

|  |
| --- |
| // A C++ program to show more than one namespaces  // with different names.  #include <iostream>  using namespace std;    // first name space  namespace first  {     int func() {  return 5; }  }    // second name space  namespace second  {     int func() { return 10; }  }    int main()  {     // member function of namespace     // accessed using scope resolution operator     cout << first::func() <<"\n";     cout << second::func() <<"\n";     return 0;  } |

* Run on IDE

Output:

5

10

* **Extending namespaces (Using same name twice)**  
  It is also possible to create two namespace blocks having the same name. The second namespace block is nothing but actually the continuation of the first namespace. In simpler words, we can say that both the namespaces are not different but actually the same, which are being defined in parts.

|  |
| --- |
| // C++ program to demonstrate namespace exntension  #include <iostream>  using namespace std;    // first name space  namespace first  {     int val1 = 500;  }    // rest part of the first namespace  namespace  first  {     int val2 = 501;  }    int main()  {     cout << first::val1 <<"\n";     cout << first::val2 <<"\n";     return 0;  } |

Run on IDE

Output:

500

501

**Unnamed Namespaces**

* They are directly usable in the same program and are used for declaring unique identifiers.
* In unnamed namespaces, name of the namespace in not mentioned in the declaration of namespace.
* The name of the namespace is uniquely generated by the compiler.
* The unnamed namespaces you have created will only be accessible within the file you created it in.
* Unnamed namespaces are the replacement for the static declaration of variables.

|  |
| --- |
| // C++ program to demonstrate working of unnamed  // namespaces  #include <iostream>  using namespace std;    // unnamed namespace declaration  namespace  {     int rel = 300;  }    int main()  {     cout << rel << "\n"; // prints 300     return 0;  } |

Run on IDE

Output:

300

# **Namespace in C++ | Set 3 (Accessing, creating header, nesting and aliasing)**

[Namespace in C++ | Set 1 (Introduction)](https://www.geeksforgeeks.org/namespace-in-c/)  
[Namespace in C++ | Set 2 (Extending namespace and Unnamed namespace)](https://www.geeksforgeeks.org/namespace-in-c-set-2-extending-namespace-and-unnamed-namespace/)

**Different ways to access namespace**

In C++, there are two ways of accessing namespace variables and functions.

1. **Normal way**

|  |
| --- |
| // C++ program to demonstrate accessing of variables  // in normal way, i.e., using "::"  #include <iostream>  using namespace std;  namespace geek  {      int rel = 300;  }    int main()  {      // variable ‘rel’ accessed using scope resolution operator      cout << geek::rel << "\n";  // prints 300       return 0;  } |

Output :

300

1. **“using” directive**

|  |
| --- |
| // C++ program to demonstrate accessing of variables  // in normal way, i.e., using "using" directive  #include <iostream>  using namespace std;    namespace geek  {      int rel = 300;  }    // use of ‘using’ directive  using namespace geek;    int main()  {     // variable ‘rel’ accessed     // without using scope resolution variable     cout << rel << "\n";        //prints 300       return 0;  } |

Run on IDE

Output:

300

**Using namespace in header files**

We can create namespace in one file and access contents using another program. This is done in the following manner.

* We need to create two files. One containing the namespace and all the data members and member functions we want to use later.
* And the other program can directly call the first program to use all the data members and member functions in it.

**File 1**

|  |
| --- |
| // file1.h  namespace foo  {      int value()      {         return 5;      }  } |

**File 2**

|  |
| --- |
| // file2.cpp - Not to be executed online  #include <iostream>  #include “file1.h” // Including file1  using namespace std;    int main ()  {      cout << foo::value();      return 0;  } |

Here we can see that the namespace is created in file1.h and the value() of that namespace is getting called in file2.cpp.

**Nested Namespaces**

In C++, namespaces can also be nested i.e., one namespace inside another. The resolution of namespace variables is hierarchical.

|  |
| --- |
| // C++ program to demonstrate nesting of namespaces  #include <iostream>  using namespace std;    // Nested namespace  namespace out  {    int val = 5;    namespace in    {        int val2 = val;    }  }    // Driver code  int main()  {    cout << out::in::val2;   // prints 5    return 0;  } |

OUTPUT :

5

**Namespace Aliasing**

In C++, you can use an alias name for your namespace name, for ease of use. Existing namespaces can be aliased with new names, with the following syntax:

namespace new\_name = current\_name;

|  |
| --- |
| #include <iostream>    namespace name1  {      namespace name2      {           namespace name3           {               int var = 42;           }      }  }    // Aliasing  namespace alias = name1::name2::name3;    int main()  {      std::cout << alias::var << '\n';  } |

Output :

42

# **Inline namespaces and usage of the “using” directive inside namespaces**

Prerequisite : [Namespaces in C++](https://www.geeksforgeeks.org/namespace-in-c/)

An inline namespace is a namespace that uses the optional keyword inline in its original-namespace-definition.

|  |
| --- |
| // C++ program to demonstrate working of inline namespaces  #include <iostream>  using namespace std;    namespace ns1  {     inline namespace ns2     {         int var = 10;     }  }    int main()  {     cout << ns1::var;     return 0;  } |

Output:

10

We can see from above example that members of an inline namespace are treated as if they are members of the enclosing namespace in many situations (listed below). This property is transitive: if a namespace N contains an inline namespace M, which in turn contains an inline namespace O, then the members of O can be used as though they were members of M or N.

|  |
| --- |
| // C++ program to demonstrate working of  // inline namespaces inside inline namespaces    #include <iostream>  using namespace std;    namespace ns1  {      inline namespace ns2      {          inline namespace ns3          {              int var = 10;          }      }  }    int main()  {      cout << ns1::var;      return 0;  } |

Run on IDE

Output:

10

The inline specifier makes the declarations from the nested namespace appear exactly as if they had been declared in the enclosing namespace. This means it drags out the declaration (“var” in the above example) from a nested namespace to the containing namespace.

Advantages of using inline namespaces:

* **Avoid verbose :**Consider the above code, if you want to print “var”, you write:

cout << ns1::ns2::ns3::var;

This looks good only if namespace's names are short as in the above example. But by using inline with namespaces there is no need to type the entire namespace as given above or use the "using" directive.

* **Support of Library :**The inline namespace mechanism is intended to support library evolution by providing a mechanism that supports a form of versioning. Refer [this](http://www.stroustrup.com/C++11FAQ.html#inline-namespace) for details.

**"Using" directive**

This same behavior (same as inline namesapces) can also be achieved by using the "using" declarative inside namespaces. A using-directive that names the inline namespace is implicitly inserted in the enclosing namespace (similar to the implicit using-directive for the unnamed namespace). Consider the following C++ code:

|  |
| --- |
| // C++ program to demonstrate working  // of "using" to get the same effect as inline.  #include <iostream>  using namespace std;    namespace ns1  {      namespace ns2      {          namespace ns3          {              int var = 10;          }          using namespace ns3;      }        using namespace ns2;  }    int main()  {      cout << ns1::var;      return 0;  } |

Run on IDE

Output :

10

Here again, the using directive makes the declarations from the nested namespace appear exactly as if they had been declared in the enclosing namespace.

In C++, [namespaces](https://www.geeksforgeeks.org/namespace-in-c/) can be nested, and resolution of namespace variables is hierarchical. For example, in the following code, namespace inner is created inside namespace outer, which is inside the global namespace. In the line “int z = x”, x refers to outer::x. If x would not have been in outer then this x would have referred to x in global namespace.

|  |
| --- |
| #include <iostream>    int x = 20;  namespace outer {    int x = 10;    namespace inner {      int z = x; // this x refers to outer::x    }  }    int main()  {    std::cout<<outer::inner::z; //prints 10    getchar();    return 0;  } |

Run on IDE

Output of the above program is 10.

On a side node, unlike C++ namespaces, Java packages are not hierarchical.

# **Default Arguments in C++**

A default argument is a value provided in function declaration that is automatically assigned by the compiler if caller of the function doesn’t provide a value for the argument with default value.

Following is a simple C++ example to demonstrate use of default arguments. We don’t have to write 3 sum functions, only one function works by using default values for 3rd and 4th arguments.

|  |
| --- |
| #include<iostream>  using namespace std;    // A function with default arguments, it can be called with  // 2 arguments or 3 arguments or 4 arguments.  int sum(int x, int y, int z=0, int w=0)  {      return (x + y + z + w);  }    /\* Drier program to test above function\*/  int main()  {      cout << sum(10, 15) << endl;      cout << sum(10, 15, 25) << endl;      cout << sum(10, 15, 25, 30) << endl;      return 0;  } |

Run on IDE

Output:

25

50

80

**Key Points:**

* Default arguments are different from constant arguments as constant arguments can’t be changed whereas default arguments can be overwritten if required.
* Default arguments are overwritten when calling function provides values for them. For example, calling of function sum(10, 15, 25, 30) overwrites the value of z and w to 25 and 30 respectively.
* During calling of function, arguments from calling function to called function are copied from left to right. Therefore, sum(10, 15, 25) will assign 10, 20 and 25 to x, y and z. Therefore, default value is used for w only.
* Once default value is used for an argument in function definition, all subsequent arguments to it must have default value. It can also be stated as default arguments are assigned from right to left. For example, the following function definition is invalid as subsequent argument of default variable z is not default.

|  |
| --- |
| // Invalid because z has default value, but w after it  // doesn't have default value  int sum(int x, int y, int z=0, int w) |

# **Inline Functions in C++**

Inline function is one of the important feature of C++. So, let’s first understand why inline functions are used and what is the purpose of inline function?

When the program executes the function call instruction the CPU stores the memory address of the instruction following the function call, copies the arguments of the function on the stack and finally transfers control to the specified function. The CPU then executes the function code, stores the function return value in a predefined memory location/register and returns control to the calling function. This can become overhead if the execution time of function is less than the switching time from the caller function to called function (callee). For functions that are large and/or perform complex tasks, the overhead of the function call is usually insignificant compared to the amount of time the function takes to run. However, for small, commonly-used functions, the time needed to make the function call is often a lot more than the time needed to actually execute the function’s code. This overhead occurs for small functions because execution time of small function is less than the switching time.

C++ provides an inline functions to reduce the function call overhead. Inline function is a function that is expanded in line when it is called. When the inline function is called whole code of the inline function gets inserted or substituted at the point of inline function call. This substitution is performed by the C++ compiler at compile time. Inline function may increase efficiency if it is small.  
The syntax for defining the function inline is:

inline return-type function-name(parameters)

{

// function code

}

Remember, inlining is only a request to the compiler, not a command. Compiler can ignore the request for inlining. Compiler may not perform inlining in such circumstances like:  
1) If a function contains a loop. (for, while, do-while)  
2) If a function contains static variables.  
3) If a function is recursive.  
4) If a function return type is other than void, and the return statement doesn’t exist in function body.  
5) If a function contains switch or goto statement.

**Inline functions provide following advantages:**  
1) Function call overhead doesn’t occur.  
2) It also saves the overhead of push/pop variables on the stack when function is called.  
3) It also saves overhead of a return call from a function.  
4) When you inline a function, you may enable compiler to perform context specific optimization on the body of function. Such optimizations are not possible for normal function calls. Other optimizations can be obtained by considering the flows of calling context and the called context.  
5) Inline function may be useful (if it is small) for embedded systems because inline can yield less code than the function call preamble and return.

**Inline function disadvantages:**  
1) The added variables from the inlined function consumes additional registers, After in-lining function if variables number which are going to use register increases than they may create overhead on register variable resource utilization. This means that when inline function body is substituted at the point of function call, total number of variables used by the function also gets inserted. So the number of register going to be used for the variables will also get increased. So if after function inlining variable numbers increase drastically then it would surely cause an overhead on register utilization.

2) If you use too many inline functions then the size of the binary executable file will be large, because of the duplication of same code.

3) Too much inlining can also reduce your instruction cache hit rate, thus reducing the speed of instruction fetch from that of cache memory to that of primary memory.

4) Inline function may increase compile time overhead if someone changes the code inside the inline function then all the calling location has to be recompiled because compiler would require to replace all the code once again to reflect the changes, otherwise it will continue with old functionality.

5) Inline functions may not be useful for many embedded systems. Because in embedded systems code size is more important than speed.

6) Inline functions might cause thrashing because inlining might increase size of the binary executable file. Thrashing in memory causes performance of computer to degrade.

The following program demonstrates the use of use of inline function.

|  |
| --- |
| #include <iostream>  using namespace std;  inline int cube(int s)  {      return s\*s\*s;  }  int main()  {      cout << "The cube of 3 is: " << cube(3) << "\n";      return 0;  } //Output: The cube of 3 is: 27 |

Run on IDE

**Inline function and classes:**  
It is also possible to define the inline function inside the class. In fact, all the functions defined inside the class are implicitly inline. Thus, all the restrictions of inline functions are also applied here. If you need to explicitly declare inline function in the class then just declare the function inside the class and define it outside the class using inline keyword.  
For example:

|  |
| --- |
| class S  {  public:      inline int square(int s) // redundant use of inline      {          // this function is automatically inline          // function body      }  }; |

Run on IDE

The above style is considered as a bad programming style. The best programming style is to just write the prototype of function inside the class and specify it as an inline in the function definition.  
For example:

|  |
| --- |
| class S  {  public:      int square(int s); // declare the function  };    inline int S::square(int s) // use inline prefix  {    } |

Run on IDE

The following program demonstrates this concept:

|  |
| --- |
| #include <iostream>  using namespace std;  class operation  {      int a,b,add,sub,mul;      float div;  public:      void get();      void sum();      void difference();      void product();      void division();  };  inline void operation :: get()  {      cout << "Enter first value:";      cin >> a;      cout << "Enter second value:";      cin >> b;  }    inline void operation :: sum()  {      add = a+b;      cout << "Addition of two numbers: " << a+b << "\n";  }    inline void operation :: difference()  {      sub = a-b;      cout << "Difference of two numbers: " << a-b << "\n";  }    inline void operation :: product()  {      mul = a\*b;      cout << "Product of two numbers: " << a\*b << "\n";  }    inline void operation ::division()  {      div=a/b;      cout<<"Division of two numbers: "<<a/b<<"\n" ;  }    int main()  {      cout << "Program using inline function\n";      operation s;      s.get();      s.sum();      s.difference();      s.product();      s.division();      return 0;  } |

Run on IDE

Output:

Enter first value: 45

Enter second value: 15

Addition of two numbers: 60

Difference of two numbers: 30

Product of two numbers: 675

Division of two numbers: 3

**What is wrong with macro?**  
Readers familiar with the C language knows that C language uses macro. The preprocessor replace all macro calls directly within the macro code. It is recommended to always use inline function instead of macro. According to Dr. Bjarne Stroustrup the creator of C++ that macros are almost never necessary in C++ and they are error prone. There are some problems with the use of macros in C++. Macro cannot access private members of class. Macros looks like function call but they are actually not.  
Example:

|  |
| --- |
| #include <iostream>  using namespace std;  class S  {      int m;  public:  #define MAC(S::m)    // error  }; |

Run on IDE

C++ compiler checks the argument types of inline functions and necessary conversions are performed correctly. Preprocessor macro is not capable for doing this. One other thing is that the macros are managed by preprocessor and inline functions are managed by C++ compiler.

