Chapter 1-

ACCESS SPECIFIERS

Contents

3.1. Introduction

In object-oriented programming, one of the major features is the idea of encapsulation or data hiding. This means you can keep certain things away from other functions or routines.

Data hiding is one of the important features of Object Oriented Programming which allows preventing the functions of a program to access directly the internal representation of a class type. Access specifiers defines the access rights for the statements or functions that follows it until another access specifier or till the end of a class. It defines the access control rules.

There are 3 types of access modifiers available in C++:

1. Public
2. Private
3. Protected

Access modifiers in the program, are followed by a colon. You can use either one, two or all 3 modifiers in the same class to set different boundaries for different class members.

Private:

The members declared as "private" can be accessed only within the same class and not from outside the class.

Public:

The members declared as "public" are accessible within the class as well as from outside the class.

Protected:

The members declared as "protected" cannot be accessed from outside the class, but can be accessed from a derived class. This is used when inheritaance is applied to the members of a class.

The members declared as "public" are accessible within the class as well as from outside the class.

3.2.1. Public Access specifiers

Public, means 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. Hence there are chances that they might change them. So the key members must not be declared public. Usually the data members are not declared in this part. The member functions are usually declared in this part. The public accesses specifier gives access to the member function from outside the class. This part allows the programmer of a class to easily change the data member of the class. In other words, it gives flexibility of reading and writing the data members of the class from anywhere in the class. We can also declare data member in the public part but not in 95% cases. The keyword public followed by the colon (:) means to indicate the data member and member function that are visible outside the class. If public access specifier is used while deriving class then the public data members of the

base class becomes the public member of the derived class and protected members becomes the protected in the derived class but the private members of the base class are inaccessible.

Syntax:

```
class PublicAccess
{          // public access modifier
   public:
   int x;          // Data Member Declaration
   void display();   // Member Function decaration
}
```

A public class is accessible everywhere.

```
class S {
 public: // n, f, E, A, B, C, U are public members
    int n;
    static void f() {}
    enum E {A, B, C};
    struct U {};
};
int main()
{
    S::f(); // S::f is accessible in main
    S s;
    s.n = S::B; // S::n and S::B are accesisble in main
    S::U x; // S::U is accessible in main
}
```

Example 1: Program demonstrating public access specifier

```
// public access specifier.cpp
#include <iostream>
using namespace std;
class base
{
        private:
      int x;
        protected:
           int y;
        public:
           int z;
        base() //constructor to initialize data members
        {
          x = 1;
          y = 2;
          z = 3;
        }
};
class derive: public base
{
        //y becomes protected and z becomes public members of class derive
        public:
           void showdata()
           {
              cout << "x is not accessible" << endl;
          cout << "value of y is " << y << endl;
          cout << "value of z is " << z << endl;
           }
};
```

```
int main()
{
    derive a; //object of derived class
    a.showdata();
    //a.x = 1;   not valid : private member can't be accessed outside of class
    //a.y = 2;   not valid : y is now private member of derived class
    //a.z = 3;   not valid : z is also now a private member of derived class
    return 0;
}       //end of program
```

Output:

```
x is not accessible
value of y is 2
value of z is 3
```

Example 2:

```
// 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;
}
```

Output:

```
Radius is:5.5
Area is:94.985
```

In the above program the data member radius is public so we are allowed to access it outside the class.

You can set and get the value of public variables without any member function as shown in the following example –

Example 3:

```
#include <iostream>
 using namespace std;
 class Line {
   public:
```

```
   double length;
   void setLength( double len );
   double getLength( void );
};
 // Member functions definitions
double Line::getLength(void) {
  return length ;
}

void Line::setLength( double len) {
  length = len;
}

// Main function for the program
int main() {
  Line line;

  // set line length
  line.setLength(6.0);
  cout << "Length of line : " << line.getLength() <<endl;
   // set line length without member function
  line.length = 10.0; // OK: because length is public
  cout << "Length of line : " << line.length <<endl;

  return 0;
}
```

Output:

```
Length of line : 6
Length of line : 10
```

3.2.2. Private Access Specifier

The visibility of the Private access specifier is limited to only for a class where it is defined. In other words, it can only be accessed by the definition of the class. A private member variable or function cannot be accessed, or even viewed from outside the class. Only the class and friend functions can access private members. By default all the members of a class would be private. If someone tries to access the private members of a class, they will get a compile time error.

Syntax:

```
class PrivateAccess
{
   // private access modifier
   private:
   int x;          // Data Member Declaration
   void display();   // Member Function decaration
}
```

For example in the following class width is a private member, which means until you label a member, it will be assumed a private member −

```
class Box {
   double width;
    public:
      double length;
      void setWidth( double wid );
      double getWidth( void );
};
```

Example 1: Program demonstrating private access specifier

```
#include <iostream>
```

```
using namespace std;

class base
{
        private:
      int x;

        protected:
           int y;

        public:
           int z;

        base() //constructor to initialize data members
        {
          x = 1;
          y = 2;
          z = 3;
        }
};

class derive: private base
{
        //y and z becomes private members of class derive and x remains private
        public:
           void showdata()
           {
              cout << "x is not accessible" << endl;
          cout << "value of y is " << y << endl;
          cout << "value of z is " << z << endl;
```

```
        }
};
int main()
{
    derive a; //object of derived class
    a.showdata();
    //a.x = 1;   not valid : private member can't be accessed outside of class
    //a.y = 2;   not valid : y is now private member of derived class
    //a.z = 3;   not valid : z is also now a private member of derived class
    return 0;
}       //end of program
```

Output:

```
x is not accessible
value of y is 2
value of z is 3
```

Example 2:

```
#include <iostream>
 using namespace std;
 class Box {
   public:
      double length;
      void setWidth( double wid );
      double getWidth( void );

   private:
```

```
    double width;
};

// Member functions definitions
double Box::getWidth(void) {
  return width ;
}

void Box::setWidth( double wid ) {
  width = wid;
}

// Main function for the program
int main() {
  Box box;

  // set box length without member function
  box.length = 10.0; // OK: because length is public
  cout << "Length of box : " << box.length <<endl;

  // set box width without member function
  // box.width = 10.0; // Error: because width is private
  box.setWidth(10.0);  // Use member function to set it.
  cout << "Width of box : " << box.getWidth() <<endl;
  return 0;
}
```

Output:

```
Length of box : 10
Width of box : 10
```

Example 3:

```
// 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()
		{ // member function can access private
			// data member radius
			return 3.14*radius*radius;
		}
	};

// main function
int main()
{
	// creating object of the class
	Circle obj;

	// trying to access private data member
```

```
    // directly outside the class
    obj.radius = 1.5;


    cout << "Area is:" << obj.compute_area();
    return 0;
}
```

Output:

The output of above program will be a compile time error because we are not allowed to access the private data members of a class directly outside the class.

```
In function 'int main()':
11:16: error: 'double Circle::radius' is private
    double radius;
           ^
31:9: error: within this context
  obj.radius = 1.5;
      ^
```

However, we can access the private data members of a class indirectly using the public member functions of the class. Below program explains how to do this:

Example 4:

```
// C++ program to demonstrate private access modifier

#include<iostream>
using namespace std;

class Circle
{
```

```
        // private data member
        private:
                double radius;

        // public member function
        public:
                void 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;
}
```

Output:

```
Radius is:1.5
Area is: 7.065
```

3.2.3. Protected Access Specifier

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. In this case, the members of the base class can be used only within the derived class as protected members except for the private members.

Syntax:

```
class ProtectedAccess
{
   // protected access modifier
   protected:
   int x;          // Data Member Declaration
   void display();   // Member Function decaration
}
```

Example 1:

```
// 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 the derived class can
        // access the protected data members of the base class

        obj1.setId(81);
        obj1.displayId();
        return 0;
}
```

Output:

```
id_protected is:81
```

Example 2:

In given example, width member will be accessible by any member function of its derived class SmallBox.

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

class Box {
  protected:
    double width;
};

class SmallBox:Box { // SmallBox is the derived class.
  public:
    void setSmallWidth( double wid );
    double getSmallWidth( void );
```

```
};

// Member functions of child class
double SmallBox::getSmallWidth(void) {
  return width ;
}

void SmallBox::setSmallWidth( double wid ) {
  width = wid;
}

// Main function for the program
int main() {
  SmallBox box;

  // set box width using member function
  box.setSmallWidth(5.0);
  cout << "Width of box : "<< box.getSmallWidth() << endl;

  return 0;
}
```

Output:

```
Width of box : 5
```

Example 3: Program demonstrating protected access specifier

```
// protected access specifier.cpp
#include <iostream>
using namespace std;

class base
{
```

```
        private:
      int x;

        protected:
           int y;

        public:
           int z;

        base() //constructor to initialize data members
        {
          x = 1;
          y = 2;
          z = 3;
        }
};

class derive: protected base
{
        //y and z becomes protected members of class derive
        public:
           void showdata()
           {
              cout << "x is not accessible" << endl;
          cout << "value of y is " << y << endl;
          cout << "value of z is " << z << endl;
           }
};
int main()
{
```

```
    derive a; //object of derived class
    a.showdata();
    //a.x = 1;   not valid : private member can't be accessed outside of class
    //a.y = 2;   not valid : y is now private member of derived class
    //a.z = 3;   not valid : z is also now a private member of derived class
    return 0;
}       //end of program
```

Output:

```
x is not accessible
value of y is 2
value of z is 3
```

3.3. Data Member

Accessing a data member depends solely on the access control of that data member. If its public, then the data member can be easily accessed using the direct member access (.) operator with the object of that class. If, the data member is defined as private or protected, then we cannot access the data variables directly. Then we will have to create special public member functions to access, use or initialize the private and protected data members. These member functions are also called Accessors and Mutator methods or getter and setter functions.

Initialization of data members

In C++, class variables are initialized in the same order as they appear in the class declaration.

Example 1:

```
#include<iostream>

using namespace std;

class Test {
```

```
private:
        int y;
        int x;
public:
        Test() : x(10), y(x + 10) {}
        void print();
};
void Test::print()
{
cout<<"x = "<<x<<" y = "<<y;
}
int main()
{
        Test t;
        t.print();
        getchar();
        return 0;
}
```

Output:

The program prints correct value of x, but some garbage value for y, because y is initialized before x as it appears before in the class declaration.