Remember: It is true that all the functions defined inside the class are implicitly inline and C++ compiler will perform inline call of these functions, but C++ compiler cannot perform inlining if the function is virtual. The reason is call to a virtual function is resolved at runtime instead of compile time. Virtual means wait until runtime and inline means during compilation, if the compiler doesn’t know which function will be called, how it can perform inlining?

One other thing to remember is that it is only useful to make the function inline if the time spent during a function call is more compared to the function body execution time. An example where inline function has no effect at all:

|  |
| --- |
| inline void show()  {      cout << "value of S = " << S << endl;  } |

Run on IDE

The above function relatively takes a long time to execute. In general function which performs input output (I/O) operation shouldn’t be defined as inline because it spends a considerable amount of time. Technically inlining of show() function is of limited value because the amount of time the I/O statement will take far exceeds the overhead of a function call.

Depending upon the compiler you are using the compiler may show you warning if the function is not expanded inline. Programming languages like Java & C# doesn’t support inline functions.  
But in Java, the compiler can perform inlining when the small final method is called, because final methods can’t be overridden by sub classes and call to a final method is resolved at compile time. In C# JIT compiler can also optimize code by inlining small function calls (like replacing body of a small function when it is called in a loop).

Last thing to keep in mind that inline functions are the valuable feature of C++. An appropriate use of inline function can provide performance enhancement but if inline functions are used arbitrarily then they can’t provide better result. In other words don’t expect better performance of program. Don’t make every function inline. It is better to keep inline functions as small as possible.

**References:**  
1) [Effective C++ , Scott Meyers](http://www.flipkart.com/effective-c-55-specific-ways-improve-your-programs-designs-3rd/p/itmczzfe2gfvfuch?pid=9788131714805&affid=sandeepgfg)  
2) <http://www.parashift.com/c++-faq/inline-and-perf.html>  
3) <http://www.cplusplus.com/forum/articles/20600/>  
4) [Thinking in C++, Volume 1, Bruce Eckel](http://www.flipkart.com/thinking-c-volume-1-with-cd/p/itmdwuafcz75hzjy?pid=9788131706619&affid=sandeepgfg).  
5) [C++ the complete reference, Herbert Schildt](http://www.flipkart.com/c-complete-reference/p/itmdwxz7nyaxabtj?pid=9780070532465&affid=sandeepgfg)

# **Return from void functions in C++**

Void functions are “void” due to the fact that they are not supposed to return values. True, but not completely. We cannot return values but there is something we can surely return from void functions. Some of cases are listed below.

**A void function can do return**  
We can simply write return statement in a void fun(). In-fact it is considered a good practice (for readability of code) to write return; statement to indicate end of function.

|  |
| --- |
| #include <iostream>  using namespace std;    void fun()  {     cout << "Hello";       // We can write return in void     return;  }    int main()  {     fun();     return 0;  } |

Run on IDE

Output :

Hello

**A void fun() can return another void function**

|  |
| --- |
| // C++ code to demonstrate void()  // returning void()  #include<iostream>  using namespace std;    // A sample void function  void work()  {      cout << "The void function has returned "              " a void() !!! \n";  }    // Driver void() returning void work()  void test()  {      // Returning void function      return work();  }    int main()  {      // Calling void function      test();      return 0;  } |

Run on IDE

Output:

The void function has returned a void() !!!

The above code explains how void() can actually be useful to return void functions without giving error.

**A void() can return a void value.**  
A void() cannot return a value that can be used. But it can return a value which is void without giving an error.

|  |
| --- |
| // C++ code to demonstrate void()  // returning a void value  #include<iostream>  using namespace std;    // Driver void() returning a void value  void test()  {      cout << "Hello";        // Returning a void value      return (void)"Doesn't Print";  }  int main()  {      test();      return 0;  } |

Run on IDE

Output:

Hello

# **Returning multiple values from a function using Tuple and Pair in C++**

There can be some instances where you need to return multiple values (may be of different data types ) while solving a problem. One method to do the same is by using pointers, structures or global variables, already discussed [here](https://www.geeksforgeeks.org/how-can-i-return-multiple-values-from-a-function/)  
There is another interesting method to do the same without using the above methods,  using tuples (for returning multiple values ) and pair (for two values).

We can declare the function with return type as pair or tuple (whichever required) and can pack the values to be returned and return the packed set of values. The returned values can be unpacked in the calling function.

**Tuple**

* A tuple is an object capable to hold a collection of elements where each element can be of a different type.
* Class template std::tuple is a fixed-size collection of heterogeneous values

**Pair**

* This class couples together a pair of values, which may be of different types
* A pair is a specific case of a std::tuple with two elements

Note : Tuple can also be used to return two values instead of using pair .

|  |
| --- |
| #include<bits/stdc++.h>  using namespace std;    // A Method that returns multiple values using  // tuple in C++.  tuple<int, int, char> foo(int n1, int n2)  {      // Packing values to return a tuple      return make\_tuple(n2, n1, 'a');  }    // A Method returns a pair of values using pair  std::pair<int, int> foo1(int num1, int num2)  {      // Packing two values to return a pair      return std::make\_pair(num2, num1);  }    int main()  {      int a,b;      char cc;        // Unpack the elements returned by foo      tie(a, b, cc) = foo(5, 10);        // Storing  returned values in a pair      pair<int, int> p = foo1(5,2);        cout << "Values returned by tuple: ";      cout << a << " " << b << " " << cc << endl;        cout << "Values returned by Pair: ";      cout << p.first << " " << p.second;      return 0;  } |

Run on IDE

Output:

Values returned by tuple: 10 5 a

Values returned by Pair: 2 5

# **Difference between “int fun()” and “int fun(void)” in C/C++?**

Consider the following two definitions of main().

|  |
| --- |
| int main()  {     /\*  \*/     return 0;  } |

and

|  |
| --- |
| int main(void)  {     /\*  \*/     return 0;  } |

What is the difference?

In C++, there is no difference, both are same.

Both definitions work in C also, but the second definition with void is considered technically better as it clearly specifies that main can only be called without any parameter.  
In C, if a function signature doesn’t specify any argument, it means that the function can be called with any number of parameters or without any parameters.

In C++, func() is equivalent to func(void)  
In C, func() is equivalent to func(…)

For example, try to compile and run following two C programs (remember to save your files as .c). Note the difference between two signatures of fun().

|  |
| --- |
| // Program 1 (Compiles and runs fine in C, but not in C++)  void fun() {  }  int main(void)  {      fun(10, "GfG", "GQ");      return 0;  } |

Run on IDE

The above program compiles and runs fine (See [this](http://ideone.com/AQoVZW)), but the following program fails in compilation (see [this](http://ideone.com/IXojiK))

|  |
| --- |
| // Program 2 (Fails in compilation in both C and C++)  void fun(void) {  }  int main(void)  {      fun(10, "GfG", "GQ");      return 0;  } |

Run on IDE

Unlike C, in C++, both of the above programs fails in compilation. In C++, both fun() and fun(void) are same.

So the difference is, in C, int main() can be called with any number of arguments, but int main(void)can only be called without any argument. Although it doesn’t make any difference most of the times, using “int main(void)” is a recommended practice in C.

**Exercise:**  
Predict the output of following **C** programs.

**Question 1**

|  |
| --- |
| #include <stdio.h>  int main()  {      static int i = 5;      if (--i){          printf("%d ", i);          main(10);      }  } |

Output: 4 3 2 1

**Question 2**

|  |
| --- |
| #include <stdio.h>  int main(void)  {      static int i = 5;      if (--i){          printf("%d ", i);          main(10);      }  } |

Output: error. Too many arguments to funcion main()

# **Functors in C++**

Please note that the title is **Functors** (Not Functions)!!

Consider a function that takes only one argument. However, while calling this function we have a lot more information that we would like to pass to this function, but we cannot as it accepts only one parameter. What can be done?

One obvious answer might be global variables. However, good coding practices do not advocate the use of global variables and say they must be used only when there is no other alternative.

**Functors** are objects that can be treated as though they are a function or function pointer. Functors are most commonly used along with STLs in a scenario like following:

Below program uses [transform() in STL](https://www.geeksforgeeks.org/transform-c-stl-perform-operation-elements/) to add 1 to all elements of arr[].

|  |
| --- |
| // A C++ program uses transform() in STL to add  // 1 to all elements of arr[]  #include <bits/stdc++.h>  using namespace std;    int increment(int x) {  return (x+1); }    int main()  {      int arr[] = {1, 2, 3, 4, 5};      int n = sizeof(arr)/sizeof(arr[0]);        // Apply increment to all elements of      // arr[] and store the modified elements      // back in arr[]      transform(arr, arr+n, arr, increment);        for (int i=0; i<n; i++)          cout << arr[i] << S" ";        return 0;  } |

Run on IDE

Output:

2 3 4 5 6

This code snippet adds only one value to the contents of the arr[]. Now suppose, that we want to add 5 to contents of arr[].

See what’s happening? As transform requires a unary function(a function taking only one argument) for an array, we cannot pass a number to increment(). And this would, in effect, make us write several different functions to add each number. What a mess. This is where functors come into use.

A functor (or function object) is a C++ class that acts like a function. Functors are called using the same old function call syntax. To create a functor, we create a object that overloads the operator().

**The line,**

MyFunctor(10);

**Is same as**

MyFunctor.operator()(10);

Let’s delve deeper and understand how this can actually be used in conjunction with STLs.

|  |
| --- |
| // C++ program to demonstrate working of  // functors.  #include <bits/stdc++.h>  using namespace std;    // A Functor  class increment  {  private:      int num;  public:      increment(int n) : num(n) {  }        // This operator overloading enables calling      // operator function () on objects of increment      int operator () (int arr\_num) const {          return num + arr\_num;      }  };    // Driver code  int main()  {      int arr[] = {1, 2, 3, 4, 5};      int n = sizeof(arr)/sizeof(arr[0]);      int to\_add = 5;        transform(arr, arr+n, arr, increment(to\_add));        for (int i=0; i<n; i++)          cout << arr[i] << " ";  } |

Run on IDE

Output:

6 7 8 9 10

Thus, here, Increment is a functor, a c++ class that acts as a function.

**The line,**

transform(arr, arr+n, arr, increment(to\_add));

**is the same as writing below two lines,**

// Creating object of increment

increment obj(to\_add);

// Calling () on object

transform(arr, arr+n, arr, obj);

Thus, an object a is created that overloads the operator(). Hence, functors can be used effectively in conjunction with C++ STLs.

# **Const member functions in C++**

A function becomes const when const keyword is used in function’s declaration. The idea of const functions is not allow them to modify the object on which they are called. It is recommended practice to make as many functions const as possible so that accidental changes to objects are avoided.

Following is a simple example of const function.

|  |
| --- |
| #include<iostream>  using namespace std;    class Test {      int value;  public:      Test(int v = 0) {value = v;}        // We get compiler error if we add a line like "value = 100;"      // in this function.      int getValue() const {return value;}  };    int main() {      Test t(20);      cout<<t.getValue();      return 0;  } |

Run on IDE

Output:

20

When a function is declared as const, it can be called on any type of object. Non-const functions can only be called by non-const objects.

For example the following program has compiler errors.

|  |
| --- |
| #include<iostream>  using namespace std;    class Test {      int value;  public:      Test(int v = 0) {value = v;}      int getValue() {return value;}  };    int main() {      const Test t;      cout << t.getValue();      return 0;  } |

Run on IDE

Output:

passing 'const Test' as 'this' argument of 'int

Test::getValue()' discards qualifiers

# **The C++ Standard Template Library (STL)**

The Standard Template Library (STL) is a set of C++ template classes to provide common programming data structures and functions such as lists, stacks, arrays, etc. It is a library of container classes, algorithms and iterators. It is a generalized library and so, its components are parameterized. A working knowledge of [template classes](https://www.geeksforgeeks.org/template-specialization-c/) is a prerequisite  for working with STL.

**STL has four components**

* Algorithms
* Containers
* Functions
* Iterators

**Algorithms**

The header algorithm defines a collection of functions especially designed to be used on ranges of elements.They act on containers and provide means for various operations  for the contents of the containers.

* Algorithm
  + [Sorting](http://quiz.geeksforgeeks.org/sort-algorithms-the-c-standard-template-library-stl/)
  + [Searching](http://quiz.geeksforgeeks.org/binary-search-algorithms-the-c-standard-template-library-stl/)
  + [Important STL Algorithms](https://www.geeksforgeeks.org/c-magicians-stl-algorithms/)
  + [Useful Array algorithms](https://www.geeksforgeeks.org/useful-array-algorithms-in-c-stl/)
  + [Partition Operations](https://www.geeksforgeeks.org/stdpartition-in-c-stl/)
* Numeric
  + valarray class

**Containers**

Containers or container classes store objects and data. There are in total seven standard “first-class” container classes  and three container adaptor classes and only seven header files that provide access to these containers or container adaptors.

* Sequence Containers:  implement data structures which can be accessed in a sequential manner.
  + [vector](http://quiz.geeksforgeeks.org/vector-sequence-containers-the-c-standard-template-library-stl-set-1/)
  + [list](http://quiz.geeksforgeeks.org/list-sequence-containers-the-c-standard-template-library-stl/)
  + [deque](http://quiz.geeksforgeeks.org/deque-sequence-containers-the-c-standard-template-library-stl/)
  + [arrays](https://www.geeksforgeeks.org/array-class-c/)
  + [forward\_list](https://www.geeksforgeeks.org/forward-list-c-set-1-introduction-important-functions/)( Introduced in C++11)
* Container Adaptors :  provide a different interface for sequential containers.
  + [queue](http://quiz.geeksforgeeks.org/queue-container-adaptors-the-c-standard-template-library-stl/)
  + [priority\_queue](http://quiz.geeksforgeeks.org/priority-queue-container-adaptors-the-c-standard-template-library-stl/)
  + [stack](http://quiz.geeksforgeeks.org/stack-container-adaptors-the-c-standard-template-library-stl/)
* Associative Containers :  implement sorted data structures that can be quickly searched (O(log n) complexity).
  + [set](http://quiz.geeksforgeeks.org/set-associative-containers-the-c-standard-template-library-stl/)
  + [multiset](http://quiz.geeksforgeeks.org/multiset-associative-containers-the-c-standard-template-library-stl/)
  + [map](http://quiz.geeksforgeeks.org/map-associative-containers-the-c-standard-template-library-stl/)
  + [multimap](http://quiz.geeksforgeeks.org/multimap-associative-containers-the-c-standard-template-library-stl/)

**Functions**

The STL includes classes that overload the function call operator. Instances of such classes are called function objects or functors. Functors allow the working of the associated function to be customized with the help of parameters to be passed.

* [Functors](https://www.geeksforgeeks.org/functors-in-cpp/)

**Iterators**

As the name suggests, iterators are used for working upon a sequence of values. They are the major feature that allow generality in STL.

* [Iterators](https://www.geeksforgeeks.org/iterators-c-stl/)

**Utility Library**

Defined under <utility header>

* [pair](http://quiz.geeksforgeeks.org/pair-simple-containers-the-c-standard-template-library-stl/)

**References:**

* http://en.cppreference.com/w/cpp/
* http://cs.stmarys.ca/~porter/csc/ref/stl/headers.html

# **Sort in C++ Standard Template Library (STL)**

Sorting is one of the most basic functions applied on data.  
The prototype for sort is :

sort(startaddress, endaddress)

|  |
| --- |
| #include <iostream>  #include <algorithm>    using namespace std;    void show(int a[])  {      for(int i = 0; i < 10; ++i)          cout << '\t' << a[i];  }    int main()  {      int a[10]= {1, 5, 8, 9, 6, 7, 3, 4, 2, 0};      cout << "\n The array before sorting is : ";      show(a);        sort(a, a+10);        cout << "\n\n The array after sorting is : ";      show(a);        return 0;    } |

Run on IDE

The outut of the above program is :

The array before sorting is : 1 5 8

9 6 7 3 4 2 0

The array after sorting is : 0 1 2

3 4 5 6 7 8 9

# **Binary Search in C++ Standard Template Library (STL)**

Binary search is a widely used searching algorithm that requires the array to be sorted before search is applied.  
   
The prototype for binary search is :

binary\_search(startaddress, endaddress, valuetofind)

|  |
| --- |
| #include <iostream>  #include <algorithm>    using namespace std;    void show(int a[], int arraysize)  {      for(int i = 0; i < arraysize; ++i)          cout << '\t' << a[i];  }      int main()  {      int a[]= {1, 5, 8, 9, 6, 7, 3, 4, 2, 0};      int asize = sizeof(a) / sizeof(a[0]);      cout << "\n The array is : ";      show(a, asize);        cout << "\n\nLet's say we want to search for 2 in the array";      cout << "\n So, we first sort the array";      sort(a, a + 10);      cout << "\n\n The array after sorting is : ";      show(a, asize);      cout << "\n\nNow, we do the binary search";      if (binary\_search(a, a + 10, 2))         cout << "\nElement found in the array";      else         cout << "\nElement not found in the array";        cout << "\n\nNow, say we want to search for 10";       if (binary\_search(a, a + 10, 10))         cout << "\nElement found in the array";      else         cout << "\nElement not found in the array";        return 0;    } |

Run on IDE

The output of the above program is :

The array is : 1 5 8 9

6 7 3 4 2 0

Let's say we want to search for 2 in the array

So, we first sort the array

The array after sorting is : 0 1 2

3 4 5 6 7 8 9

Now, we do the binary search

Element found in the array

Now, say we want to search for 10

Element not found in the array

# **std::bsearch in C++**

std::bsearch searches for an element in a sorted array. Finds an element equal to element pointed to by key in an array pointed to by ptr.  
If the array contains several elements that comp would indicate as equal to the element searched for, then it is unspecified which element the function will return as the result.  
**Syntax :**

**void\* bsearch( const void\* key, const void\* ptr, std::size\_t count,**

**std::size\_t size, \* comp );**

**Parameters :**

key - element to be found

ptr - pointer to the array to examine

count - number of element in the array

size - size of each element in the array in bytes

comp - comparison function which returns ?a negative integer value if

the first argument is less than the second,

a positive integer value if the first argument is greater than the second

and zero if the arguments are equal.

**Return value :**

Pointer to the found element or null pointer if the element has not been found.

**Implementing the binary predicate comp :**

|  |
| --- |
| // Binary predicate which returns 0 if numbers found equal  int comp(int\* a, int\* b)  {        if (\*a < \*b)          return -1;        else if (\*a > \*b)          return 1;        // elements found equal      else          return 0;  } |

**Implementation**

|  |
| --- |
| // CPP program to implement  // std::bsearch  #include <bits/stdc++.h>    // Binary predicate  int compare(const void\* ap, const void\* bp)  {      // Typecasting      const int\* a = (int\*)ap;      const int\* b = (int\*)bp;        if (\*a < \*b)          return -1;      else if (\*a > \*b)          return 1;      else          return 0;  }    // Driver code  int main()  {      // Given array      int arr[] = { 1, 2, 3, 4, 5, 6, 7, 8 };        // Size of array      int ARR\_SIZE = sizeof(arr) / sizeof(arr[0]);        // Element to be found      int key1 = 4;        // Calling std::bsearch      // Typecasting the returned pointer to int      int\* p1 = (int\*)std::bsearch(&key1, arr, ARR\_SIZE, sizeof(arr[0]), compare);        // If non-zero value is returned, key is found      if (p1)          std::cout << key1 << " found at position " << (p1 - arr) << '\n';      else          std::cout << key1 << " not found\n";        // Element to be found      int key2 = 9;        // Calling std::bsearch      // Typecasting the returned pointer to int      int\* p2 = (int\*)std::bsearch(&key2, arr, ARR\_SIZE, sizeof(arr[0]), compare);        // If non-zero value is returned, key is found      if (p2)          std::cout << key2 << " found at position " << (p2 - arr) << '\n';      else          std::cout << key2 << " not found\n";  } |

Run on IDE

Output:

4 found at position 3

9 not found

**Where to use :**Binary search can be used on sorted data where a key is to be found. It can be used in cases like computing frequency of a key in a sorted list.

**Why Binary Search?**  
Binary search is much more effective than linear search because it halves the search space at each step. This is not significant for our array of length 9. Here, linear search takes at most 9 steps and binary search takes at most 4 steps. But consider an array with 1000 elements, here linear search takes at most 1000 steps, while binary search takes at most 10 steps.  
For 1 billion elements, binary search will find our key in at most 30 steps.

# **Pair in C++ Standard Template Library (STL)**

The pair container is a simple container defined in **<utility>** header consisting of two data elements or objects.

* The first element is referenced as ‘first’ and the second element as ‘second’ and the order is fixed (first, second).
* Pair is used to combine together two values which may be different in type. Pair provides a way to store two heterogeneous objects as a single unit.
* Pair can be assigned, copied and compared. The array of objects allocated in a map or hash\_map are of type ‘pair’ by default in which all the ‘first’ elements are unique keys associated with their ‘second’ value objects.
* To access the elements, we use variable name followed by dot operator followed by the keyword first or second.

**Syntax :**

pair (data\_type1, data\_type2) Pair\_name;

|  |
| --- |
| //CPP program to illustrate pair STL  #include <iostream>  #include <utility>  using namespace std;    int main()  {      pair <int, char> PAIR1 ;        PAIR1.first = 100;      PAIR1.second = 'G' ;        cout << PAIR1.first << " " ;      cout << PAIR1.second << endl ;        return 0;  } |

Run on IDE

Output:

100 G

**Initializing a pair**

We can also initialize a pair.  
Syntax :

pair (data\_type1, data\_type2) Pair\_name (value1, value2) ;

Different ways to initialize pair:

pair g1; //default

pair g2(1, 'a'); //initialized, different data type

pair g3(1, 10); //initialized, same data type

pair g4(g3); //copy of g3

Another way to initialize a pair is by using the make\_pair() function.

g2 = make\_pair(1, 'a');

|  |
| --- |
| //CPP program to illustrate Initializing of pair STL  #include <iostream>  #include <utility>  using namespace std;    int main()  {      pair <string,double> PAIR2 ("GeeksForGeeks", 1.23);        cout << PAIR2.first << " " ;      cout << PAIR2.second << endl ;    return 0;  } |

Run on IDE

Output:

GeeksForGeeks 1.23

**Note:** If not initialized, the first value of the pair gets automatically initialized.

|  |
| --- |
| //CPP program to illustrate auto-initializing of pair STL  #include <iostream>  #include <utility>    using namespace std;    int main()  {      pair <int, double> PAIR1 ;      pair <string, char> PAIR2 ;        cout << PAIR1.first ;  //it is initialised to 0      cout << PAIR1.second ; //it is initialised to 0        cout << " ";        cout << PAIR2.first ;  //it prints nothing i.e NULL      cout << PAIR2.second ; //it prints nothing i.e NULL        return 0;  } |

Run on IDE

Output:

00

**Member Functions**

1. **make\_pair()** : This template function allows to create a value pair without writing the types explicitly.  
   Syntax :

Pair\_name = make\_pair (value1,value2);

|  |
| --- |
| #include <iostream>  #include <utility>  using namespace std;    int main()  {      pair <int, char> PAIR1 ;      pair <string, double> PAIR2 ("GeeksForGeeks", 1.23) ;      pair <string, double> PAIR3 ;        PAIR1.first = 100;      PAIR1.second = 'G' ;        PAIR3 = make\_pair ("GeeksForGeeks is Best",4.56);        cout << PAIR1.first << " " ;      cout << PAIR1.second << endl ;        cout << PAIR2.first << " " ;      cout << PAIR2.second << endl ;        cout << PAIR3.first << " " ;      cout << PAIR3.second << endl ;        return 0;  } |

Run on IDE

Output:

100 G

GeeksForGeeks 1.23

GeeksForGeeks is Best 4.56

1. **operators(=, ==, !=, >=, <=) :** We can use operators with pairs as well.
   * **using equal(=) :** It assigns new object for a pair object.  
     Syntax :

pair& operator= (const pair& pr);

This Assigns pr as the new content for the pair object. The first value is assigned the first value of pr and the second value is assigned the second value of pr .

* + **Comparison (==) operator with pair :** For given two pairs say pair1 and pair2, the comparison operator compares the first value and second value of those two pairs i.e. if pair1.first is equal to pair2.first or not AND if pair1.second is equal to pair2.second or not .
  + **Not equal (!=) operator with pair :** For given two pairs say pair1 and pair2, the != operator compares the first values of those two pairs i.e. if pair1.first is equal to pair2.first or not, if they are equal then it checks the second values of both.
  + **Logical( >=, <= )operators with pair :** For given two pairs say pair1 and pair2, the =, >, can be used with pairs as well.

.

|  |
| --- |
| //CPP code to illustrate operators in pair  #include <iostream>  #include<utility>  using namespace std;    int main()  {      pair<int, int>pair1 = make\_pair(1, 12);      pair<int, int>pair2 = make\_pair(9, 12);          cout << (pair1 == pair2) << endl;      cout << (pair1 != pair2) << endl;      cout << (pair1 >= pair2) << endl;      cout << (pair1 <= pair2) << endl;      cout << (pair1 > pair2) << endl;      cout << (pair1 < pair2) << endl;        return 0;  } |

Run on IDE

Output:

0

1

0

1

0

1

1. **swap :**This function swaps the contents of one pair object with the contents of another pair object. The pairs must be of same type.  
   Syntax :

pair1.swap(pair2) ;

For two given pairs say pair1 and pair2 of same type, swap function will swap the pair1.first with pair2.first and pair1.second with pair2.second.

|  |
| --- |
| #include <iostream>  #include<utility>    using namespace std;    int main()  {      pair<char, int>pair1 = make\_pair('A', 1);      pair<char, int>pair2 = make\_pair('B', 2);        cout << "Before swapping:\n " ;      cout << "Contents of pair1 = " << pair1.first << " " << pair1.second ;      cout << "Contents of pair2 = " << pair2.first << " " << pair2.second ;      pair1.swap(pair2);        cout << "\nAfter swapping:\n ";      cout << "Contents of pair1 = " << pair1.first << " " << pair1.second ;      cout << "Contents of pair2 = " << pair2.first << " " << pair2.second ;        return 0;  } |

Run on IDE

Output:

Before swapping:

Contents of pair1 = (A, 1)

Contents of pair2 = (B, 2)

After swapping:

Contents of pair1 = (B, 2)

Contents of pair2 = (A, 1)

# **Vector in C++ STL**

# Vector

Vectors are same as dynamic arrays with the ability to resize itself automatically when an element is inserted or deleted, with their storage being handled automatically by the container. Vector elements are placed in contiguous storage so that they can be accessed and traversed using iterators. In vectors, data is inserted at the end. Inserting at the end takes differential time, as sometimes there may be a need of extending the array.Removing the last element takes only constant time, because no resizing happens. Inserting and erasing at the beginning or in the middle is linear in time.

Certain functions are associated with vector :  
**Iterators**  
1. begin() – Returns an iterator pointing to the first element in the vector  
2. end() – Returns an iterator pointing to the theoretical element that follows last element in the vector  
3. rbegin() – Returns a reverse iterator pointing to the last element in the vector (reverse beginning). It moves from last to first element  
4. rend() – Returns a reverse iterator pointing to the theoretical element preceding the first element in the vector (considered as reverse end)

|  |
| --- |
| #include <iostream>  #include <vector>    using namespace std;    int main()  {      vector <int> g1;      vector <int> :: iterator i;      vector <int> :: reverse\_iterator ir;        for (int i = 1; i <= 5; i++)          g1.push\_back(i);        cout << "Output of begin and end\t:\t";      for (i = g1.begin(); i != g1.end(); ++i)          cout << \*i << '\t';        cout << endl << endl;      cout << "Output of rbegin and rend\t:\t";      for (ir = g1.rbegin(); ir != g1.rend(); ++ir)          cout << '\t' << \*ir;        return 0;    } |

Run on IDE

The output of the above program is :

Output of begin and end : 1 2 3 4 5

Output of rbegin and rend : 5 4 3 2 1

**Capacity**  
1. size() – Returns the number of elements in the vector  
2. max\_size() – Returns the maximum number of elements that the vector can hold  
3. capacity() – Returns the size of the storage space currently allocated to the vector expressed as number of elements  
4. resize(size\_type g) – Resizes the container so that it contains ‘g’ elements  
5. empty() – Returns whether the container is empty

|  |
| --- |
| #include <iostream>  #include <vector>    using namespace std;    int main()  {      vector <int> g1;        for (int i = 1; i <= 5; i++)          g1.push\_back(i);        cout << "Size : " << g1.size();      cout << "\nCapacity : " << g1.capacity();      cout << "\nMax\_Size : " << g1.max\_size();        return 0;    } |

Run on IDE

The output of the above program is :

Size : 5

Capacity : 8

Max\_Size : 4611686018427387903

**Accessing the elements**  
1. reference operator [g] – Returns a reference to the element at position ‘g’ in the vector  
2. at(g) – Returns a reference to the element at position ‘g’ in the vector  
3. front() – Returns a reference to the first element in the vector  
4. back() – Returns a reference to the last element in the vector

|  |
| --- |
| #include <iostream>  #include <vector>  #include <string>  using namespace std;    int main()  {      vector <int> g1;        for (int i = 1; i <= 10; i++)          g1.push\_back(i \* 10);        cout << "Reference operator [g] : g1[2] = " << g1[2];      cout << endl;      cout << "at : g1.at(4) = " << g1.at(4);      cout << endl;      cout << "front() : g1.front() = " << g1.front();      cout << endl;      cout << "back() : g1.back() = " << g1.back();      cout << endl;        return 0;    } |

Run on IDE

The output of the above program is :

Reference operator [g] : g1[2] = 30

at : g1.at(4) = 50

front() : g1.front() = 10

back() : g1.back() = 100

# **Ways to copy a vector in C++**

In case of arrays, there is no much choice to copy an array into other, other than iterative method i.e running a loop to copy each element at respective index. But Vector classes has more than one methods to copy entire vector into other in easier ways.There are basically two types of copying :-