```
x = 10 y = 10
```

So one of the following two versions can be used to avoid the problem in above code.

Solution 1:

```
// Change the order of declaration.
class Test {
private:
        int x;
        int y;
public:
        Test() : x(10), y(x + 10) {}
        void print();
};
```

Solution 2:

```
// Change the order of initialization.
class Test {
private:
        int y;
        int x;
public:
        Test() : x(y-10), y(20) {}
        void print();
};
```

3.3.1. Static Data Member

When we declare a normal variable (data member) in a class, different copies of those data members create with the associated objects. In some cases when we need a common data member that should be same for all objects, we cannot do this using normal data members. To fulfil such cases, we need static data members.

It is a variable which is declared with the static keyword, it is also known as class member, and thus only single copy of the variable creates for all objects. Any changes in the static data member through one member function will reflect in all other object's member functions.

Declaration of a static member:

```
static data_type member_name;
```

Defining the static data member:

It should be defined outside of the class following this syntax:

```
data_type class_name :: member_name =value;
```

If you are calling a static data member within a member function, member function should be declared as static (i.e. a static member function can access the static data members).

Example 1:

```
#include <iostream>
using namespace std;
class Demo {
        private:
                static int X;
        public:
                static void fun()
                {
                        cout <<"Value of X: " << X << endl;
                }
};
//defining
int Demo :: X =10;
int main()
{
        Demo X;
        X.fun();
        return 0;
}
```

Output:

```
Value of X: 10
```

- **Accessing static data member without static member function**

A static data member can also be accessed through the class name without using the static member function (as it is a class member), here we need an Scope Resolution Operator (SRO) :: to access the static data member without static member function.

Syntax:

```
class_name :: static_data_member;
```

Example 2:

```
#include <iostream>
using namespace std;
class Demo  {
        public:
                static int ABC;
};
//defining
int Demo :: ABC =10;
int main()    {
        cout<<"\nValue of ABC: "<<Demo::ABC;
        return 0;
}
```

Output

```
Value of ABC: 10
```

In above program ABC is a class member (static data member), it can directly access with help on scope resolution operator.

Summary

C++ offers possibility to control access to class members and functions by using access specifiers.

Access specifiers are used to protect data from misuse.

Types of access specifiers in C++

- public
- private
- protected

Public class members and functions can be used from outside of a class by any function or other classes.

Protected class members and functions can be used inside its class. Protected members and functions cannot be accessed from other classes directly.

Protected data members and functions can be used by the class derived from this class.

Private class members and functions can be used only inside of class and by friend functions and classes.

For classes, default access specifier is private. The default access specifier for unions and structs is public.

Review Questions

1. Define private, protected and public access control.
2. Explain the different access specifiers for the class member in C++.
3. What is output of following code?

```
#include <iostream>

using namespace std;

class access          {

      int a = 10;

      void disp()     {

            cout<< "a: "<< a;

      }        };

int main()   {

      access a;

      a.disp();

      return 0;        }
```

4. What is output of following code?

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

```
class access     {
private:
    int a_pri = 10;
protected:
    int b_pro = 20;
public:
    int c_public = 30;
};
int main() {
    access a;
    cout<< "private: " << a.a_pri;
    cout<< "protected: "<< a.b_pro;
    cout<< "public: " << a.c_public;
    return 0;      }
```

5. What is output of following code?

```
#include <iostream>
using namespace std;
class access     {
public:
        int a_public = 30;
};
class access_modifier: private access
{
};
class inheritance:public access_modifier
{
public:
void disp()
{
cout<< access::a_public;
}
};
int main()
{
inheritance a;
a.disp();
return 0;
}
```

6. Output?

```
#include <iostream>
using namespace std;
class rectangle {
int x, y;
public:
void val (int, int);
int area ()       {
        return (x * y);
}         };
```

```
void rectangle::val (int a, int b) {
x = a;
y = b;
}
int main () {
rectangle rect;
rect.val (3, 4);
cout << "rect area: " << rect.area();
return 0;
}
```

7. What is the role of protected access specifier?
8. Explain the static member function.
9. Which access specifier/s can help to achive data hiding in C++?
10. When a class member is defined outside the class, which operator can be used to associate the function definition to a particular class?

Chapter 4

INHERITENCE

Contents

10.1. Introduction

Reusability is an important feature of OOP. Reusability is re- usage of structure without changing the existing one and adding new features or characteristics to it. Since classes can be derived from existing class, size of the code is reduced. Hence adds code efficiency and time saving feature to the language. This is possible by creating a new class from the existing class. The newly created class will have features of both the existing as well as new class. In OOPs, the concept of inheritance provides the idea of reusability. The outcome of Inheritance explains the concept of reusability.

Inheritance is the mechanism by which one class can inherit the properties of another. It allows a hierarchy of classes to be build, moving from the most general to the most specific. When one class is inherited by another, the class that is inherited is called the **base class**. The inheriting class is called the **derived class.** In general, the process of inheritance begins with the definition of a base class. The base class defines all qualities that will be common to any derived class. The base class represent the most general behaviour of a class. The derived class inherits those general behaviour and adds properties that are specific to that class.

10.2. Defining Base Class and Derived Class

The class whose properties, data members or methods are inherited by other class is called Base Class. The base class is also called as super class, parent class or ancestor class.

The class that inherits properties, data members or methods from another class (i.e. base class) is called as Derived Class. The derived class is also called as sub class, child class. It allows to build new classes using previously defined ones and expand or modify the operation of a class. Derived class has access to public and protected members.

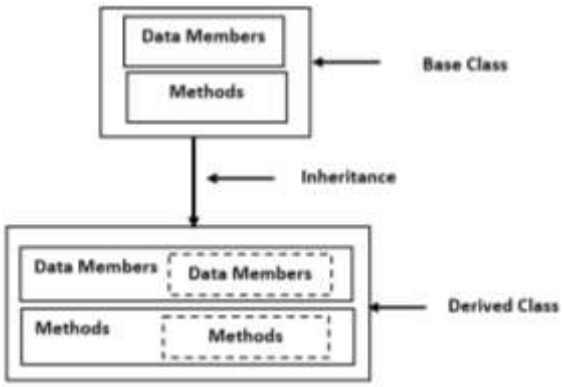


Figure 1- Inheritance

Syntax to define a derived class is:

```
class derivedclass_name : visibility-mode  baseclass_name
{
    // data members
   // methods
}
```

The colon indicates that the *derivedclass_name* is derived from *baseclass_name*. The *visibility-mode* is optional, if present may be either public or private. Visibility-mode specifies whether the features of base class are privately derived or publicly derived. The default visibility-mode is private.

Examples of Visibility mode:

privately derived (by default)

```
class ABC: XYZ
(
        members of ABC
};
```

publicly derived

```
class ABC: public XYZ
(
        members of ABC
};
```

privately derived

```
class ABC: private XYZ
(
        members of ABC
};
```

When a base class is **privately inherited** by a derived class, 'public members' of the base class become 'private members' of the derived class and therefore the public members of the base class can only be accessed by the member functions of the derived class. They are inaccessible to the objects of the derived class. A public member of a class can be accessed by its own objects using the dot operator.The result is that no member of the base class is accessible to the objects of the derived class.

On the other hand, when the base class is publicly inherited 'public members' of the base class become ‘public members' of the derived class and therefore they are accessible to the objects of the derived class. In both cases the private members are not inherited and therefore, the private members of a base class will never become the members of its derived class.

10.3 Access specifiers in Inheritance

When creating a derived class from a base class, different access specifiers can be used to inherit the data members of the base class. These can be **public, protected or private**.

The three access specifiers are explained below in detail:

i. **Public Access specifier:** If a sub class is derive 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.
ii. **Protected Access specifier:** If a sub class is derive from a protected base class, then both public member and protected members of the base class will become protected in derived class.
iii. **Private Access specifier:** If a sub class is derive from a Private base class, then both public member and protected members of the base class will become Private in derived class.

Access specifiers in Base Class	**Access specifiers in Derived Class**		
	Public	**Protected**	**Private**
Public	*Public*	*Protected*	*Private*
Protected	*Protected*	*Protected*	*Private*
Private	*Not Accessible*	*Not Accessible*	*Not Accessible*

Table 1 – Summary of three access specifiers

Example of public, protected and private access specifiers in C++