**Method 1 : Iterative method.**  
This method is a general method to copy, in this method a loop is used to push\_back() the old vector elements into new vector.They are deeply copied

|  |
| --- |
| // C++ code to demonstrate copy of vector  // by iterative method.  #include<iostream>  #include<vector>  using namespace std;    int main()  {      // Initializing vector with values      vector<int> vect1{1, 2, 3, 4};        // Declaring new vector      vector<int> vect2;        // A loop to copy elements of      // old vector into new vector      // by Iterative method      for (int i=0; i<vect1.size(); i++)          vect2.push\_back(vect1[i]);        cout << "Old vector elements are : ";      for (int i=0; i<vect1.size(); i++)          cout << vect1[i] << " ";      cout << endl;        cout << "New vector elements are : ";      for (int i=0; i<vect2.size(); i++)          cout << vect2[i] << " ";      cout<< endl;        // Changing value of vector to show that a new      // copy is created.      vect1[0] = 2;        cout << "The first element of old vector is :";      cout << vect1[0] << endl;      cout << "The first element of new vector is :";      cout << vect2[0] <<endl;        return 0;  } |

Run on IDE

Output:

Old vector elements are : 1 2 3 4

New vector elements are : 1 2 3 4

The first element of old vector is : 2

The first element of new vector is : 1

In the above code, on changing the value at one vector did not alter value at other vector, hence they are not allocated at same address, hence deep copy.  
**Method 2 : By**[**assignment “=” operator**](http://quiz.geeksforgeeks.org/operators-in-c-set-1-arithmetic-operators/)**.**  
Simply assigning the new vector to old one copies the vector. This way of assignment is not possible in case of arrays.

|  |
| --- |
| // C++ code to demonstrate copy of vector  // by iterative method.  #include<iostream>  #include<vector>  using namespace std;    int main()  {      // Initializing vector with values      vector<int> vect1{1, 2, 3, 4};        // Declaring new vector      vector<int> vect2;        // Using assignment operator to copy one      // vector to other      vect2 = vect1;        cout << "Old vector elements are : ";      for (int i=0; i<vect1.size(); i++)          cout << vect1[i] << " ";      cout << endl;        cout << "New vector elements are : ";      for (int i=0; i<vect2.size(); i++)          cout << vect2[i] << " ";      cout<< endl;        // Changing value of vector to show that a new      // copy is created.      vect1[0] = 2;        cout << "The first element of old vector is :";      cout << vect1[0] << endl;      cout << "The first element of new vector is :";      cout << vect2[0] <<endl;        return 0;  } |

Run on IDE

Output:

Old vector elements are : 1 2 3 4

New vector elements are : 1 2 3 4

The first element of old vector is : 2

The first element of new vector is : 1

**Method 3 : By passing vector as constructor.** At the time of declaration of vector, passing an old initialized vector, copies the elements of passed vector into the newly declared vector. They are deeply copied.

|  |
| --- |
| // C++ code to demonstrate copy of vector  // by constructor method.  #include<bits/stdc++.h>  using namespace std;    int main()  {      // Initializing vector with values      vector<int> vect1{1, 2, 3, 4};        // Declaring new vector and copying      // element of old vector      // constructor method, Deep copy      vector<int> vect2(vect1);        cout << "Old vector elements are : ";      for (int i=0; i<vect1.size(); i++)          cout << vect1[i] << " ";      cout << endl;        cout << "New vector elements are : ";      for (int i=0; i<vect2.size(); i++)          cout << vect2[i] << " ";      cout<< endl;        // Changing value of vector to show that a new      // copy is created.      vect1[0] = 2;        cout << "The first element of old vector is :";      cout << vect1[0] << endl;      cout << "The first element of new vector is :";      cout << vect2[0] <<endl;        return 0;  } |

Run on IDE

Output:

Old vector elements are : 1 2 3 4

New vector elements are : 1 2 3 4

The first element of old vector is :2

The first element of new vector is :1

**Method 4 : By using inbuilt functions**

|  |
| --- |
| **copy(first\_iterator\_o, last\_iterator\_o, back\_inserter())** :- This is another way to copy old vector into new one. This function takes 3 arguments, first, the first iterator of old vector, second, the last iterator of old vector and third is back\_inserter function to insert values from back. This also generated a deep copy.  // C++ code to demonstrate copy of vector  // by assign() and copy().  #include<iostream>  #include<vector> // for vector  #include<algorithm> // for copy() and assign()  #include<iterator> // for back\_inserter  using namespace std;  int main()  {      // Initializing vector with values      vector<int> vect1{1, 2, 3, 4};        // Declaring new vector      vector<int> vect2;        // Copying vector by copy function      copy(vect1.begin(), vect1.end(), back\_inserter(vect2));        cout << "Old vector elements are : ";      for (int i=0; i<vect1.size(); i++)          cout << vect1[i] << " ";      cout << endl;        cout << "New vector elements are : ";      for (int i=0; i<vect2.size(); i++)          cout << vect2[i] << " ";      cout<< endl;        // Changing value of vector to show that a new      // copy is created.      vect1[0] = 2;        cout << "The first element of old vector is :";      cout << vect1[0] << endl;      cout << "The first element of new vector is :";      cout << vect2[0] <<endl;        return 0;  } |

* Run on IDE
* Output :
* Old vector elements are : 1 2 3 4
* New vector elements are : 1 2 3 4
* The first element of old vector is :2
* The first element of new vector is :1
* **assign(first\_iterator\_o, last\_iterator\_o)** :- This method assigns the same values to new vector as old one. This takes 2 arguments, first iterator to old vector and last iterator to old vector.This generates a deep copy.

|  |
| --- |
| // C++ code to demonstrate copy of vector  // by assign()  #include<iostream>  #include<vector> // for vector  #include<algorithm> // for copy() and assign()  #include<iterator> // for back\_inserter  using namespace std;    int main()  {      // Initializing vector with values      vector<int> vect1{1, 2, 3, 4};        // Declaring another vector      vector<int> vect2;        // Copying vector by assign function      vect2.assign(vect1.begin(), vect1.end());        cout << "Old vector elements are : ";      for (int i=0; i<vect1.size(); i++)          cout << vect1[i] << " ";      cout << endl;        cout << "New vector elements are : ";      for (int i=0; i<vect2.size(); i++)          cout << vect2[i] << " ";      cout<< endl;        // Changing value of vector to show that a new      // copy is created.      vect1[0] = 2;        cout << "The first element of old vector is :";      cout << vect1[0] << endl;      cout << "The first element of new vector is :";      cout << vect2[0] <<endl;        return 0;  } |

* Run on IDE
* Output:
* Old vector elements are : 1 2 3 4
* New vector elements are : 1 2 3 4
* The first element of old vector is :2
* The first element of new vector is :1

# **Modifiers for Vector in C++ STL**

Click here for [Set 1](http://quiz.geeksforgeeks.org/vector-sequence-containers-the-c-standard-template-library-stl-set-1) of Vectors.

**Modifiers**  
1.1 assign(input\_iterator first, input\_iterator last) – Assigns new content to vector and resize  
1.2 assign(size\_type n, const value\_type g) – Assigns new content to vector and resize

|  |
| --- |
| #include <iostream>  #include <vector>    using namespace std;    int main()  {      vector <int> g1;      vector <int> g2;      vector <int> g3;        g1.assign(5, 10);   // 5 elements with value 10 each        vector <int> :: iterator it;      it = g1.begin() + 1;        g2.assign(it, g1.end() - 1); // the 3 middle values of g1        int gquiz[] = {1, 2};      g3.assign(gquiz, gquiz + 2);   // assigning from array        cout << "Size of g1: " << int(g1.size()) << '\n';      cout << "Size of g2: " << int(g2.size()) << '\n';      cout << "Size of g3: " << int(g3.size()) << '\n';      return 0;  } |

Run on IDE

The output of the above program is :

Size of g1: 5

Size of g2: 3

Size of g3: 2

2. push\_back(const value\_type g) – Adds a new element ‘g’ at the end of the vector and increases the vector container size by 1  
3. pop\_back() – Removes the element at the end of the vector, i.e., the last element and decreases the vector container size by 1

|  |
| --- |
| #include <iostream>  #include <vector>    using namespace std;    int main()  {    vector <int> gquiz;    int sum = 0;    gquiz.push\_back(10);    gquiz.push\_back(20);    gquiz.push\_back(30);      while (!gquiz.empty())    {      sum += gquiz.back();      gquiz.pop\_back();    }      cout << "The sum of the elements of gquiz is :  "         << sum << '\n';      return 0;  } |

Run on IDE

The output of the above program is :

The sum of the elements of gquiz is : 60

4.1 insert(const\_iterator q, const value\_type g) – Adds element ‘g’ before the element referenced by iterator ‘q’ and returns an iterator that points to the newly added element  
4.2insert(const\_iterator q, size\_type n, const value\_type g) – Adds ‘n’ elements each with value ‘g’ before the element currently referenced by iterator ‘q’ and returns an iterator that points to the first of the newly added elements  
4.3 insert(const\_iterator q, InputIterator first, InputIterator last) – Adds a range of elements starting from first to last, the elements being inserted before the position currently referred by ‘q’

|  |
| --- |
| #include <iostream>  #include <vector>    using namespace std;    int main()  {      vector <int> gquiz1(3, 10);      vector <int> :: iterator it;        it = gquiz1.begin();      it = gquiz1.insert(it, 20);        gquiz1.insert(it, 2, 30);        it = gquiz1.begin();        vector <int> gquiz2(2, 40);      gquiz1.insert(it + 2, gquiz2.begin(), gquiz2.end());        int gq [] = {50, 60, 70};      gquiz1.insert(gquiz1.begin(), gq, gq + 3);        cout << "gquiz1 contains : ";      for (it = gquiz1.begin(); it < gquiz1.end(); it++)          cout << \*it << '\t';        return 0;  } |

Run on IDE

The output of the above program is :

gquiz1 contains : 50 60 70 30 30

40 40 20 10 10 10

5.1 erase(const\_iterator q) – Deletes the element referred by ‘q’ and returns an iterator to the element followed by the deleted element  
5.2 erase(const\_iterator first, const\_iterator last) – Deletes the elements in the range first to last, with the first iterator included in the range and the last iterator not included, and returns an iterator to the element followed by the last deleted element

|  |
| --- |
| #include <iostream>  #include <vector>  using namespace std;    int main ()  {      vector <int> gquiz;        for (int i = 1; i <= 10; i++)          gquiz.push\_back(i \* 2);        // erase the 5th element      gquiz.erase(gquiz.begin() + 4);        // erase the first 5 elements:      gquiz.erase(gquiz.begin(), gquiz.begin() + 5);        cout << "gquiz contains :";      for (int i = 0; i < gquiz.size(); ++i)          cout << gquiz[i] << '\t';        return 0;  } |

Run on IDE

The output of the above program is :

gquiz contains :14 16 18 20

6. swap(vector q, vector r) – Swaps the contents of ‘q’ and ‘r’  
7. clear() – Removes all elements from the vector

|  |
| --- |
| #include <iostream>  #include <vector>    using namespace std;    int main()  {      vector <int> gquiz1;      vector <int> gquiz2;      vector <int> :: iterator i;        gquiz1.push\_back(10);      gquiz1.push\_back(20);        gquiz2.push\_back(30);      gquiz2.push\_back(40);        cout << "Before Swapping, \n"           <<"Contents of vector gquiz1 : ";        for (i = gquiz1.begin(); i != gquiz1.end(); ++i)          cout << \*i << '\t';        cout << "\nContents of vector gquiz2 : ";      for (i = gquiz2.begin(); i != gquiz2.end(); ++i)          cout << \*i << '\t';        swap(gquiz1, gquiz2);        cout << "\n\nAfter Swapping, \n";      cout << "Contents of vector gquiz1 : ";      for (i = gquiz1.begin(); i != gquiz1.end(); ++i)          cout << \*i << '\t';        cout << "\nContents of vector gquiz2 : ";      for (i = gquiz2.begin(); i != gquiz2.end(); ++i)          cout << \*i << '\t';        cout << "\n\nNow, we clear() and then add "           << "an element 1000 to vector gquiz1 : ";      gquiz1.clear();      gquiz1.push\_back(1000);      cout << gquiz1.front();        return 0;  } |

Run on IDE

The output of the above program is :

Before Swapping,

Contents of vector gquiz1 : 10 20

Contents of vector gquiz2 : 30 40

After Swapping,

Contents of vector gquiz1 : 30 40

Contents of vector gquiz2 : 10 20

Now, we clear() and then add an element 1000 to vector gquiz1 : 1000

# **Passing vector to a function in C++**

When we [pass an array to a function](https://www.geeksforgeeks.org/how-arrays-are-passed-to-functions-in-cc/), a pointer is actually passed.

When a [vector](https://www.geeksforgeeks.org/vector-in-cpp-stl/) is passed to a function, a copy of the vector is created. For example, we can see below program, changes made inside the function are not reflected outside because function has a copy.

|  |
| --- |
| // C++ program to demonstrate that when vectors  // are passed to functions without &, a copy is  // created.  #include<bits/stdc++.h>  using namespace std;    // The vect here is a copy of vect in main()  void func(vector<int> vect)  {     vect.push\_back(30);  }    int main()  {      vector<int> vect;      vect.push\_back(10);      vect.push\_back(20);        func(vect);        // vect remains unchanged after function      // call      for (int i=0; i<vect.size(); i++)         cout << vect[i] << " ";        return 0;  } |

Run on IDE

Output:

10 20

The above style of passing might also take a lot of time in cases of large vectors. So it is a good idea to pass by reference.

|  |
| --- |
| // C++ program to demonstrate how vectors  // can be passed by reference.  #include<bits/stdc++.h>  using namespace std;    // The vect is passed by reference and changes  // made here reflect in main()  void func(vector<int> &vect)  {     vect.push\_back(30);  }    int main()  {      vector<int> vect;      vect.push\_back(10);      vect.push\_back(20);        func(vect);        for (int i=0; i<vect.size(); i++)         cout << vect[i] << " ";        return 0;  } |

Run on IDE

Output:

10 20 30

If we do not want a function to modify a vector, we can pass it as a const reference.

|  |
| --- |
| // C++ program to demonstrate how vectors  // can be passed by reference with modifications  // restricted.  #include<bits/stdc++.h>  using namespace std;    // The vect is passed by constant reference  // and cannot be changed by this function.  void func(const vector<int> &vect)  {      // vect.push\_back(30);  // Uncommenting this line would                               // below error      // "prog.cpp: In function 'void func(const std::vector<int>&)':      // prog.cpp:9:18: error: passing 'const std::vector<int>'      // as 'this' argument discards qualifiers [-fpermissive]"        for (int i=0; i<vect.size(); i++)         cout << vect[i] << " ";  }    int main()  {      vector<int> vect;      vect.push\_back(10);      vect.push\_back(20);        func(vect);        return 0;  } |

Run on IDE

Output:

10 20

# **List in C++ Standard Template Library (STL)**

# List

Lists are sequence containers that allow non-contiguous memory allocation. As compared to vector, list has slow traversal, but once a position has been found, insertion and deletion are quick. Normally, when we say a List, we talk about doubly linked list. For implementing a singly linked list, we use forward list.