```
class base
{
	public:
		int x;
	protected:
		int y;
	private:
		int z;
};
class publicDerived: public base
{
	// x is public
	// y is protected
	// z is not accessible from publicDerived
};
class protectedDerived: protected base
{
	// x is protected
	// y is protected
	// z is not accessible from protectedDerived
};
class privateDerived: private base
{
	// x is private
	// y is private
	// z is not accessible from privateDerived
};
```

In the above example,

i. base has three member variables: **x, y** and **z** which are **public, protected** and **private** member respectively

ii. **publicDerived** inherits variables x and y as public and protected. **z** is not inherited as it is a **private** member variable of base.

iii. **protectedDerived** inherits variables x and y. Both variables become protected. z is not inherited. If we derive a class **derivedFromProtectedDerived** from **protectedDerived**, variables x and y are also inherited to the derived class.

iv. **privateDerived** inherits variables x and y. Both variables become private. z is not inherited. If we derive a class **derivedFromPrivateDerived** from **privateDerived**, variables x and y are not inherited because they are private variables of privateDerived.

10.4. Types of Inheritance

A class can be derived from more than one classes, which means it can inherit data members and methods from multiple base classes. Based on the ways of inheriting the features of base class into derived class inheritance can be classified into five main categories.

Types of Inheritance are:

- Single inheritance

- Multiple inheritance
- Multilevel inheritance
- Hierarchical inheritance
- Hybrid inheritance

10.4.1. Single Inheritance

The process in which a derived class inherits functions or methods from only one base class, is called as single inheritance. In single inheritance, there is only one base class and one derived class. The derived class can inherit the behaviour and attributes of the base class. However the vice versa is not possible. The derived class can add its own properties i.e. data members (variables) and functions. It can extend or use properties of the base class without any modification to the base class.

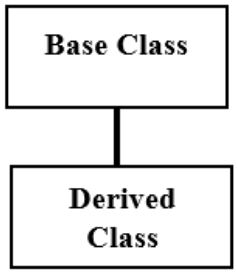


Figure 2- Single Inheritance

Example 1:

```
#include <iostream>
using namespace std;
class A {
public:
  A(){
    cout<<"Constructor of A class"<<endl;
  }
};
class B: public A {
public:
  B(){
    cout<<"Constructor of B class";
  }
};
int main() {
   //Creating object of class B
   B obj;
   return 0;
}
```

Given below is the output of above c++ code:

```
Enter First Number : 20

Enter Second Number : 50

Addition of Two Numbers : 70
```

Example 2:

```
// C++ program to explain Single inheritance
#include <iostream>
using namespace std;

// base class
class Vehicle {
public:
        Vehicle()
        {
        cout << "This is a Vehicle" << endl;
        }
};

// sub class derived from two base classes
class Car: public Vehicle{

};

// main function
int main()
{
        // creating object of sub class will
        // invoke the constructor of base classes
        Car obj;
        return 0;
}
```

Output:

This is a vehicle

10.4.2. Multiple Inheritance

The process in which a derived class inherits traits from several base classes, is called multiple inheritance. In Multiple inheritance, there is only one derived class and several base classes.

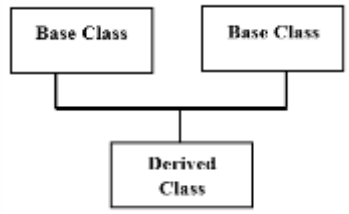


Figure 3- Multiple Inheritance

Syntax for declaring base and derived class in multiple inheritance:

```
class A
 {
   properties;
    methods;
 };
 class B
 {
   properties;
    methods;
  } ;
 class C : acess_specifier A,access_specifier A // derived class from A and B
 {
   properties;
    methods;
 } ;
```

Example of Multiple Inheritance:

```
#include <iostream>
using namespace std;
class A {
public:
  A(){
     cout<<"Constructor of A class"<<endl;
  }
};
class B {
public:
  B(){
     cout<<"Constructor of B class"<<endl;
  }
};
class C: public A, public B {
public:
  C(){
```

```
    cout<<"Constructor of C class"<<endl;
  }
};
int main() {
  //Creating object of class C
  C obj;
  return 0;
}
```

Output of above code is:

```
Constructor of A class
Constructor of B class
Constructor of C class
```

Example 2:

```
// C++ program to explain
// multiple inheritance
#include <iostream>
using namespace std;

// first base class
class Vehicle {
public:
        Vehicle()
        {
        cout << "This is a Vehicle" << endl;
        }
};

// second base class
class FourWheeler {
public:
        FourWheeler()
        {
        cout << "This is a 4 wheeler Vehicle" << endl;
        }
};

// sub class derived from two base classes
class Car: public Vehicle, public FourWheeler {

};

// main function
int main()
```

```
{
        // creating object of sub class will
        // invoke the constructor of base classes
        Car obj;
        return 0;
}
```

Output:

```
This is a Vehicle
This is a 4 wheeler Vehicle
```

10.4.3. Multilevel Inheritance

In this type of inheritance, a derived class is created from another derived class. The derived class in this type of inheritance can be derived not only from base class but can also be derived from another derived class.

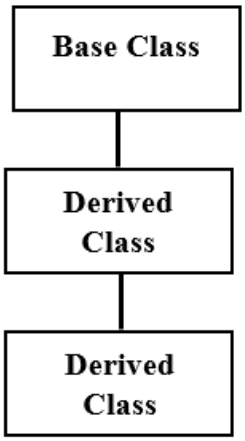


Figure 4- Multilevel Inheritance

Syntax for declaring base and derived class in multilevel inheritance:

```
class A
{
properties;
   methods;
};
class B: public A
{
properties;
   methods;
};
class C: public B
{
properties;
   methods;
```

```
};
```

Here, class **B** is derived from the base class **A** and the class **C** is derived from the derived class **B**.

Example 1:

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

class A
{
   public:
     void display()
     {
        cout<<"Base class content.";
     }
};

class B : public A
{

};

class C : public B
{

};

int main()
{
   C obj;
   obj.display();
   return 0;
}
```

Output:

```
Base class content.
```

- In this program, class ***C*** is derived from class ***B*** (which is derived from base class A).
- The ***obj*** object of class C is defined in the **main()** function.
- When the ***display()*** function is called, display() in class **A** is executed. It's because there is no **display()** function in class **C** and class **B**.

- The compiler first looks for the **display()** function in class **C**. Since the function doesn't exist there, it looks for the function in class **B** (as C is derived from B).
- The function also doesn't exist in class **B**, so the compiler looks for it in class **A** (as B is derived from A).

Example 2:

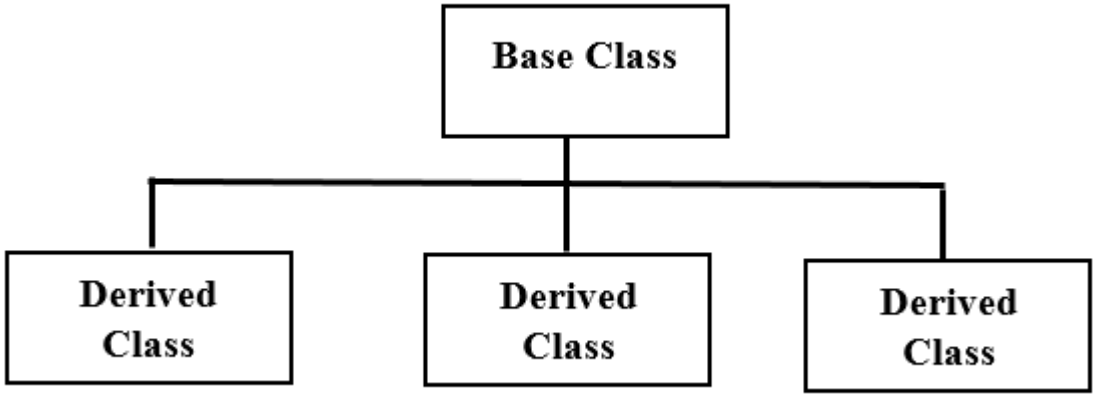
```
// C++ program to implement
// Multilevel Inheritance
#include <iostream>
using namespace std;

// 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
```

10.4.4. Hierarchical Inheritance

If more than one class is inherited from the base class, it's known as hierarchical inheritance. In hierarchical inheritance, all features that are common in child classes are included in the base class.



Syntax of Hierarchical Inheritance

```
class base_class {
   properties;
   methods;
}
class first_derived_class: public base_class {
   properties;
   methods;
}
class second_derived_class: public base_class {
    properties;
   methods;
}
class third_derived_class: public base_class {
   properties;
   methods;
}
```

Example 1:

```
#include <iostream>
using namespace std;
class A {
public:
  A(){
    cout<<"Constructor of A class"<<endl;
  }
};
class B: public A {
public:
  B(){
```

```
    cout<<"Constructor of B class"<<endl;
  }
};
class C: public A{
public:
  C(){
    cout<<"Constructor of C class"<<endl;
  }
};
int main() {
   //Creating object of class C
   C obj;
   return 0;
}
```

Output:

```
Constructor of A class
Constructor of C class
```

Example 2:

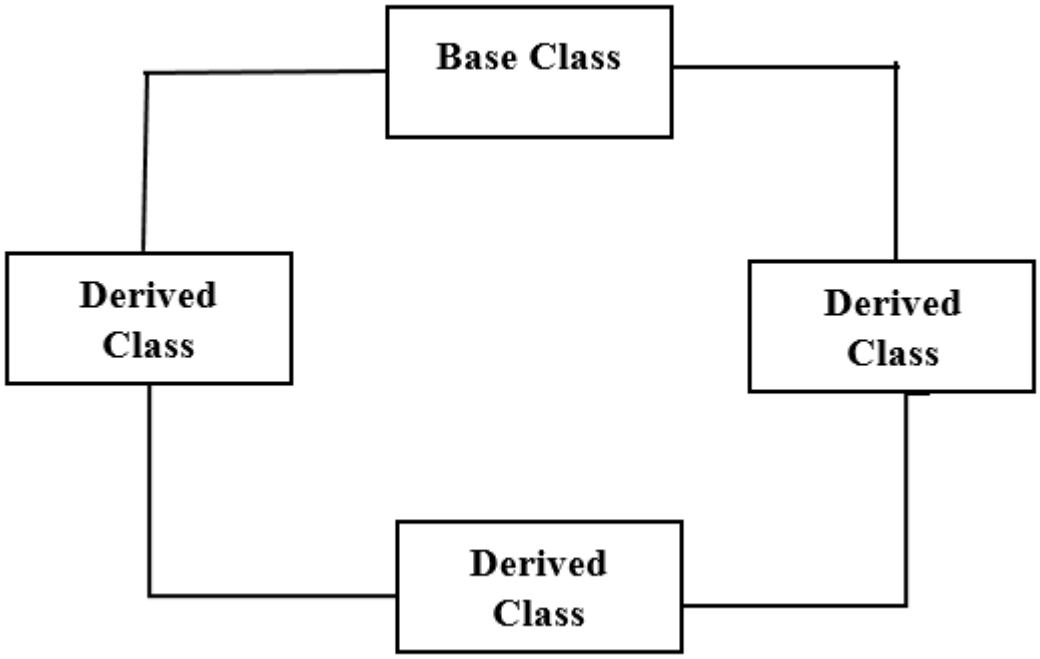
```
// C++ program to implement
// Hierarchical Inheritance
#include <iostream>
using namespace std;

// 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;
}
```

Output:

```
This is a Vehicle
This is a Vehicle
```

10.4.5. Hybrid Inheritance



Hybrid inheritance is a combination of more than one type of inheritance. For example, A child and parent class relationship that follows multiple and hierarchical inheritance both can be called hybrid inheritance.

Example 1:

```
#include<iostream>
using namespace std;
int a,b,c,d,e;
class A
{
protected:
```

```
public:
 void getab()
 {
 cout<<"\nEnter a and b value:";
 cin>>a>>b;
 }
};

class B:public A    {
protected:
public:
void getc()
 {
 cout<<"Enter c value:";
 cin>>c;
 }
};

class C
{
protected:
public:
 void getd()
 {
 cout<<"Enter d value:";
 cin>>d;
 }
};

class D:public B,public C
{
protected:
public:
 void result()
 {
 getab();    getc();
 getd();    e=a+b+c+d;
 cout<<"\n Addition is :"<<e;
 }
};

int main()
{
 D d1;
 d1.result();

 return 0;
}
```

Output:

```
Enter a and b value:10
20
Enter c value:12
Enter d value:10

 Addition is :52
```

Example 2:

```
// 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;
}
```

Output:

```
This is a Vehicle
Fare of Vehicle
```

Summary

- One of the keys to the power of object-oriented programming is achieving software reusability through inheritance.
- The programmer can designate that the new class is to inherit the data members and member functions of a previously defined base class. In this case, the new class is referred to as a derived class.
- With single inheritance, a class is derived from only one base class. With multiple inheritance, a derived class inherits from multiple (possibly unrelated) base classes.
- A derived class normally adds data members and member functions of its own, so a derived class generally has a larger definition than its base class. A derived class is more specific than its base class and normally represents fewer objects.
- A derived class cannot access the private members of its base class; allowing this would violate the encapsulation of the base class. A derived class can, however, access the public and protected members of its base class.
- A derived-class constructor always calls the constructor for its base class first to create and initialize the derived class's base-class members.
- Inheritance can be accomplished from existing class libraries.
- Someday most software will be constructed from standardized reusable components, as most hardware is constructed today.
- The implementor of a derived class does not need access to the source code of a base class, but it does need the interface to the base class and the base class's object code.
- An object of a derived class can be treated as an object of its corresponding public base class. However, the reverse is not true.
- A base class exists in a hierarchical relationship with its singly derived classes.
- A class can exist by itself. When that class is used with the mechanism of inheritance, it becomes either a base class that supplies attributes and behaviors to other classes, or the class becomes a derived class that inherits those attributes and behaviors.
- An inheritance hierarchy can be arbitrarily deep within the physical limitations of a particular system.
- Hierarchies are useful tools for understanding and managing complexity. With software becoming increasingly complex, C++ provides mechanisms for supporting hierarchical structures through inheritance and polymorphism.
- An explicit cast can be used to convert a base-class pointer to a derived-class pointer. Such a pointer should not be dereferenced unless it actually points to an object of the derived class type.
- A base class may be either a direct base class of a derived class or an indirect base class of a derived class. A direct base class is explicitly listed where the derived class is declared. An indirect base class is not explicitly listed; rather it is inherited from several levels up the class hierarchy tree.
- When a base-class member is inappropriate for a derived class, we may simply redefine that member in the derived class.
- It is important to distinguish between is a relationships and has a relationships. In a has a relationship, a class object has an object of another class as a member. In an is a relationship, an object of a derived-class type may also be treated as an object of the base-class type. Is a is inheritance. Has a is composition.

Review Questions

1. What is inheritance?
2. How to implement inheritance ?
3. What is Base class ?
4. What is Subclass?
5. What is the difference between public and private access specifier ?
6. What are the advantages of inheritance ?
7. What are the types of inheritance ?
8. What is the difference between inheritance and polymorphism ?
9. Is inheritance possible in C ?

Chapter 14

TEMPLATES

Contents

14.1. Introduction

This chapter introduces advanced C++ features i.e Templates. It is a new concept which enable us to define generic classes and templates functions and thus provides support for templates generic programming. Generic programming is an approach where generic types are used as parameters in algorithms so that they work for a variety of suitable data types and data structures. A template can be used to create a family of classes or functions. For example, a class template for an array class would enable us to create arrays of various data types such as int array and float Array. Similarly, we can define a template for a function, say mul(), that would help us create various versions of mul() for multiplying int, float and double type values.

A template can be considered as a kind of macro. When an object of a specific type is defined for actual use, the template definition for that class is substituted with the required data type. Since a template is defined with a parameter that would be replaced by a specified data type at the time of actual use of the class or function, the templates are sometimes called parameterized classes or functions,

Templates make it possible to use one function or class to handle many different data types. The template concept can be used in two different ways: with functions and with classes.

Working of templates

Templates are expanded at compiler time. This is like macros. The difference is, compiler does type checking before template expansion. The idea is simple, source code contains only function/class, but compiled code may contain multiple copies of same function/class.

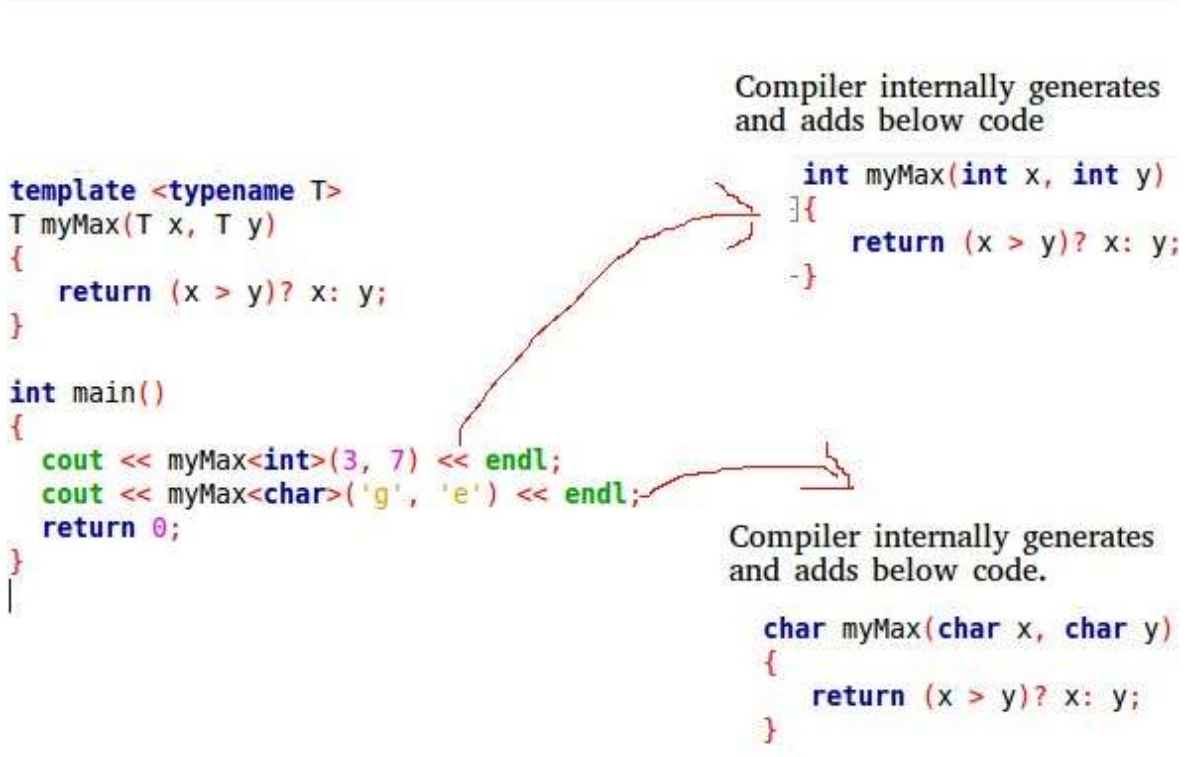


Figure 1: How templates work

14.2. Class Template

A class template provides a specification for generating classes based on parameters. *Class templates* are generally used to implement containers. A class template is instantiated by passing a given set of types to it as template arguments. Here is an example of a class, MyTemplate, that can store one element of any type and that has just one member function *divideBy2*, which divides its value by 2.