Functions used with List :  
front() – Returns reference to the first element in the list  
back() – Returns reference to the last element in the list  
push\_front(g) – Adds a new element ‘g’ at the beginning of the list  
push\_back(g) – Adds a new element ‘g’ at the end of the list  
pop\_front() – Removes the first element of the list, and reduces size of the list by 1  
pop\_back() – Removes the last element of the list, and reduces size of the list by 1  
begin() – Returns an iterator pointing to the first element of the list  
end() – Returns an iterator pointing to the theoretical last element which follows the last element  
empty() – Returns whether the list is empty(1) or not(0)  
insert() – Inserts new elements in the list before the element at a specified position  
erase() – Removes a single element or a range of elements from the list  
assign() – Assigns new elements to list by replacing current elements and resizes the list  
remove() – Removes all the elements from the list, which are equal to given element  
reverse() – Reverses the list  
size() – Returns the number of elements in the list  
sort() – Sorts the list in increasing order

|  |
| --- |
| #include <iostream>  #include <list>  #include <iterator>  using namespace std;    //function for printing the elements in a list  void showlist(list <int> g)  {      list <int> :: iterator it;      for(it = g.begin(); it != g.end(); ++it)          cout << '\t' << \*it;      cout << '\n';  }    int main()  {      list <int> gqlist1, gqlist2;      for (int i = 0; i < 10; ++i)      {          gqlist1.push\_back(i \* 2);          gqlist2.push\_front(i \* 3);      }      cout << "\nList 1 (gqlist1) is : ";      showlist(gqlist1);        cout << "\nList 2 (gqlist2) is : ";      showlist(gqlist2);        cout << "\ngqlist1.front() : " << gqlist1.front();      cout << "\ngqlist1.back() : " << gqlist1.back();        cout << "\ngqlist1.pop\_front() : ";      gqlist1.pop\_front();      showlist(gqlist1);        cout << "\ngqlist2.pop\_back() : ";      gqlist2.pop\_back();      showlist(gqlist2);        cout << "\ngqlist1.reverse() : ";      gqlist1.reverse();      showlist(gqlist1);        cout << "\ngqlist2.sort(): ";      gqlist2.sort();      showlist(gqlist2);        return 0;    } |

Run on IDE

The output of the above program is :

List 1 (gqlist1) is : 0 2 4 6

8 10 12 14 16 18

List 2 (gqlist2) is : 27 24 21 18

15 12 9 6 3 0

gqlist1.front() : 0

gqlist1.back() : 18

gqlist1.pop\_front() : 2 4 6 8

10 12 14 16 18

gqlist2.pop\_back() : 27 24 21 18

15 12 9 6 3

gqlist1.reverse() : 18 16 14 12

10 8 6 4 2

gqlist2.sort(): 3 6 9 12

15 18 21 24 27

# **List in C++ | Set 2 (Some Useful Functions)**

We have discussed List and some of its functions in the following article:

[List : Sequence Containers](http://quiz.geeksforgeeks.org/list-sequence-containers-the-c-standard-template-library-stl/)

Some more useful functions are discussed in this article

**1. emplace(position, value)** :- This function is used to **insert** an element at the **position** specified.

**2. emplace\_back(value)** :- This function adds **value at end** of list. It is different from push\_back() by the fact that it directly creates element at position whereas push\_back() first makes a temporary copy and copies from there. emplace\_back() is faster in implementation than push\_back() in most situations.

**3. emplace\_front** :- This function adds **value at beginning** of list. It is different from push\_front() by the fact that it directly creates element at position whereas push\_front() first makes a temporary copy and copies from there. emplace\_front() is faster in implementation than push\_front() in most situations.

|  |
| --- |
| // C++ code to demonstrate the working of  // emplace(), emplace\_front() and emplace\_back()  #include<iostream>  #include<list> // for list functions  using namespace std;  int main()  {      // Declaring a list      list<int> gqlist;        // Initialising list iterator      list<int>::iterator it= gqlist.begin();        // Entering list element using emplace\_back()      for (int i=1; i<=5; i++)      gqlist.emplace\_back(i);        // Displaying list elements      cout << "List after emplace\_back operation is : ";      for (int &x : gqlist) cout << x << " ";      cout << endl;        // Entering list element using emplace\_front()      for (int i=10; i<=50; i+=10)      gqlist.emplace\_front(i);        // Displaying list elements      cout << "List after emplace\_front operation is : ";      for (int &x : gqlist) cout << x << " ";      cout << endl;        // using advance() to advance iterator position      advance(it,2);         // inserting element at 2nd position using emplace()      gqlist.emplace(it,100);         // Displaying list elements      cout << "List after emplace operation is : ";      for (int &x : gqlist) cout << x << " ";      cout << endl;        return 0;    } |

Run on IDE

Output:

List after emplace\_back operation is : 1 2 3 4 5

List after emplace\_front operation is : 50 40 30 20 10 1 2 3 4 5

List after emplace operation is : 50 100 40 30 20 10 1 2 3 4 5

**4. merge(list2)** :- This function is used to **merge list2 with list1**. If both the lists are in sorted order, then the resulting list is also sorted.

**5. remove\_if(condition)** :- This function **removes the element** from list on the **basis of condition**given in its argument.

**The** remove\_if() function is used to remove all the values from the list that correspond true to the *predicate* or condition given as parameter to the function. The function iterates through every member of the list container and removes all the element that return true for the predicate.

**Syntax :**

***listname*.remove\_if(*predicate*)**

**Parameters :**

The predicate in the form of aa function pointer

or function object is passed as the parameter.

**Result :**

Removes all the elements of the container

which return true for the predicate.

Examples:

Input : list list{1, 2, 3, 4, 5};

list.remove\_if(odd);

Output : 2, 4

Input : list list{1, 2, 2, 2, 5, 6, 7};

list.remove\_if(even);

Output : 1, 5, 7

**Errors and Exceptions**

1. Shows no exception throw guarantee if the predicate function feature doesn’t throw any exception.

|  |
| --- |
| // CPP program to illustrate  // Implementation of remove\_if() function  #include <iostream>  #include <list>  using namespace std;    // Predicate implemented as a function  bool even(const int& value) { return (value % 2) == 0; }    // Main function  int main()  {      list<int> mylist{ 1, 2, 2, 2, 5, 6, 7 };      mylist.remove\_if(even);      for (auto it = mylist.begin(); it != mylist.end(); ++it)          cout << ' ' << \*it;  } |

Run on IDE

Output:

1 5 7

**Application :** Given a list of integers, remove all the prime numbers from the list and print the list.

Input : 2, 4, 6, 7, 9, 11, 13

Output : 4, 6, 9

|  |
| --- |
| // CPP program to illustrate  // Application of remove\_if() function  #include <iostream>  #include <list>  using namespace std;    // Predicate implemented as a function  bool prime(const int& value)  {      int i;      for (i = 2; i < value; i++) {          if (value % i == 0) {              return false;              ;              break;          }      }      if (value == i) {          return true;          ;      }  }    // Main function  int main()  {      list<int> mylist{ 2, 4, 6, 7, 9, 11, 13 };      mylist.remove\_if(prime);      for (auto it = mylist.begin(); it != mylist.end(); ++it)          cout << ' ' << \*it;  } |

Run on IDE

Output:

4 6 9

|  |
| --- |
| // C++ code to demonstrate the working of  // merge() and remove\_if()  #include<iostream>  #include<list> // for list functions  using namespace std;  int main()  {      // Initializing list1      list<int> gqlist1 = {1, 2, 3};        // Initializing list2      list<int> gqlist2 = {2, 4, 6};        // using merge() to merge list1 with list2      gqlist1.merge(gqlist2);        // Displaying list elements      cout << "list1 after merge operation is : ";      for (int &x : gqlist1) cout << x << " ";      cout << endl;        // using remove\_if() to remove odd elements      // removes 1 and 3      gqlist1.remove\_if([](int x){return x%2!=0;});        // Displaying list elements      cout << "list1 after remove\_if operation is : ";      for (int &x : gqlist1) cout << x << " ";      cout << endl;        return 0;    } |

Run on IDE

Output:

list1 after merge operation is : 1 2 2 3 4 6

list1 after remove\_if operation is : 2 2 4 6

**6. unique()** :- This function is used to **delete the repeated occurrences** of the number. List has to be**sorted** for this function to get executed.

**7. splice(position, list2)** :- This function is used to **transfer elements**from one list into another.

|  |
| --- |
| // C++ code to demonstrate the working of  // unique() and splice()  #include<iostream>  #include<list> // for list functions  using namespace std;  int main()  {      // Initializing list1      list<int> gqlist1 = {1, 1, 1, 2, 2, 3, 3, 4};        // Initializing list2      list<int> gqlist2 = {2, 4, 6};        // Initializing list1 iterator      list<int>::iterator it = gqlist1.begin();        // using advance() to increment iterator position      advance(it, 3);          // Displaying list elements      cout << "list1 before unique operation is : ";      for (int &x : gqlist1) cout << x << " ";      cout << endl;        // using unique() to remove repeating elements      gqlist1.unique();        // Displaying list elements      cout << "list1 after unique operation is : ";      for (int &x : gqlist1) cout << x << " ";      cout << endl << endl;        // using splice() to splice list2 in list1 at position it      // inserts list2 after 2nd position      gqlist1.splice(it, gqlist2);        // Displaying list elements      cout << "list1 after splice operation is : ";      for (int &x : gqlist1) cout << x << " ";      cout << endl;          return 0;    } |

Run on IDE

Output:

list1 before unique operation is : 1 1 1 2 2 3 3 4

list1 after unique operation is : 1 2 3 4

list1 after splice operation is : 1 2 4 6 2 3 4

**8. swap(list2)** :- This function is used to **swap one list element with other**.

|  |
| --- |
| // C++ code to demonstrate the working of  // swap()  #include<iostream>  #include<list> // for list functions  using namespace std;  int main()  {      // Initializing list1      list<int> gqlist1 = {1, 2, 3, 4};        // Initializing list1      list<int> gqlist2 = {2, 4, 6};         // Displaying list before swapping       cout << "The contents of 1st list "               "before swapping are : ";       for (int &x : gqlist1)           cout << x << " ";       cout << endl;       cout << "The contents of 2nd list "               "before swapping are : ";       for (int &x : gqlist2)           cout << x << " ";       cout << endl;         // Use of swap() to swap the list       gqlist1.swap(gqlist2);         // Displaying list after swapping       cout << "The contents of 1st list "               "after swapping are : ";       for (int &x : gqlist1)           cout << x << " ";       cout << endl;         cout << "The contents of 2nd list "               "after swapping are : ";       for (int &x : gqlist2)           cout << x << " ";       cout << endl;        return 0;    } |

Run on IDE

Output:

The contents of 1st list before swapping are : 1 2 3 4

The contents of 2nd list before swapping are : 2 4 6

The contents of 1st list after swapping are : 2 4 6

The contents of 2nd list after swapping are : 1 2 3 4

# **Forward List in C++ | Set 1 (Introduction and Important Functions)**

Forward list in STL implements singly linked list. Introduced from C++11, forward list are useful than other containers in insertion, removal and moving operations (like sort) and allows time constant insertion and removal of elements.

It differs from [list](http://quiz.geeksforgeeks.org/list-sequence-containers-the-c-standard-template-library-stl/) by the fact that forward list keeps track of location of only next element while list keeps track to both next and previous elements, thus increasing the storage space required to store each element. The drawback of forward list is that it cannot be iterated backwards and its individual elements cannot be accessed directly.

Forward List is preferred over list when only forward traversal is required (same as singly linked list is preferred over doubly linked list) as we can save space. Some example cases are, chaining in hashing, adjacency list representation of graph, etc.

**Operations on Forward List :**  
**1. assign()**:- This function is used to assign values to forward list, its another variant is used to assign repeated elements.

|  |
| --- |
| // C++ code to demonstrate forward list  // and assign()  #include<iostream>  #include<forward\_list>  using namespace std;    int main()  {      // Declaring forward list      forward\_list<int> flist1;        // Declaring another forward list      forward\_list<int> flist2;        // Assigning values using assign()      flist1.assign({1, 2, 3});        // Assigning repeating values using assign()      // 5 elements with value 10      flist2.assign(5, 10);        // Displaying forward lists      cout << "The elements of first forward list are : ";      for (int&a : flist1)          cout << a << " ";      cout << endl;        cout << "The elements of second forward list are : ";      for (int&b : flist2)          cout << b << " ";      cout << endl;        return 0;  } |

Run on IDE

Output:

The elements of first forward list are : 1 2 3

The elements of second forward list are : 10 10 10 10 10

**2. push\_front()** :- This function is used to insert the element at the first position on forward list. The value from this function is copied to the space before first element in the container. The size of forward list increases by 1.

**3. emplace\_front()** :- This function is similar to the previous function but in this no copying operation occurs, the element is created directly at the memory before the first element of the forward list.

**4. pop\_front()** :- This function is used to delete the first element of list.

|  |
| --- |
| // C++ code to demonstrate working of  // push\_front(), emplace\_front() and pop\_front()  #include<iostream>  #include<forward\_list>  using namespace std;    int main()  {      // Initializing forward list      forward\_list<int> flist = {10, 20, 30, 40, 50};        // Inserting value using push\_front()      // Inserts 60 at front      flist.push\_front(60);        // Displaying the forward list      cout << "The forward list after push\_front operation : ";      for (int&c : flist)          cout << c << " ";      cout << endl;        // Inserting value using emplace\_front()      // Inserts 70 at front      flist.emplace\_front(70);        // Displaying the forward list      cout << "The forward list after emplace\_front operation : ";      for (int&c : flist)         cout << c << " ";      cout << endl;        // Deleting first value using pop\_front()      // Pops 70      flist.pop\_front();        // Displaying the forward list      cout << "The forward list after pop\_front operation : ";      for (int&c : flist)          cout << c << " ";      cout << endl;        return 0;  } |

Run on IDE

Output:

The forward list after push\_front operation : 60 10 20 30 40 50

The forward list after emplace\_front operation : 70 60 10 20 30 40 50

The forward list after pop\_front operation : 60 10 20 30 40 50

**4. insert\_after()** This function gives us a choice to insert elements at any position in forward list. The arguments in this function are copied at the desired position.

**5. emplace\_after()** This function also does the same operation as above function but the elements are directly made without any copy operation.