Example 1:

```
template <class T>
class MyTemplate {
T element;
public:
MyTemplate (T arg) {element=arg;}
T divideBy2 () {return element/2;}
};
```

It is also possible to define a different implementation of a template for a specific type. This is called *Template Specialization*. For the template given above, we find that a different implementation for type *char* will be more useful, so we write a function *printElement* to print the *char* element:

```
// class template specialization:
template <>
class MyTemplate <char> {
char element;
public:
MyTemplate (char arg) {element=arg;}
char printElement ()  {
return element;  }   };
```

Example 2:

```
#include <iostream>
using namespace std;
template <typename T>
class Array {
private:
	T *ptr;
	int size;
public:
	Array(T arr[], int s);
	void print();	};
template <typename T>
Array<T>::Array(T arr[], int s) {
	ptr = new T[s];
	size = s;
	for(int i = 0; i < size; i++)
		ptr[i] = arr[i];	}
template <typename T>
void Array<T>::print() {
	for (int i = 0; i < size; i++)
		cout<<" "<<*(ptr + i);
	cout<<endl;	}
int main() {
	int arr[5] = {1, 2, 3, 4, 5};
	Array<int> a(arr, 5);
	a.print();
	return 0;
}
```

Output:

```
1  2  3  4  5
```

Like normal parameters, we can pass more than one data types as arguments to templates. The following example demonstrates the same:

Example 3:

```
#include<iostream>
using namespace std;
template<class T, class U>
class A {
        T x;
        U y;
public:
        A() { cout<<"Constructor Called"<<endl; }
};
int main() {
A<char, char> a;
A<int, double> b;
return 0;
}
```

Output:

```
Constructor Called
Constructor Called
```

14.3. Function Templates

Function templates are special functions that can operate with generic types. This allows us to create a function template whose functionality can be adapted to more than one type or class without repeating the entire code for each type.

In C++ this can be achieved using template parameters. A template parameter is a special kind of parameter that can be used to pass a type as argument: just like regular function parameters can be used to pass values to a function, template parameters allow to pass also types to a function. These function templates can use these parameters as if they were any other regular type.

The format for declaring function templates with type parameters is:

```
template <class identifier> function_declaration;
template <typename identifier> function_declaration;
```

The only difference between both prototypes is the use of either the keyword class or the keyword typename. Its use is indistinct, since both expressions have exactly the same meaning and behave exactly the same way.

For example, to create a template function that returns the greater one of two objects we could use:

```
template <class myType>
myType GetMax (myType a, myType b) {
 return (a>b?a:b);            }
```

To use this function template we use the following format for the function call:

function_name <type> (parameters);

For example, to call GetMax to compare two integer values of type int we can write:

```
int x,y;
GetMax <int> (x,y);
```

When the compiler encounters this call to a template function, it uses the template to automatically generate a function replacing each appearance of myType by the type passed as the actual template parameter (int in this case) and then calls it. This process is automatically performed by the compiler and is invisible to the programmer.

Example 4:

```
// function template
#include <iostream>
using namespace std;

template <class T>
T GetMax (T a, T b) {
  T result;
  result = (a>b)? a : b;
  return (result);
}

int main () {
  int i=5, j=6, k;
  long l=10, m=5, n;
  k=GetMax<int>(i,j);
  n=GetMax<long>(l,m);
  cout << k << endl;
  cout << n << endl;
  return 0;
}
```

In this case, we have used T as the template parameter name instead of myType because it is shorter and in fact is a very common template parameter name. But you can use any identifier you like.

In the example above we used the function template GetMax() twice. The first time with arguments of type int and the second one with arguments of type long. The compiler has instantiated and then called each time the appropriate version of the function.

As you can see, the type T is used within the GetMax() template function even to declare new objects of that type:

T result;

Therefore, result will be an object of the same type as the parameters a and b when the function template is instantiated with a specific type.

In this specific case where the generic type T is used as a parameter for GetMax the compiler can find out automatically which data type has to instantiate without having to explicitly specify it within angle brackets (like we have done before specifying <int> and <long>). So we could have written instead:

```
int i,j;
GetMax (i,j);
```

Since both i and j are of type int, and the compiler can automatically find out that the template parameter can only be int. This implicit method produces exactly the same result:

```
// function template II
#include <iostream>
using namespace std;

template <class T>
T GetMax (T a, T b) {
  return (a>b?a:b);
}

int main () {
  int i=5, j=6, k;
  long l=10, m=5, n;
  k=GetMax(i,j);
  n=GetMax(l,m);
  cout << k << endl;
  cout << n << endl;
  return 0;
}
```

In this case, we called our function template GetMax() without explicitly specifying the type between angle-brackets <>. The compiler automatically determines what type is needed on each call.

Because our template function includes only one template parameter (class T) and the function template itself accepts two parameters, both of this T type, we cannot call our function template with two objects of different types as arguments:

```
int i;
long l;
k = GetMax (i,l);
```

What the Compiler Does

What does the compiler do when it sees the template keyword and the function definition that follows it? Well, nothing right away. The function template itself doesn't cause the compiler to generate any code. It can't generate code because it doesn't know yet what data type the function will be working with. It simply remembers the template for possible future use. Code generation doesn't take place until the function is actually called (invoked) by a statement within the program. In TEMPABS this happens in expressions like abs(int1) in the statement cout << "\nabs(" << int << ")=" << abs(int1);

When the compiler sees such a function call, it knows that the type to use is int, because that's the type of the argument int1. So it generates a specific version of the abs() function for type int, substituting int wherever it sees the name T in the function template. This is called instantiating the

function template, and each instantiated version of the function is called a template function. (That is, a template function is a specific instance of a function template. Isn't English fun?) The compiler also generates a call to the newly instantiated function, and inserts it into the code where abs(int1) is. Similarly, the expression abs(lon1) causes the compiler to generate a version of abs() that operates on type long and a call to this function, while the abs(dub1) call generates a function that works on type double. Of course, the compiler is smart enough to generate only one version of abs() for each data type. Thus, even though there are two calls to the int version of the function, the code for this version appears only once in the executable code. Simplifying the Listing Notice that the amount of RAM used by the program is the same whether we use the template approach or actually write three separate functions. The template approaches simply saves us from having to type three separate functions into the source file. This makes the listing shorter and easier to understand. Also, if we want to change the way the function works, we need to make the change in only one place in the listing instead of three places. The Deciding Argument The compiler decides how to compile the function based entirely on the data type used in the function call's argument (or arguments). The function's return type doesn't enter into this decision. This is similar to the way the compiler decides which of several overloaded functions to call.

14.4. Function Templates with Multiple Arguments

This one takes three arguments: two that are template arguments and one of a basic type. The purpose of this function is to search an array for a specific value. The function returns the array index for that value if it finds it, or –1 if it can't find it. The arguments are a pointer to the array, the value to search for, and the size of the array. In main() we define four different arrays of different types, and four values to search for. We treat type char as a number. Then we call the template function once for each array. Here's the listing for TEMPFIND:

Example 5:

In the given example below, we name the template argument atype. It appears in two of the function's arguments: as the type of a pointer to the array, and as the type of the item to be matched. The third function argument, the array size, is always type int; it's not a template argument. Here's the output of the program: 5 in chrArray: index=2 6 in intArray: index=-1 11 in lonArray: index=4 4 in dubArray: index=-1

The compiler generates four different versions of the function, one for each type used to call it. It finds a 5 at index 2 in the character array, does not find a 6 in the integer array, and so on.

```
// template used for function that finds number in array
#include <iostream>
using namespace std;
//function returns index number of item, or -1 if not found
template <class atype>
int find(atype* array, atype value, int size)          {
for(int j=0; j<size; j++)
if(array[j]==value)
return j;
return -1;          }
char chrArr[] = {1, 3, 5, 9, 11, 13}; //array
char ch = 5; //value to find
int intArr[] = {1, 3, 5, 9, 11, 13};
int in = 6;
long lonArr[] = {1L, 3L, 5L, 9L, 11L, 13L};
long lo = 11L;
double dubArr[] = {1.0, 3.0, 5.0, 9.0, 11.0, 13.0};
double db = 4.0;
int main()
{
cout << "\n 5 in chrArray: index=" << find(chrArr, ch, 6);
cout << "\n 6 in intArray: index=" << find(intArr, in, 6);
cout << "\n11 in lonArray: index=" << find(lonArr, lo, 6);
cout << "\n 4 in dubArray: index=" << find(dubArr, db, 6);
cout << endl;
return 0;
}
```

14.5. Difference between class template and Function Template

The name of a template class is a compound name consisting of the template name and the full template argument list enclosed in angle braces. Any references to a template class must use this complete name. For example:

```
template <class T, int range> class ex
{
    T a;
     int r;
    // ...
};
//...
ex<double,20> obj1;     // valid
ex<double> obj2;        // error
ex obj3;                // error
```

C++ requires this explicit naming convention to ensure that the appropriate class can be generated.