**6. erase\_after()**This function is used to erase elements from a particular position in the forward list.

|  |
| --- |
| // C++ code to demonstrate working of  // insert\_after(), emplace\_after() and erase\_after()  #include<iostream>  #include<forward\_list>  using namespace std;    int main()  {      // Initializing forward list      forward\_list<int> flist = {10, 20, 30} ;        // Declaring a forward list iterator      forward\_list<int>::iterator ptr;        // Inserting value using insert\_after()      // starts insertion from second position      ptr =  flist.insert\_after(flist.begin(), {1, 2, 3});        // Displaying the forward list      cout << "The forward list after insert\_after operation : ";      for (int&c : flist)          cout << c << " ";      cout << endl;        // Inserting value using emplace\_after()      // inserts 2 after ptr      ptr = flist.emplace\_after(ptr,2);        // Displaying the forward list      cout << "The forward list after emplace\_after operation : ";      for (int&c : flist)          cout << c << " ";      cout << endl;        // Deleting value using erase.after Deleted 2      // after ptr      ptr = flist.erase\_after(ptr);        // Displaying the forward list      cout << "The forward list after erase\_after operation : ";      for (int&c : flist)          cout << c << " ";      cout << endl;        return 0;  } |

Run on IDE

Output:

The forward list after insert\_after operation : 10 1 2 3 20 30

The forward list after emplace\_after operation : 10 1 2 3 2 20 30

The forward list after erase\_after operation : 10 1 2 3 2 30

**7. remove()** :- This function removes the particular element from the forward list mentioned in its argument.

**8. remove\_if()** :- This function removes according to the condition in its argument.

|  |
| --- |
| // C++ code to demonstrate working of remove() and  // remove\_if()  #include<iostream>  #include<forward\_list>  using namespace std;    int main()  {      // Initializing forward list      forward\_list<int> flist = {10, 20, 30, 25, 40, 40};        // Removing element using remove()      // Removes all occurrences of 40      flist.remove(40);        // Displaying the forward list      cout << "The forward list after remove operation : ";      for (int&c : flist)          cout << c << " ";      cout << endl;        // Removing according to condition. Removes      // elements greater than 20. Removes 25 and 30      flist.remove\_if([](int x){ return x>20;});        // Displaying the forward list      cout << "The forward list after remove\_if operation : ";      for (int&c : flist)         cout << c << " ";      cout << endl;        return 0;    } |

Run on IDE

Output:

The forward list after remove operation : 10 20 30 25

The forward list after remove\_if operation : 10 20

**9. splice\_after()** :- This function transfers elements from one forward list to other.

|  |
| --- |
| // C++ code to demonstrate working of  // splice\_after()  #include<iostream>  #include<forward\_list> // for splice\_after()  using namespace std;    int main()  {      // Initializing forward list      forward\_list<int> flist1 = {10, 20, 30};        // Initializing second list      forward\_list<int> flist2 = {40, 50, 60};        // Shifting elements from first to second      // forward list after 1st position      flist2.splice\_after(flist2.begin(),flist1);        // Displaying the forward list      cout << "The forward list after splice\_after operation : ";      for (int&c : flist2)         cout << c << " ";      cout << endl;        return 0;  } |

Run on IDE

Output:

The forward list after splice\_after operation : 40 10 20 30 50 60

# **Auto keyword new meaning in C++**

auto keyword as in C programming language is deprecated in c++. It has a new meaning on c++

auto i = 3 ;implies i is int

auto d = 3.7; implies double

auto c =d; implies double

We cannot just write auto i;. Basically we let the compiler decide the type- this is the new meaning of auto keyword.

# **Forward List in C++ | Set 2 (Manipulating Functions)**

[Forward List in C++ | Set 1 (Introduction and Important Functions)](https://www.geeksforgeeks.org/forward-list-c-set-1-introduction-important-functions/)

More functions are discussed in this article

Some of the operations other than insertions and deletions that can be used in forward lists are as follows :

**1. merge()** :- This function is used to merge one forward list with other. If both the lists are sorted then the resulted list returned is also sorted.

**2. operator “=”**:- This operator copies one forward list into other. The copy made in this case is deep copy.

|  |
| --- |
| // C++ code to demonstrate the working of  // merge() and operator=  #include<iostream>  #include<forward\_list>  using namespace std;    int main()  {       // Initializing 1st forward list       forward\_list<int> flist1 = {1, 2, 3};         // Declaring 2nd forward list       forward\_list<int> flist2;         // Creating deep copy using "="       flist2 = flist1;         // Displaying flist2       cout << "The contents of 2nd forward list"               " after copy are : ";       for (int &x : flist2)           cout << x << " ";       cout << endl;         // Using merge() to merge both list in 1       flist1.merge(flist2);         // Displaying merged forward list       // Prints sorted list       cout << "The contents of forward list "               "after merge are : ";       for (int &x : flist1)          cout << x << " ";       cout << endl;         return 0;  } |

Run on IDE

Output:

The contents of 2nd forward list after copy are : 1 2 3

The contents of forward list after merge are : 1 1 2 2 3 3

**3. sort()** :- This function is used to sort the forward list.

**4. unique()** :- This function deletes the multiple occurrences of a number and returns a forward list with unique elements. The forward list should be sorted for this function to execute successfully.

|  |
| --- |
| // C++ code to demonstrate the working of  // sort() and unique()  #include<iostream>  #include<forward\_list> // for sort() and unique()  using namespace std;    int main()  {       // Initializing 1st forward list       forward\_list<int> flist1 = {1, 2, 3, 2, 3, 3, 1};         // Sorting the forward list using sort()       flist1.sort();         // Displaying sorted forward list       cout << "The contents of forward list after "               "sorting are : ";       for (int &x : flist1)           cout << x << " ";       cout << endl;         // Use of unique() to remove repeated occurrences       flist1.unique();         // Displaying forward list after using unique()       cout << "The contents of forward list after "               "unique operation are : ";       for (int &x : flist1)           cout << x << " ";       cout << endl;         return 0;  } |

Run on IDE

Output:

The contents of forward list after sorting are : 1 1 2 2 3 3 3

The contents of forward list after unique operation are : 1 2 3

**5. reverse()** :- This function is used to reverse the forward list.

**6. swap()** :- This function swaps the content of one forward list with other.

|  |
| --- |
| // C++ code to demonstrate the working of  // reverse() and swap()  #include<iostream>  #include<forward\_list> // for reverse() and swap()  using namespace std;  int main()  {       // Initializing 1st forward list       forward\_list<int> flist1 = {1, 2, 3,};         // Initializing 2nd forward list       forward\_list<int> flist2 = {4, 5, 6};         // Using reverse() to reverse 1st forward list       flist1.reverse();         // Displaying reversed forward list       cout << "The contents of forward list after"               " reversing are : ";       for (int &x : flist1)           cout << x << " ";       cout << endl << endl;         // Displaying forward list before swapping       cout << "The contents of 1st forward list "               "before swapping are : ";       for (int &x : flist1)           cout << x << " ";       cout << endl;       cout << "The contents of 2nd forward list "               "before swapping are : ";       for (int &x : flist2)           cout << x << " ";       cout << endl;         // Use of swap() to swap the list       flist1.swap(flist2);         // Displaying forward list after swapping       cout << "The contents of 1st forward list "               "after swapping are : ";       for (int &x : flist1)           cout << x << " ";       cout << endl;         cout << "The contents of 2nd forward list "               "after swapping are : ";       for (int &x : flist2)           cout << x << " ";       cout << endl;         return 0;  } |

Run on IDE

Output:

The contents of forward list after reversing are : 3 2 1

The contents of 1st forward list before swapping are : 3 2 1

The contents of 2nd forward list before swapping are : 4 5 6

The contents of 1st forward list after swapping are : 4 5 6

The contents of 2nd forward list after swapping are : 3 2 1

**7. clear()** :- This function clears the contents of forward list. After this function, the forward list becomes empty.

**8. empty()** :- This function returns true if the list is empty otherwise false.

|  |
| --- |
| // C++ code to demonstrate the working of  // clear() and empty()  #include<iostream>  #include<forward\_list> // for clear() and empty()  using namespace std;  int main()  {       // Initializing  forward list       forward\_list<int> flist1 = {1, 2, 3,};         // Displaying forward list before clearing       cout << "The contents of forward list  are : ";       for (int &x : flist1)           cout << x << " ";       cout << endl;         // Using clear() to clear the forward list       flist1.clear();         // Displaying reversed forward list       cout << "The contents of forward list after "            << "clearing are : ";       for (int &x : flist1)           cout << x << " ";       cout << endl;         // Checking if list is empty       flist1.empty() ? cout << "Forward list is empty" :                        cout << "Forward list is not empty";         return 0;  } |

Run on IDE

Output:

The contents of forward list are : 1 2 3

The contents of forward list after clearing are :

Forward list is empty

# **Virtual Functions and Runtime Polymorphism in C++ | Set 1 (Introduction)**

Consider the following simple program which is an example of runtime polymorphism.  
The main thing to note about the program is, derived class function is called using a base class pointer. The idea is, [virtual functions](https://www.geeksforgeeks.org/virtual-function-cpp/) are called according to the type of object pointed or referred, not according to the type of pointer or reference. In other words, virtual functions are resolved late, at runtime.

|  |
| --- |
| #include<iostream>  using namespace std;    class Base  {  public:      virtual void show() { cout<<" In Base \n"; }  };    class Derived: public Base  {  public:      void show() { cout<<"In Derived \n"; }  };    int main(void)  {      Base \*bp = new Derived;      bp->show();  // RUN-TIME POLYMORPHISM      return 0;  } |

Run on IDE

Output:

In Derived

**What is the use?**  
Virtual functions allow us to create a list of base class pointers and call methods of any of the derived classes without even knowing kind of derived class object. For example, consider a employee management software for an organization, let the code has a simple base class Employee , the class contains virtual functions like raiseSalary(), transfer(), promote(),.. etc. Different types of employees like Manager, Engineer, ..etc may have their own implementations of the virtual functions present in base class Employee. In our complete software, we just need to pass a list of employees everywhere and call appropriate functions without even knowing the type of employee. For example, we can easily raise salary of all employees by iterating through list of employees. Every type of employee may have its own logic in its class, we don’t need to worry because if raiseSalary() is present for a specific employee type, only that function would be called.

|  |
| --- |
| class Employee  {  public:      virtual void raiseSalary()      {  /\* common raise salary code \*/  }        virtual void promote()      { /\* common promote code \*/ }  };    class Manager: public Employee {      virtual void raiseSalary()      {  /\* Manager specific raise salary code, may contain            increment of manager specific incentives\*/  }        virtual void promote()      { /\* Manager specific promote \*/ }  };    // Similarly, there may be other types of employees    // We need a very simple function to increment salary of all employees  // Note that emp[] is an array of pointers and actual pointed objects can  // be any type of employees. This function should ideally be in a class  // like Organization, we have made it global to keep things simple  void globalRaiseSalary(Employee \*emp[], int n)  {      for (int i = 0; i < n; i++)          emp[i]->raiseSalary(); // Polymorphic Call: Calls raiseSalary()                                 // according to the actual object, not                                 // according to the type of pointer  } |

Run on IDE

like globalRaiseSalary(), there can be many other operations that can be appropriately done on a list of employees without even knowing the type of actual object.  
Virtual functions are so useful that later languages like [Java keep all methods as virtual by default](https://www.geeksforgeeks.org/g-fact-43/).

**How does compiler do this magic of late resolution?**  
Compiler maintains two things to this magic :  
virtualFuns  
[**vtable:**](http://en.wikipedia.org/wiki/Virtual_method_table) A table of function pointers. It is maintained per class.  
[**vptr:**](http://en.wikipedia.org/wiki/Virtual_method_table#Implementation)A pointer to vtable. It is maintained per object (See [this](http://geeksquiz.com/c-virtual-functions-question-12/) for an example).

Compiler adds additional code at two places to maintain and use vptr.  
**1)** Code in every constructor. This code sets vptr of the object being created. This code sets vptr to point to vtable of the class.  
**2)** Code with polymorphic function call (e.g. bp->show() in above code). Wherever a polymorphic call is made, compiler inserts code to first look for vptr using base class pointer or reference (In the above example, since pointed or referred object is of derived type, vptr of derived class is accessed). Once vptr is fetched, vtable of derived class can be accessed. Using vtable, address of derived derived class function show() is accessed and called.

**Is this a standard way for implementation of run-time polymorphism in C++?**  
The C++ standards do not mandate exactly how runtime polymophism must be implemented, but compilers generally use minor variations on the same basic model.

[Quiz on Virtual Functions](http://quiz.geeksforgeeks.org/c-plus-plus/virtual-functions/).

Predict output of the following program

|  |
| --- |
| #include<iostream>  using namespace std;    class Base  {  public:      virtual void show() { cout<<" In Base \n"; }  };    class Derived: public Base  {  public:      void show() { cout<<"In Derived \n"; }  };    int main(void)  {      Base \*bp = new Derived;      bp->show();        Base &br = \*bp;      br.show();        return 0;  } |

Answer: In Derived In derived- Since show() is virtual in base class, it is called according to the type of object being referred or pointed, rather than the type of pointer or reference.

2 Which of the following is true about pure virtual functions? 1) Their implementation is not provided in a class where they are declared. 2) If a class has a pure virtual function, then the class becomes abstract class and an instance of this class cannot be created.

Answer: Only 2 is true.

3 #include<iostream>

using namespace std;

class Base

{

public:

    virtual void show() = 0;

};

int main(void)

{

    Base b;

    Base \*bp;

    return 0;

}

Answer: Compile error in line  Base b; Since Base has a pure virtual function, it becomes an abstract class and an instance of it cannot be created. So there is an error in line "Base b". Note that there is no error in line "Base \*bp;". We can have pointers or references of abstract classes.

4 Predict the output of following program.

|  |
| --- |
| #include<iostream>  using namespace std;  class Base  {  public:      virtual void show() = 0;  };    class Derived : public Base { };    int main(void)  {      Derived q;      return 0;  } |

Answer: Compile error. Derived is abstract class. If we don't override the pure virtual function in derived class, then derived class also becomes abstract class.

5 Can a constructor be virtual? Will the following program compile?

|  |
| --- |
| #include <iostream>  using namespace std;  class Base {  public:    virtual Base() {}  };  int main() {     return 0;  } |

Answer: No. There is nothing like Virtual Constructor. Making constructors virtual doesn't make sense as constructor is responsible for creating an object and it can’t be delegated to any other object by virtual keyword means.

6 Can a destructor be virtual? Will the following program compile?

|  |
| --- |
| #include <iostream>  using namespace std;  class Base {  public:    virtual ~Base() {}  };  int main() {     return 0;  } |

Answer: Yes. A destructor can be virtual. We may want to call appropriate destructor when a base class pointer points to a derived class object and we delete the object. If destructor is not virtual, then only the base class destructor may be called. For example, consider the following program.

// Not good code as destructor is not virtual

#include<iostream>

using namespace std;

class Base {

public:

Base() { cout << "Constructor: Base" << endl; }

~Base() { cout << "Destructor : Base" << endl; }

};

class Derived: public Base {

public:

Derived() { cout << "Constructor: Derived" << endl; }

~Derived() { cout << "Destructor : Derived" << endl; }

};

int main() {

Base \*Var = new Derived();

delete Var;

return 0;

}

Output on GCC:

Constructor: Base

Constructor: Derived

Destructor : Base

7 Can static functions be virtual? Will the following program compile?

#include<iostream>

using namespace std;

class Test

{

   public:

      virtual static void fun()  { }

};

Answer: No. Static functions are class specific and may not be called on objects. Virtual functions are called according to the pointed or referred object.

8 Predict the output of following C++ program. Assume that there is no alignment and a typical implementation of virtual functions is done by the compiler.

#include <iostream>

using namespace std;

class A

{

public:

virtual void fun();

};

class B

{

public:

void fun();

};

int main()

{

int a = sizeof(A), b = sizeof(B);

if (a == b) cout << "a == b";

else if (a > b) cout << "a > b";

else cout << "a < b";

return 0;

}

Answer: a > b. Class A has a VPTR which is not there in class B. In a typical implementation of virtual functions, compiler places a VPTR with every object. Compiler secretly adds some code in every constructor to this.

9 #include <iostream>

using namespace std;

class A

{

public:

virtual void fun() { cout << "A::fun() "; }

};

class B: public A

{

public:

void fun() { cout << "B::fun() "; }

};

class C: public B

{

public:

void fun() { cout << "C::fun() "; }

};

int main()

{

B \*bp = new C;

bp->fun();

return 0;

}

Answer: C::fun().The important thing to note here is B::fun() is virtual even if we have not uses virtual keyword with it. When a class has a virtual function, functions with same signature in all descendant classes automatically become virtual. We don't need to use virtual keyword in declaration of fun() in B and C. They are anyways virtual.

10 Predict the output of following C++ program

#include<iostream>

using namespace std;

class Base

{

public:

virtual void show() { cout<<" In Base \n"; }

};

class Derived: public Base

{

public:

void show() { cout<<"In Derived \n"; }

};

int main(void)

{

Base \*bp = new Derived;

bp->Base::show(); // Note the use of scope resolution here

return 0;

}

Answer: In Base. A base class function can be accessed with scope resolution operator even if the function is virtual.

# **Reasons for a C++ program crash**

We sometimes come across abnormal crash of C++ programs. Below are some possible reasons which may cause C++ to crash abnormally.

* [**Segmentation Fault**](http://www.geeksforgeeks.org/core-dump-segmentation-fault-c-cpp/)**:** It is the major reason for program to crash. These are may be the reasons for the such cause:
  + Attempting to access memory location that doesn’t exist in our system.
  + There may be an attempt to write on a **read only memory** location.

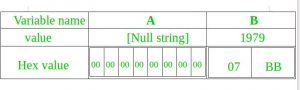
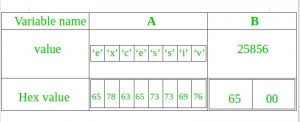
|  |
| --- |
| // CPP program to demonstrate  int main()  {     char \*str;       /\* Stored in read only part of data segment \*/     str = "GfG";       /\* Problem:  trying to modify read only memory \*/     \*(str+1) = 'n';     return 0;  } |

* + Run on IDE
  + Output :
  + Segmentation fault (core dumped)
  + There may be an attempt to access **protected memory location** such as **kernel memory**
  + [**Stack Overflow**](https://www.geeksforgeeks.org/heap-overflow-stack-overflow/)**:** There may case of **non terminating recursion** with memory location.

|  |
| --- |
| // C program to demonstrate stack overflow  // by creating a non-terminating recursive  // function.  #include<stdio.h>    void fun(int x)  {      if (x == 1)         return;      x = 6;      fun(x);  }    int main()  {     int x = 5;     fun(x);  } |

* + Run on IDE
  + Output :
  + Segmentation fault (core dumped)
* [**Buffer Overflow:**](https://www.geeksforgeeks.org/buffer-overflow-attack-with-example/) It is an anomaly where a program, while writing data to a buffer, overruns the buffer’s boundary and overwrites adjacent memory locations.
  + Consider below C++ program.

|  |
| --- |
| // C++ code to demonstrate buffer  // overflow.  #include <bits/stdc++.h>  using namespace std;    // driver code  int main()  {      char A[8] = "";      unsigned short B = 1979;      strcpy(A, "excessive");      return 0;  } |

* + Run on IDE
  + Output :
  + \*\*\* stack smashing detected \*\*\*: /home/gfg/a terminated
  + Aborted (core dumped)
  + The program has two variables which are adjacent in memory: an 8-byte-long string buffer, A, and a two-byte [big-endian](https://www.geeksforgeeks.org/little-and-big-endian-mystery/) integer, B.
  + *char A[8] = “”;  
    unsigned short B = 1979;*
  + Initially, A contains nothing but zero bytes, and B contains the number 1979.  
      
    Now, the program attempts to store the null-terminated string “excessive” with ASCII encoding in the A buffer.
  + *strcpy(A, “excessive”);*
  + This string is of 9 characters long and encodes to 10 bytes including the null terminator, but A can take only 8 bytes. By failing to check the length of the string, it also overwrites the value of B:
  + 
  + B’s value has now been inadvertently replaced by a number formed from part of the character string. In this example “e” followed by a zero byte would become 25856.
  + This overflow is called **buffer buffer overflow**.
* [**Memory Leak:**](https://www.geeksforgeeks.org/what-is-memory-leak-how-can-we-avoid/)  
  If we allocate some memory by some program and let it be as it is. After sometime there will be huge memory that is allocated but not used so this would lack of memory after sometime. And thus program start crashing.

|  |
| --- |
| // C program to demonstrate heap overflow  // by continuously allocating memory  #include<stdio.h>    int main()  {      for (int i=0; i<10000000; i++)      {         // Allocating memory without freeing it         int \*ptr = (int \*)malloc(sizeof(int));      }  } |

* Run on IDE
* **Exceptions**
  + Divide by zero.

|  |
| --- |
| // C++ code to demonstrate divide by 0.  #include <bits/stdc++.h>  using namespace std;    // driver code  int main()  {      int x = 10;      int y = 0;      cout << x/y;      return 0;  } |

* + Run on IDE
  + Output :
  + Floating point exception (core dumped)

# **malloc() vs new**

**Following are the differences between malloc() and operator new.**:

1. **Calling Constructors:**new calls constructors, while malloc() does not. In fact primitive data types (char, int, float.. etc) can also be initialized with new. For example, below program prints 10.

|  |
| --- |
| #include<iostream>    using namespace std;    int main()  {     int \*n = new int(10); // initialization with new()     cout << \*n;     getchar();     return 0;  } |

1. **operator vs function:** new is an operator, while malloc() is a function.
2. **return type:** new returns exact data type, while malloc() returns void \*.
3. **Failure Condition:**On failure, malloc() returns NULL where as new Throws.
4. **Memory:** In case of new, memory is allocated from free store where as in malloc() memory allocation is done from heap.
5. **Overriding:** We are allowed to override new operator where as we can not override the malloc() function legally.
6. **Size:** Required size of memory is calculated by compiler for new, where as we have to manually calculate size for malloc().

|  |  |
| --- | --- |
| **NEW** | **MALLOC** |
| calls constructor | doesnot calls constructors |
| It is an operator | It is a function |
| Returns exact data type | Returns void \* |
| on failure, Throws | On failure, returns NULL |
| Memory allocated from free store | Memory allocated from heap |
| can be overridden | cannot be overridden |
| size is calculated by compiler | size is calculated manually |

Predict the output of following program?

#include <iostream>

using namespace std;

class Test

{

private:

int x;

public:

Test(int i)

{

x = i;

cout << "Called" << endl;

}

};

int main()

{

Test t(20);

t = 30; // conversion constructor is called here.

return 0;

}

|  |  |
| --- | --- |
| A | Compiler Error |
| B | Called  Called |
| C | Called |

Answer: B

**Question 16 Explanation:**

If a class has a constructor which can be called with a single argument, then this constructor becomes conversion constructor because such a constructor allows automatic conversion to the class being constructed. A conversion constructor can be called anywhere when the type of single argument is assigned to the object. The output of the given program is

Called

Called

# **Why copy constructor argument should be const in C++?**

When we create our own copy constructor, we pass an object by reference and we generally pass it as a const reference.   
One reason for passing const reference is, we should use const in C++ wherever possible so that objects are not accidentally modified. This is one good reason for passing reference as const, but there is more to it. For example, predict the output of following C++ program. Assume that [copy elision](https://www.geeksforgeeks.org/copy-elision-in-c/) is not done by compiler.

|  |
| --- |
| #include<iostream>  using namespace std;    class Test  {     /\* Class data members \*/  public:     Test(Test &t) { /\* Copy data members from t\*/}     Test()        { /\* Initialize data members \*/ }  };    Test fun()  {      cout << "fun() Called\n";      Test t;      return t;  }    int main()  {      Test t1;      Test t2 = fun();      return 0;  } |

Run on IDE

Output:

Compiler Error in line "Test t2 = fun();"

The program looks fine at first look, but it has compiler error. If we add const in copy constructor, the program works fine, i.e., we change copy constructor to following.

|  |
| --- |
| Test(const Test &t) { cout << "Copy Constructor Called\n"; } |

Run on IDE

Or if we change the line “Test t2 = fun();” to following two lines, then also the program works fine.

|  |
| --- |
| Test t2;  t2 = fun(); |

Run on IDE

The function fun() returns by value. So the compiler creates a temporary object which is copied to t2 using copy constructor in the original program (The temporary object is passed as an argument to copy constructor). The reason for compiler error is, compiler created temporary objects cannot be bound to non-const references and the original program tries to do that. It doesn’t make sense to modify compiler created temporary objects as they can die any moment.

A copy constructor is called when an object is passed by value. Copy constructor itself is a function. So if we pass argument by value in a copy constructor, a call to copy constructor would be made to call copy constructor which becomes a non-terminating chain of calls. Therefore compiler doesn’t allow parameters to be pass by value. See <http://geeksquiz.com/copy-constructor-in-cpp/> for details.

# **How ro access global variable when both global and local variable have same name?**

Using scope operator as shown in example below.

{

public:

    int i;

    void get();

};

void Test::get()

{

    std::cout << "Enter the value of i: ";

    std::cin >> i;

}

Test t;  // Global object

int main()

{

    Test t;  // local object

    t.get();

    std::cout << "value of i in local t: "<<t.i<<'\n';

    ::t.get();

    std::cout << "value of i in global t: "<<::t.i<<'\n';

    return 0;

}

# **Sizeof for derived and base class**

#include <iostream>

using namespace std;

class base {

int arr[10];

};

class b1: public base { };

class b2: public base { };

class derived: public b1, public b2 {};

int main(void)

{

cout << sizeof(base) << endl;

cout << sizeof(b1) << endl;

cout << sizeof(b2) << endl;

cout << sizeof(derived);

return 0;

}

Output

40

40

40

80

Here, Since b1 and b2 both inherit from class base, two copies of class base are there in class derived. This kind of inheritance without virtual causes wastage of space and ambiguities. virtual base classes are used to save space and avoid ambiguities in such cases. For example, following program prints 48. 8 extra bytes are for bookkeeping information stored by the compiler (See this for details)

#include <iostream>

using namespace std;

class base {

int arr[10];

};

class b1: virtual public base { };

class b2: virtual public base { };

class derived: public b1, public b2 {};

int main(void)

{

cout << sizeof(base) << endl;

cout << sizeof(b1) << endl;

cout << sizeof(b2) << endl;

cout << sizeof(derived);

return 0;

}

40

48

48

56

# Const keyword

1. Predict the output of following program

|  |
| --- |
| #include <iostream>  using namespace std;  int main()  {      const char\* p = "12345";      const char \*\*q = &p;      \*q = "abcde";      const char \*s = ++p;      p = "XYZWVU";      cout << \*++s;      return 0;  } |

Output: c

Output is ‘c’ const char\* p = “12345″ declares a pointer to a constant. So we can’t assign something else to \*p, but we can assign new value to p. const char \*\*q = &p; declares a pointer to a pointer. We can’t assign something else to \*\*q, but we can assign new values to q and \*q. \*q = “abcde”; changes p to point to “abcde” const char \*s = ++p; assigns address of literal ”bcde” to s. Again \*s can’t be assigned a new value, but s can be changed. The statement printf(“%cn”, \*++s) changes s to “cde” and first character at s is printed.

1. Which of these statements are true?

In C++, const qualifier can be applied to 1) Member functions of a class 2) Function arguments 3) To a class data member which is declared as static 4) Reference variables

Ans: All.

When a function is declared as const, it cannot modify data members of its class. When we don't want to modify an argument and pass it as reference or pointer, we use const qualifier so that the argument is not accidentally modified in function. Class data members can be declared as both const and static for class wide constants. Reference variables can be const when they refer a const location.

3 Predict the output of following program.

|  |
| --- |
| #include <iostream>  using namespace std;  class Point  {      int x, y;  public:   Point(int i = 0, int j =0)     { x = i; y = j;  }     int getX() const { return x; }     int getY() {return y;}  };    int main()  {      const Point t;      cout << t.getX() << " ";      cout << t.gety();      return 0;  } |

Output: There is compiler Error in line cout << t.gety(); A const object can only call const functions.

4 #include <stdio.h>

int main()

{

   const int x;

   x = 10;

   printf("%d", x);

   return 0;

}

Output: Compiler error. One cannot change the value of 'const' variable except at the time of initialization. Compiler does check this.

# **Private Destructor**

Predict the output of following programs.

|  |
| --- |
| #include <iostream>  using namespace std;    class Test  {  private:     ~Test() {}  };  int main()  { } |

Run on IDE

The above program compiles and runs fine. It is **not**compiler error to create private destructors. What do you say about below program.

|  |
| --- |
| #include <iostream>  using namespace std;    class Test  {  private:     ~Test() {}  };  int main()  {    Test t;  } |

Run on IDE

The above program fails in compilation. The compiler notices that the local variable ‘t’ cannot be destructed because the destructor is private. What about the below program?

|  |
| --- |
| #include <iostream>  using namespace std;    class Test  {  private:     ~Test() {}  };  int main()  {     Test \*t;  } |

Run on IDE

The above program works fine. There is no object being constructed, the program just creates a pointer of type “Test \*”, so nothing is destructed. What about the below program?

|  |
| --- |
| #include <iostream>  using namespace std;    class Test  {  private:     ~Test() {}  };  int main()  {     Test \*t = new Test;  } |

Run on IDE

The above program also works fine. When something is created using dynamic memory allocation, it is programmer’s responsibility to delete it. So compiler doesn’t bother.

The below program fails in compilation. When we call delete, desturctor is called.

|  |
| --- |
| #include <iostream>  using namespace std;    class Test  {  private:     ~Test() {}  };  int main()  {     Test \*t = new Test;     delete t;  } |

Run on IDE

We noticed in the above programs, when a class has private destructur, only dynamic objects of that class can be created. Following is a way to create classes with private destructors and have a function as friend of the class. The function can only delete the objects.

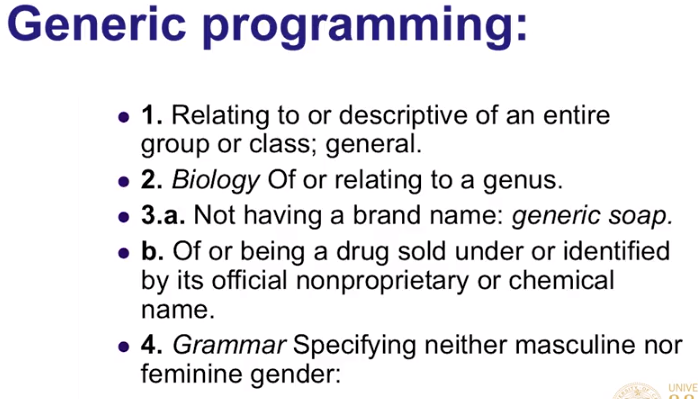
|  |
| --- |
| #include <iostream>    // A class with private destuctor  class Test  {  private:      ~Test() {}  friend void destructTest(Test\* );  };    // Only this function can destruct objects of Test  void destructTest(Test\* ptr)  {      delete ptr;  }    int main()  {      // create an object      Test \*ptr = new Test;        // destruct the object      destructTest (ptr);        return 0;  } |

Run on IDE

**What is the use of private destructor?**  
Whenever we want to control destruction of objects of a class, we make the destructor private. For dynamically created objects, it may happen that you pass a pointer to the object to a function and the function deletes the object. If the object is referred after the function call, the reference will become dangling. See [this](http://blogs.msdn.com/b/larryosterman/archive/2005/07/01/434684.aspx)for more details.