A template function, on the other hand, has the name of its function template and the particular function chosen to resolve a given template function call is determined by the type of the calling arguments. In the following example, the call min(a,b) is effectively a call to min(int a, int b), and the call min(af, bf) is effectively a call to min(float a, float b):

```
// This example illustrates a template function.

template<class T> T min(T a, T b)
{
    if (a < b)
        return a;
    else
        return b;
}
void main()
{
    int a = 0;
```

```
    int b = 2;
    float af = 3.1;
    float bf = 2.9;
    cout << "Here is the smaller int " << min(a,b) << endl;
    cout << "Here is the smaller float " << min(af, bf) << endl;
}
```

14.6. Template Overloading

You can use more than one template argument in a function template. For example, suppose you like the idea of the find() function template, but you aren't sure how large an array it might be applied to. If the array is too large then type long would be necessary for the array size, instead of type int. On the other hand, you don't want to use type long if you don't need to. You want to select the type of the array size, as well as the type of data stored, when you call the function. To make this possible, you could make the array size into a template argument as well. We'll call it btype:

```
template <class atype, class btype>
btype find(atype* array, atype value, btype size)
{
for(btype j=0; j<size; j++) //note use of btype
if(array[j]==value)
return j;
return static_cast<btype>(-1);
}
```

Now you can use either type int or type long (or even a user-defined type) for the size, whichever is appropriate. The compiler will generate different functions based not only on the type of the array and the value to be searched for, but also on the type of the array size. Note that multiple template arguments can lead to many functions being instantiated from a single template. Two such arguments, if there were six basic types that could reasonably be used for each one, would allow the creation of 36 functions. This can take up a lot of memory if the functions are large. On the other hand, you don't instantiate a version of the function unless you actually call it.

Summary:

- C++ supports a mechanism known as template to implement the concept of generic programming.
- Templates allows us to generate a family of classes or a family of functions to handle different data types.
- Template classes and functions eliminate code duplication for different types and thus make the program development easier and more manageable,
- We can use multiple parameters in both the class templates and function templates.
- A specific class created from a class template is called a template class and the process of creating a template class is known as instantiation. Similarly, a specific function created from a function template is called a template function.
- Like other functions, template functions can be overloaded.
- Member functions of a class template must be defined as function templates using the parameters of the class template,
- We may also use non-type parameters such basic or derived data types as arguments templates.

Review Questions:

1. 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;
}
```

2. Output of following program? Assume that the size of char is 1 byte and size of int is 4 bytes, and there is no alignment done by the compiler.

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

template<class T, class U>
class A  {
   T x;
   U y;
   static int count;
};

int main()  {
  A<char, char> a;
  A<int, int> b;
  cout << sizeof(a) << endl;
  cout << sizeof(b) << endl;
  return 0;
}
```

3. Output?

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

template <int i>
void fun()
{
  i = 20;
  cout << i;
}

int main()
{
  fun<10>();
  return 0;
}
```

4. What is output of below program?

```
#include < iostream >
#include < string >
using namespace std;
template < typename T >
void print_mydata(T output)
{
cout << output << endl;
}
int main()
{
double d = 5.5;
string s("Hello World");
print_mydata( d );
print_mydata( s );
return 0;
}
```

Chapter 5-

ABSTRACT BASE CLASS

Contents

5.1.Introduction

An interface describes the behavior or capabilities of a C++ class without committing to a particular implementation of that class. The C++ interfaces are implemented using abstract classes and these abstract classes should not be confused with data abstraction which is a concept of keeping implementation details separate from associated data.

Syntax and Structure:

```
//declaring abstract base class
class base_class
 {
   virtual return_type func_name() = 0; //pure virtual function
 }
```

Sample Program to demonstrate Abstract base class

```
//pure_virtual_func.cpp
#include <iostream>
using namespace std;
class base_class
{
 public:
  virtual void display() = 0;
};
class derived_class : public base_class
{
 public:
 void display()
 {
  cout<<"This is simple illustration of abstract class and pure virtual function";
 }
};
int main()
{
  derived_class obj;
  obj.display();
 return 0;
}
```

Output:

```
This is simple illustration of abstract class and pure virtual function
```

An abstract class is a class that is designed to be specifically used as a base class. An abstract class contains at least one pure virtual function. A pure virtual function is specified by placing "= 0" in its declaration as follows −

Declaration:

```
class Box {
   public:
      // pure virtual function
      virtual double getVolume() = 0;
      private:
      double length;      // Length of a box
      double breadth;     // Breadth of a box
      double height;      // Height of a box
};.
```

The purpose of an abstract class (often referred to as an ABC) is to provide an appropriate base class from which other classes can inherit. Abstract classes cannot be used to instantiate objects and serves only as an interface. Attempting to instantiate an object of an abstract class causes a compilation error. Thus, if a subclass of an ABC needs to be instantiated, it has to implement each of the virtual functions, which means that it supports the interface declared by the ABC. Failure to override a pure virtual function in a derived class, then attempting to instantiate objects of that class, is a compilation error. Classes that can be used to instantiate objects are called concrete classes.

Example 1:

```
#include <iostream>

using namespace std;

// Base class
class Shape {
   public:
      // pure virtual function providing interface framework.
      virtual int getArea() = 0;
      void setWidth(int w) {
         width = w;
      }

      void setHeight(int h) {
```

```
      height = h;
    }

  protected:
    int width;
    int height;
};

// Derived classes
class Rectangle: public Shape {
  public:
    int getArea() {
      return (width * height);
    }
};

class Triangle: public Shape {
  public:
    int getArea() {
      return (width * height)/2;
    }
};

int main(void) {
  Rectangle Rect;
  Triangle  Tri;

  Rect.setWidth(5);
  Rect.setHeight(7);
```

```
   // Print the area of the object.
   cout << "Total Rectangle area: " << Rect.getArea() << endl;

   Tri.setWidth(5);
   Tri.setHeight(7);

   // Print the area of the object.
   cout << "Total Triangle area: " << Tri.getArea() << endl;
   return 0;
}
```

Output:

```
Total Rectangle area: 35
Total Triangle area: 17
```

In above example it can been seen how an abstract class defined an interface in terms of getArea() and two other classes implemented same function but with different algorithm to calculate the area specific to the shape.

The main difference between an abstract base class and a regular polymorphic class is that because in abstract base classes at least one of its members lacks implementation we cannot create instances (objects) of it. But a class that cannot instantiate objects is not totally useless.

Example 2:

```
// abstract base class
#include <iostream>
using namespace std;
class CPolygon {
 protected:
 int width, height;
 public:
 void set_values (int a, int b)
 { width=a; height=b; }
 virtual int area (void) =0;
 };
```

```
class CRectangle: public CPolygon {
 public:
 int area (void)
 { return (width * height); }
 };
class CTriangle: public CPolygon {
 public:
 int area (void)
 { return (width * height / 2); }
 };
int main () {
 CRectangle rect;
 CTriangle trgl;
 CPolygon * ppoly1 = &rect;
 CPolygon * ppoly2 = &trgl;
 ppoly1->set_values (4,5);
 ppoly2->set_values (4,5);
 cout << ppoly1->area() << endl;
 cout << ppoly2->area() << endl;
 return 0;
}
```

Output:

```
20
10
```

If you review the program you will notice that we refer to objects of different but related classes using a unique type of pointer (CPolygon*). This can be tremendously useful. For example, now we can create a function member of the abstract base class CPolygon that is able to print on screen the result of the area() function even though CPolygon itself has no implementation for this function:

Example 3:

```
// pure virtual members can be called from the abstract base class
#include <iostream>
using namespace std;
class CPolygon {
 protected:
 int width, height;
 public:
 void set_values (int a, int b)
 { width=a; height=b; }
 virtual int area (void) =0;
 void printarea (void)
 { cout << this->area() << endl; }
 };
class CRectangle: public CPolygon {
 public:
 int area (void)
 { return (width * height); }
 };
class CTriangle: public CPolygon {
 public:
 int area (void)
 { return (width * height / 2); }
 };
int main () {
 CRectangle rect;
 CTriangle trgl;
 CPolygon * ppoly1 = &rect;
 CPolygon * ppoly2 = &trgl;
 ppoly1->set_values (4,5);
 ppoly2->set_values (4,5);
```

```
 ppoly1->printarea();
 ppoly2->printarea();
 return 0;
}
```

Output:

```
20
10
```

Example 4:

```
// dynamic allocation and polymorphism
#include <iostream>
using namespace std;
class CPolygon {
 protected:
 int width, height;
 public:
 void set_values (int a, int b)
 { width=a; height=b; }
 virtual int area (void) =0;
 void printarea (void)
 { cout << this->area() << endl; }
 };
class CRectangle: public CPolygon {
 public:
 int area (void)
 { return (width * height); }
 };
class CTriangle: public CPolygon {
```

```
 public:
 int area (void)
 { return (width * height / 2); }
 };
int main () {
 CPolygon * ppoly1 = new CRectangle;
 CPolygon * ppoly2 = new CTriangle;
 ppoly1->set_values (4,5);
 ppoly2->set_values (4,5);
 ppoly1->printarea();
 ppoly2->printarea();
 delete ppoly1;
 delete ppoly2;
 return 0;
}
```

Output:

```
20
10
```

We can have pointers and references of abstract class type. For example the following program works fine.