# Generics Templates

In general, generic programming means



In C++ generic templates, the variable used is called META variable. This is later substituted with the actual type such as int. eg:

template<class T>

inline void swap(T& d, T& s)

[

T temp = d,

d =s;

s = temp;

}

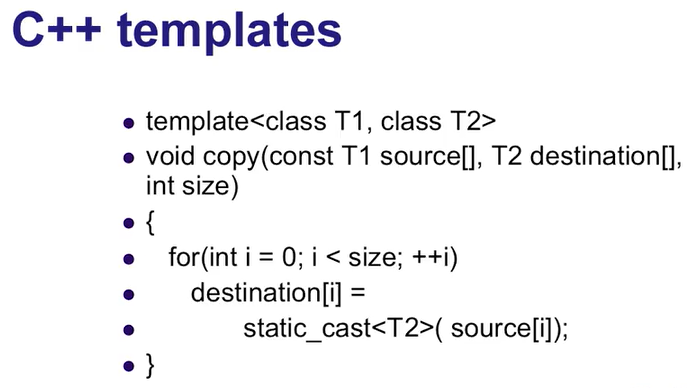
Here T is meta variable. Compiler replaces the ype for the meta variable as per the scenario. Eg:

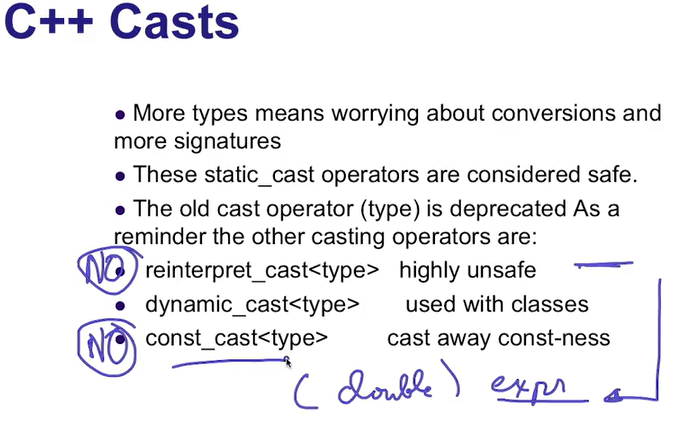
int m,n.....

...

swap(m.n); here compiler automatically compiles code with T replaced by int in swap

We can even have multiple argument templates. But we must use multiple argument templates very carefully since more geric means more complex code.





# This Pointer Which of the following is true about this pointer?

|  |  |
| --- | --- |
| A | It is passed as a hidden argument to all function calls |
| B | It is passed as a hidden argument to all non-static function calls |
| C | It is passed as a hidden argument to all static functions |
| D | None of the above |

**Answer: B**

|  |
| --- |
| **Question 2** |

What is the use of this pointer?

|  |  |
| --- | --- |
| A | When local variable’s name is same as member’s name, we can access member using this pointer. |
| B | To return reference to the calling object |
| C | Can be used for chained function calls on an object |
| D | All of the above |

**Answer: D**

|  |
| --- |
| **Question 3** |

Predict the output of following C++ program.

|  |
| --- |
| #include<iostream>  using namespace std;    class Test  {  private:    int x;  public:    Test(int x = 0) { this->x = x; }    void change(Test \*t) { this = t; }    void print() { cout << "x = " << x << endl; }  };    int main()  {    Test obj(5);    Test \*ptr = new Test (10);    obj.change(ptr);    obj.print();    return 0;  } |

Run on IDE

|  |  |
| --- | --- |
| A | x = 5 |
| B | x = 10 |
| C | Compiler Error |
| D | Runtime Error |

**Answer: C. ‘This’ is a constant pointer**

|  |
| --- |
| **Question 4** |

Predict the output of following C++ program

|  |
| --- |
| #include<iostream>  using namespace std;    class Test  {  private:    int x;    int y;  public:    Test(int x = 0, int y = 0) { this->x = x; this->y = y; }    static void fun1() { cout << "Inside fun1()"; }    static void fun2() { cout << "Inside fun2()"; this->fun1(); }  };    int main()  {    Test obj;    obj.fun2();    return 0;  } |

Run on IDE

|  |  |
| --- | --- |
| A | Inside fun2() Inside fun1() |
| B | Inside fun2() |
| C | Inside fun1() Inside fun2() |
| D | Compiler Error |

**Answer: D. There is error in fun2(). It is a static function and tries to access this pointer. this pointer is not available to static member functions as static member function can be called without any object.**

|  |
| --- |
| **Question 5** |

Predict the output of following C++ program?

|  |
| --- |
| #include<iostream>  using namespace std;    class Test  {  private:    int x;  public:    Test() {x = 0;}    void destroy()  { delete this; }    void print() { cout << "x = " << x; }  };    int main()  {    Test obj;    obj.destroy();    obj.print();    return 0;  } |

Run on IDE

|  |  |
| --- | --- |
| A | x = 0 |
| B | undefined behavior |
| C | compiler error |

**Answer: B. delete operator works only for objects allocated using operator new (See http://geeksforgeeks.org/?p=8539). If the object is created using new, then we can do delete this, otherwise behavior is undefined. See “delete this” in C++ for more examples.**

# Exception Handling

1 #include <iostream>

using namespace std;

int main()

{

   int x = -1;

   try {

      cout << "Inside try \n";

      if (x < 0)

      {

         throw x;

         cout << "After throw \n";

      }

   }

   catch (int x ) {

      cout << "Exception Caught \n";

   }

   cout << "After catch \n";

   return 0;

}

Output: Inside try

Exception caught

After catch

When an exception is thrown, lines of try block after the throw statement are not executed. When exception is caught, the code after catch block is executed. Catch blocks are generally written at the end through.

2 Which of thr below statements are true?

What is the advantage of exception handling?

1) Remove error-handling code from the software's main line of code.

2) A method writer can chose to handle certain exceptions and delegate

others to the caller.

3) An exception that occurs in a function can be handled anywhere in

the function call stack.

Ans: all 1,2 and 3

3 What should be put in a *try*block?

**1.** Statements that might cause exceptions

**2.** Statements that should be skipped in case of an exception

Ans: both 1 and 2

4 Output of following program

|  |
| --- |
| #include<iostream>  using namespace std;    class Base {};  class Derived: public Base {};  int main()  {     Derived d;     try {         throw d;     }     catch(Base b) {          cout<<"Caught Base Exception";     }     catch(Derived d) {          cout<<"Caught Derived Exception";     }     return 0;  } |

Output: Caught Base Exception

If both base and derived classes are caught as exceptions then catch block of derived class must appear before the base class. If we put base class first then the derived class catch block will never be reached. In Java, catching a base class exception before derived is not allowed by the compiler itself. In C++, compiler might give warning about it, but compiles the code.

5 #include <iostream>

using namespace std;

int main()

{

    try

    {

       throw 'a';

    }

    catch (int param)

    {

        cout << "int exception\n";

    }

    catch (...)

    {

        cout << "default exception\n";

    }

    cout << "After Exception";

    return 0;

}

Output: default exception

After exception

The block catch(...) is used for catch all, when a data type of a thrown exception doesn't match with any other catch block, the code inside catch(...) is executed. Note that the implicit type conversion doesn't happen when exceptions are caught. The character 'a' is not automatically converted to int

6 #include <iostream>

using namespace std;

int main()

{

    try

    {

       throw 10;

    }

    catch (...)

    {

        cout << "default exception\n";

    }

    catch (int param)

    {

        cout << "int exception\n";

    }

    return 0;

}

Output: Compiler error. It is compiler error to put catch all block before any other catch. The catch(...) must be the last catch block.

7 #include <iostream>

using namespace std;

int main()

{

    try

    {

        try

        {

            throw 20;

        }

        catch (int n)

        {

            cout << "Inner Catch\n";

            throw;

        }

    }

    catch (int x)

    {

        cout << "Outer Catch\n";

    }

    return 0;

}

Output: Inner catch

Outer catch.

The statement 'throw;' is used to re-throw an exception. This is useful when a function can handles some part of the exception handling and then delegates the remaining part to the caller. A catch block cleans up resources of its function, and then rethrows the exception for handling elsewhere.

8 #include <iostream>

using namespace std;

class Test {

public:

   Test() { cout << "Constructing an object of Test " << endl; }

  ~Test() { cout << "Destructing an object of Test "  << endl; }

};

int main() {

  try {

    Test t1;

    throw 10;

  } catch(int i) {

    cout << "Caught " << i << endl;

  }

}

Output: Constructing an object of Test

Destructing an object of Test

Caught 10

When an object is created inside a try block, destructor for the object is called before control is transferred to catch block.

9 #include <iostream>

using namespace std;

class Test {

  static int count;

  int id;

public:

  Test() {

    count++;

    id = count;

    cout << "Constructing object number " << id << endl;

    if(id == 4)

       throw 4;

  }

  ~Test() { cout << "Destructing object number " << id << endl; }

};

int Test::count = 0;

int main() {

  try {

    Test array[5];

  } catch(int i) {

    cout << "Caught " << i << endl;

  }

}

Output: Constructing object number 1

Constructing object number 2

Constructing object number 3

Constructing object number 4

Destructing object number 3

Destructing object number 2

Destructing object number 1

Caught 4

The destructors are called in reverse order of constructors. Also, after the try block, the destructors are called only for completely constructed objects.

10 Which of the following is true about exception handling in C++? 1) There is a standard exception class like Exception class in Java. 2) All exceptions are unchecked in C++, i.e., compiler doesn't check if the exceptions are caught or not. 3) In C++, a function can specify the list of exceptions that it can throw using comma separated list like following.

void fun(int a, char b) throw (Exception1, Exception2, ..)

Answer: all 1,2 and 3

11 What happens in C++ when an exception is thrown and not caught anywhere like following program.

#include <iostream>

using namespace std;

int fun() throw (int)

{

throw 10;

}

int main() {

fun();

return 0;

}

Answer: abnormal program termination. When an exception is thrown and not caught, the program terminates abnormally.

12 What happens when a function throws an error but doesn't specify it in the list of exceptions it can throw. For example, what is the output of following program?

#include <iostream>

using namespace std;

// Ideally it should have been "int fun() (int)"

int fun()

{

throw 10;

}

int main()

{

try

{

fun();

}

catch (int )

{

cout << "Caught";

}

return 0;

}

Answer: No compiler Error. Output is "Caught"

C++ compiler doesn't check enforce a function to list the exceptions that it can throw. In Java, it is enforced. It is up to the programmer to specify. Being a civilized programmer, a programmer should specify the list.

# C++ Templates

1. Which of the following is true about templates.  
1) Template is a feature of C++ that allows us to write one code for different data types.  
  
2) We can write one function that can be used for all data types including user defined types. Like sort(), max(), min(), ..etc.  
  
3) We can write one class or struct that can be used for all data types including user defined types. Like Linked List, Stack, Queue ..etc.  
  
4) Template is an example of compile time polymorphism.

Answer;, all 1,2,3 and 4

2 Predict the output?

|  |
| --- |
| #include <iostream>  using namespace std;    template <typename T>  void fun(const T&x)  {      static int count = 0;      cout << "x = " << x << " count = " << count << endl;      ++count;      return;  }    int main()  {      fun<int> (1);      cout << endl;      fun<int>(1);      cout << endl;      fun<double>(1.1);      cout << endl;      return 0;  } |

Output: x = 1 count = 0

x = 1 count = 1

x = 1.1 count = 0

Compiler creates a new instance of a template function for every data type. So compiler creates two functions in the above example, one for int and other for double. Every instance has its own copy of static variable. The int instance of function is called twice, so count is incremented for the second call.

3 include <iostream>

using namespace std;

template <typename T>

T max(T x, T y)

{

    return (x > y)? x : y;

}

int main()

{

    cout << max(3, 7) << std::endl;

    cout << max(3.0, 7.0) << std::endl;

    cout << max(3, 7.0) << std::endl;

    return 0;

}

Output: Compiler Error in last cout statement as call to max is ambiguous.

4 Output of following program?

|  |
| --- |
| #include <iostream>  using namespace std;    template <class T>  class Test  {  private:      T val;  public:      static int count;      Test()  {   count++;   }  };    template<class T>  int Test<T>::count = 0;    int main()  {      Test<int> a;      Test<int> b;      Test<double> c;      cout << Test<int>::count   << endl;      cout << Test<double>::count << endl;      return 0;  } |

2

1

There are two classes created by the template: Test and Test. Since count is a static member, every class has its own copy of it. Also, count gets incremented in constructor

5 Output of following program.

|  |
| --- |
| #include <iostream>  using namespace std;    template <class T, int max>  int arrMin(T arr[], int n)  {     int m = max;     for (int i = 0; i < n; i++)        if (arr[i] < m)           m = arr[i];       return m;  }    int main()  {     int arr1[]  = {10, 20, 15, 12};     int n1 = sizeof(arr1)/sizeof(arr1[0]);       char arr2[] = {1, 2, 3};     int n2 = sizeof(arr2)/sizeof(arr2[0]);       cout << arrMin<int, 10000>(arr1, n1) << endl;     cout << arrMin<char, 256>(arr2, n2);     return 0;  } |

Output: 10

1

We can pass non-type arguments to templates. Non-type parameters are mainly used for specifying max or min values or any other constant value for a particular instance of template. The important thing to note about non-type parameters is, they must be const. Compiler must know the value of non-type parameters at compile time. Because compiler needs to create functions/classes for a specified non-type value at compile time. Following is another example of non-type parameters.

#include <iostream>

using namespace std;

template < class T, int N >

T fun (T arr[], int size)

{

if (size > N)

cout << "Not possible";

T max = arr[0];

for (int i = 1; i < size; i++)

if (max < arr[i])

max = arr[i];

return max;

}

int main ()

{

int arr[] = {12, 3, 14};

cout << fun (arr, 3);

}

6 Output?

|  |
| --- |
| #include <iostream>  using namespace std;    template <int i>  void fun()  {     i = 20;     cout << i;  }    int main()  {     fun<10>();     return 0;  } |

Output: Compiler error in line "i = 20;" Non-type parameters must be const, so they cannot be modified.

7 Output?

|  |
| --- |
| #include <iostream>  using namespace std;    template <class T>  T max (T &a, T &b)  {      return (a > b)? a : b;  }    template <>  int max <int> (int &a, int &b)  {      cout << "Called ";      return (a > b)? a : b;  }    int main ()  {      int a = 10, b = 20;      cout << max <int> (a, b);  } |

Output: Called 20. Above program is an example of template specialization. Sometime we want a different behaviour of a function/class template for a particular data type. For this, we can create a specialized version for that particular data type.

8 Output?

|  |
| --- |
| #include <iostream>  using namespace std;    template<int n> struct funStruct  {      static const int val = 2\*funStruct<n-1>::val;  };    template<> struct funStruct<0>  {      static const int val = 1 ;  };    int main()  {      cout << funStruct<10>::val << endl;      return 0;  } |

Output: 1024. This is an example of template metaprogramming. The program mainly calculates 2^10.