```
#include <iostream>
using namespace std;
class Base {
public:
        virtual void show() = 0;
};

class Derived : public Base {
public:
        void show() { cout << "In Derived \n"; }
};
int main(void)
{
```

```
        Base* bp = new Derived();
        bp->show();
        return 0;
}
```

Output:

In Derived

5.2. Pure Abstract Base Class

An abstract class is one in which there is a declaration but no definition for a member function. The way this concept is expressed in C++ is to have the member function declaration assigned to zero. However, C++ allows you to create a special kind of virtual function called a pure virtual function (or abstract function) that has no body at all! A pure virtual function simply acts as a placeholder that is meant to be redefined by derived classes.

To create a pure virtual function, rather than define a body for the function, we simply assign the function the value 0.

Declaration:

```
class PureAbstractClass         {
public:
  virtual void AbstractMemberFunction() = 0;          };
```

Sample Program:

```
class Base
{
public:
   const char* sayHi() { return "Hi"; } // a normal non-virtual function

   virtual const char* getName() { return "Base"; } // a normal virtual function

   virtual int getValue() = 0; // a pure virtual function
```

```
    int doSomething() = 0; // Compile error: can not set non-virtual functions to 0
};
```

When we add a pure virtual function to our class, we are effectively saying, “it is up to the derived classes to implement this function”.

Using a pure virtual function has two main consequences: First, any class with one or more pure virtual functions becomes an abstract base class, which means that it can not be instantiated! Consider what would happen if we could create an instance of Base:

```
int main()
{
    Base base; // We can't instantiate an abstract base class, but for the sake of example, pretend this was allowed
    base.getValue(); // what would this do?
}
```

A pure Abstract class has only abstract member functions and no data or concrete member functions. In general, a pure abstract class is used to define an interface and is intended to be inherited by concrete classes. It's a way of forcing a contract between the class designer and the users of that class. The users of this class must declare a matching member function for the class to compile.

Example 1:

```
class DrawableObject
 {
  public:
    virtual void Draw(GraphicalDrawingBoard&) const = 0; //draw to GraphicalDrawingBoard
 };


 class Triangle : public DrawableObject
 {
 public:
   void Draw(GraphicalDrawingBoard&) const; //draw a triangle
 };


 class Rectangle : public DrawableObject
```

```
{
public:
  void Draw(GraphicalDrawingBoard&) const; //draw a rectangle
};

class Circle : public DrawableObject
{
public:
  void Draw(GraphicalDrawingBoard&) const; //draw a circle
};

typedef std::list<DrawableObject*> DrawableList;

DrawableList drawableList;
GraphicalDrawingBoard drawingBoard;

drawableList.pushback(new Triangle());
drawableList.pushback(new Rectangle());
drawableList.pushback(new Circle());

for(DrawableList::const_iterator iter = drawableList.begin(),
   endIter = drawableList.end();
   iter != endIter;
   ++iter)
{
  DrawableObject *object = *iter;
  object->Draw(drawingBoard);
}
```

Example 2:

```
#include <string>
class Animal
```

```
{
protected:
    std::string m_name;

    Animal(std::string name)
        : m_name(name)
    {
    }

public:
    std::string getName() { return m_name; }
    virtual const char* speak() { return "???"; }
};

class Cat: public Animal
{
public:
    Cat(std::string name)
        : Animal(name)
    {
    }

    virtual const char* speak() { return "Meow"; }
};

class Dog: public Animal
{
public:
    Dog(std::string name)
        : Animal(name)
    {
    }

    virtual const char* speak() { return "Woof"; }
};
```

There are two problems with this:

1) The constructor is still accessible from within derived classes, making it possible to instantiate an Animal object.

2) It is still possible to create derived classes that do not redefine function speak().

For example:

```
#include <iostream>
class Cow: public Animal
{
public:
    Cow(std::string name)
        : Animal(name)
```

```
    {
    }

    // We forgot to redefine speak
};

int main()
{
    Cow cow("Betsy");
    std::cout << cow.getName() << " says " << cow.speak() << '\n';
}
```

Output:

```
Betsy says ???
```

5.3. Rules of Abstract Class

1) As we have seen that any class that has a pure virtual function is an abstract class.

2) We cannot create the instance of abstract class. For example: If I have written this line Animal obj; in the above program, it would have caused compilation error.

3) We can create pointer and reference of base abstract class points to the instance of child class. For **example**, this is valid:

```
Animal *obj = new Dog();
obj->sound();
```

4) Abstract class can have constructors.

5) If the derived class does not implement the pure virtual function of parent class then the derived class becomes abstract.

Summary

A pure virtual function is a member function that has no definition. A pure virtual function is indicated by the word virtual and the notation =0 in the member function declaration. A class with one or more pure virtual functions is called an abstract class.

We can't create an object of abstract class.

An abstract class is a type and it can be used as a base class to derive other classes. However, you cannot create an object of an abstract class type (unless it is an object of some derived class).

You can assign an object of a derived class to a variable of its base class (or any ancestor class), but the member variables that are not in the base class are lost. This is known as the slicing problem.

If the domain type of the pointer pAncestor is a base class for the domain type of the pointer pDescendent, then the following assignment of pointers is allowed:

pAncestor = pDescendent;

If we do not override the pure virtual function in derived class, then derived class also becomes abstract class.

Review Questions

1. What is abstract class?
2. What is the output of this program?

```
#include <iostream>
using namespace std;
class Test {
protected:
	int width, height;

public:
	void set_values(int a, int b)
	{
		width = a;
		height = b;
	}
	virtual int area(void) = 0;
};
class r : public Test {
public:
	int area(void)
	{
		return (width * height);
	}
};
class t : public Test {
public:
	int area(void)
	{
		return (width * height / 2);
	}
};
int main()
{
	r rect;
	t trgl;
```

```
    Test* ppoly1 = □
    Test* ppoly2 = &trgl;
    ppoly1->set_values(4, 5);
    ppoly2->set_values(4, 5);
    cout << ppoly1->area();
    cout << ppoly2->area();
    return 0;
}
```

3. What is the output of this program?

```
#include <iostream>
using namespace std;
class sample {
public:
virtual void example() = 0;
};
class Ex1 : public sample {
public:
void example()
{
        cout << "GeeksForGeeks";
}
};
class Ex2 : public sample {
public:
void example()
{
        cout << " is awesome";
}
};
int main()
{
sample* arra[2];
Ex1 e1;
Ex2 e2;
arra[0] = &e1;
arra[1] = &e2;
arra[0]->example();
arra[1]->example();
}
```

Chapter 9-

ARRAYS AND STRUCTURES

Contents

Introduction

C++ is an object-oriented programming language, and a vector is an example of a software object. C++ began as an extension of the C programming language, but C does not directly support object-oriented programming. Consequently, C does not have vectors available to represent sequence types. The C language uses a more primitive construct called an array. True to its C roots, C++ supports arrays as well as vectors. Some C++ libraries use arrays in instead of vectors. In addition, C++ programs can use any of the large number of C libraries that have been built up over the past 40+ years, and many of these libraries process arrays. While a more modern construct like std::vector may be better suited for many of the roles an array has played in the past, it nonetheless is important for C++ programmers to be familiar with arrays.

An array is a variable that refers to a block of memory that, like a vector, can hold multiple values simultaneously. An array has a name, and the values it contains are accessed via their position within the block of memory designated for the array. Also like a vector, the elements within an array must all be of the same type. Arrays may be local or global variables. Arrays are built into the core language of both C and C++. This means you do not need to add any #include directives to use an array within a program.

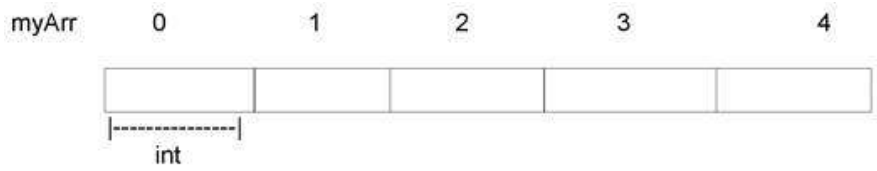
Defining of an Array

An array contains multiple objects of identical types stored sequentially in memory. The individual objects in an array, referred to as array elements, can be addressed using a number, the so-called index or subscript. An array is also referred to as a vector.

An array is a series of elements of the same type placed in contiguous memory locations that can be individually referenced by adding an index to a unique identifier.

That means that, for example, we can store 5 values of type int in an array without having to declare 5 different variables, each one with a different identifier. Instead of that, using an array we can store 5 different values of the same type, int for example, with a unique identifier.

For example, an array to contain 5 integer values of type int called myArr could be represented like this:



where each blank panel represents an element of the array, that in this case are integer values of type int. These elements are numbered from 0 to 4 since in arrays the first index is always 0, independent of its length.

A typical declaration for an array in C++ is:

Syntax:

<datatype> array_name [size];

where datatype is a valid type (like int, float…), name is a valid identifier and the size(which is always enclosed in square brackets [ ]), specifies how many elements the array will hold.

Therefore, in order to declare an array called myArr as shown in the above diagram, it is as simple as:

int myArr [5];

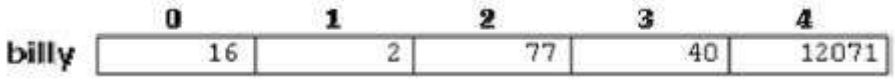
Initializing arrays.

When declaring a regular array of local scope (within a function, for example), if we do not specify otherwise, its elements will not be initialized to any value by default, so their content will be undetermined until we store some value in them. The elements of global and static arrays, on the other hand, are automatically initialized with their default values, which for all fundamental types this means they are filled with zeros.

In both cases, local and global, when we declare an array, we have the possibility to assign initial values to each one of its elements by enclosing the values in braces { }. For example:

int myArr [5] = {16, 2, 77, 40, 12071};

This declaration would have created an array like this:



The amount of values between braces { } must not be larger than the number of elements that we declare for the array between square brackets [ ]. For example, in the example of array myArr we have declared that it has 5 elements and in the list of initial values within braces { } we have specified 5 values, one for each element.

When an initialization of values is provided for an array, C++ allows the possibility of leaving the square brackets empty [ ]. In this case, the compiler will assume a size for the array that matches the number of values included between braces { }:

int myArr [ ] = { 16, 2, 77, 40, 12071 };

After this declaration, array myArr would be 5 ints long, since we have provided 5 initialization values.

Accessing the values of an array.

In any point of a program in which an array is visible, we can access the value of any of its elements individually as if it was a normal variable, thus being able to both read and modify its value. The format is as simple as:

Syntax: array_name[index] Following the previous examples in which myArr had 5 elements and each of those elements was of type int, the name which we can use to refer to each element is the following:

For example, to store the value 75 in the third element of myArr, we could write the following statement:

myArr[2] = 75;

and, for example, to pass the value of the third element of myArr to a variable called a, we could write:

a = myArr[2];

Therefore, the expression myArr[2] is for all purposes like a variable of type int.

C++ Program to Calculate Product and Sum of all Elements in an Array

Program:

```
#include<iostream>

using namespace std;

int main ()

{

   int arr[10], n, i, sum = 0, pro = 1;

   cout << "Enter the size of the array : ";

   cin >> n;

   cout << "\nEnter the elements of the array : ";

   for (i = 0; i < n; i++)

   cin >> arr[i];

   for (i = 0; i < n; i++)

   {

      sum += arr[i];

      pro *= arr[i];

   }

   cout << "\nSum of array elements : " << sum;

   cout << "\nProduct of array elements : " << pro;

   return 0;

}
```

Output:

```
Enter the size of the array : 5

Enter the elements of the array : 1

2

3
```

4

5

Sum of array elements: 15

Product of array elements: 120

Program Explanation

1. The user is initially asked to enter the size of the array and the value is stored in ‘n’.

2. The variables ‘sum’ and ‘pro’ are initialized as 0 and 1 respectively.

3. The user is asked to enter the array elements. Using a for loop, the elements are stored in the array ‘arr’.

4. Taking a for loop and initializing ‘i’ as 0, the sum and product are calculated and stored in sum and pro respectively.

5. i is incremented in every iteration. The loop continues till i is less than n.

6. The result is then printed.

2.

```
#include <iostream>
#include<conio.h>
using namespace std;
#define ARRAY_SIZE 5
int main()
{
   int numbers[ARRAY_SIZE], i;
         cout<<"Reading Array with Position : \n";
   for (i = 0; i < ARRAY_SIZE; i++)
   {
                  cout<<"Enter the Number : "<< (i+1) <<"  : ";
      cin>>numbers[i];
   }
   cout<<"\nPrinting Array: \n";
   for (i = 0; i < ARRAY_SIZE; ++i)
   {
      cout<<numbers[i] << "\n";
   }
```

```
    getch();
    return 0;
}
```

Output :

Reading Array with Position :

Enter the Number : 1 : 1

Enter the Number : 2 : 2

Enter the Number : 3 : 3

Enter the Number : 4 : 4

Enter the Number : 5 : 5

Printing Array:

1

2

3

4

5

Write a program which will copy contents of one array to another.

```
#include<stdio.h>
void main()
{
int A[5],B[5];
        // take 5 numbers from user
printf("Enter 5 numbers");
for( i=0; i<5; ++i )
     scanf("%d",&A[i]);

for( i=0 ;i<5;++i )
       B[i]=A[i];

printf("\n  Number in array A are :");
for( i=1 ;i<5 ;++i )
      printf(\n "%d",A[i]);
```

```
printf("\n  Number in array B are :");
for( i=1 ;i<5 ;++i )
      printf(\n "%d",B[i]);

}
```

OUTPUT :

```
Enter 5 numbers
10
20
30
40
50
Number in Array A are
10
20
30
40
50
Number in Array B are
10
20
30
40
50
```

Multidimensional Arrays

An array in which data are arranged in the form of array of arrays is called multi-dimensional array. An array can have as much dimensions as required. However, two dimensional and three dimensional array are commonly used. A multi-dimensional array of dimension n is a collection of items that are accessed with the help of n subscript values. Generally, the arrays of three or more dimensions are not used because of their huge memory requirements and the complexities involved in their manipulation.

For Example:

int x[3][4];

Here, x is a two dimensional array. It can hold a maximum of 12 elements.

This array can be consider as table with 3 rows and each row has 4 columns as shown below.

	Column 1	Column 2	Column 3	Column 4
Row 1	x[0] [0]	x[0] [1]	x[0] [2]	x[0] [3]
Row 2	x[1] [0]	x[1] [1]	x[1] [2]	x[1] [3]
Row 3	x[2] [0]	x[2] [1]	x[2] [2]	x[2] [3]

Two-dimensional Arrays

The simplest form of the multidimensional array is the two-dimensional array. A two-dimensional array is, in essence, a list of one-dimensional arrays. To declare a two-dimensional integer array of size x,y, you would write something as follows −

datatype array_name[d1][d2];

Here, array_name is an array of type datatype and it has 2 dimensions. The number of elements in array_name is equal to d1*d2.

For Example:

For example,

int a[10][10]; // declares an integer array with 100 elements

float f[5][10]; // declares an float array with 50 elements

char n[5][50]; // declares an character array with 250 elements

A two-dimensional array can be think as a table, which will have x number of rows and y number of columns. Thus, every element in array a is identified by an element name of the form a[i][j], where a is the name of the array, and i and j are the subscripts that uniquely identify each element in a.

C++ program to enter elements of a matrix and display them.

Program:

```
#include <iostream>

#include <conio.h>

using namespace std;


int main()

{

   int arr[10][10],row,col,i,j;

   cout<<"Enter size of row and column: ";

   cin>>row>>col;
```

```
    cout<<"Enter elements of matrices(row wise)"<<endl;
    for(i=0;i<row;i++)
        for(j=0;j<col;j++)
            cin>>arr[i][j];
    cout<<"Displaying matrix"<<endl;
    for(i=0;i<row;i++)
    {
        for(j=0;j<col;j++)
            cout<<arr[i][j]<<"  ";
        cout<<endl;
    }
    getch();
    return 0;
}
```

Output:

Enter size of row and column: 3 2

Enter elements of matrices(row wise)

1 2 3 4 5 6

Displaying matrix

1 2

3 4

5 6

In this program, a two dimensional array is used to store the content of a matrix. The size of row and column is entered by user. Nested for loop is used to ask the content of elements of matrix and display them. The number of elements in the array (matrix) is equal to the product of size of row and column.

Three-dimensional Arrays

C++ Program to Store value entered by user in three dimensional array and display it.

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

```
    // This array can store upto 12 elements (2x3x2)
    int test[2][3][2];
    cout << "Enter 12 values: \n";
    // Inserting the values into the test array
    // using 3 nested for loops.
    for(int i = 0; i < 2; ++i)
    {
        for (int j = 0; j < 3; ++j)
        {
            for(int k = 0; k < 2; ++k )
            {
                cin >> test[i][j][k];
            }
        }
    }
    cout<<"\nDisplaying Value stored:"<<endl;
    // Displaying the values with proper index.
    for(int i = 0; i < 2; ++i)
    {
        for (int j = 0; j < 3; ++j)
        {
            for(int k = 0; k < 2; ++k)
            {
                cout << "test[" << i << "][" << j << "][" << k << "] = " << test[i][j][k] << endl;
            }
        }
    }
    return 0;
}
```

Ouput:

Enter 12 values:

1

2

3

4

5

6

7

8

9

0

1

2

Displaying Value stored:

test[0][0][0] = 1

test[0][0][1] = 2

test[0][1][0] = 3

test[0][1][1] = 4

test[0][2][0] = 5

test[0][2][1] = 6

test[1][0][0] = 7

test[1][0][1] = 8

test[1][1][0] = 9

test[1][1][1] = 0

test[1][2][0] = 1

test[1][2][1] = 2

C++ Program to Add Two Matrices using Multi-dimensional Arrays

```
#include <iostream>
using namespace std;
int main()
{
   int r, c, a[100][100], b[100][100], sum[100][100], i, j;
   cout << "Enter number of rows (between 1 and 100): ";
   cin >> r;
   cout << "Enter number of columns (between 1 and 100): ";
   cin >> c;
   cout << endl << "Enter elements of 1st matrix: " << endl;
   // Storing elements of first matrix entered by user.
   for(i = 0; i < r; ++i)
     for(j = 0; j < c; ++j)
     {
        cout << "Enter element a" << i + 1 << j + 1 << " : ";
        cin >> a[i][j];
     }
   // Storing elements of second matrix entered by user.
   cout << endl << "Enter elements of 2nd matrix: " << endl;
   for(i = 0; i < r; ++i)
     for(j = 0; j < c; ++j)
     {
        cout << "Enter element b" << i + 1 << j + 1 << " : ";
        cin >> b[i][j];
     }
   // Adding Two matrices
   for(i = 0; i < r; ++i)
      for(j = 0; j < c; ++j)
         sum[i][j] = a[i][j] + b[i][j];
```

```
    // Displaying the resultant sum matrix.
    cout << endl << "Sum of two matrix is: " << endl;
    for(i = 0; i < r; ++i)
        for(j = 0; j < c; ++j)
        {
            cout << sum[i][j] << "  ";
            if(j == c - 1)
                cout << endl;
        }
    return 0;
}
```

Output:

Enter number of rows (between 1 and 100): 2 3

Enter number of columns (between 1 and 100): 3

Enter elements of 1st matrix:

Enter element a11 : 1

Enter element a12 : 2

Enter element a13 : 3

Enter element a21 : 3

Enter element a22 : 3

Enter element a23 : 4

Enter element a31 : 5

Enter element a32 : 2

Enter element a33 : 3

Enter elements of 2nd matrix:

Enter element b11 : 1

Enter element b12 : 2

Enter element b13 : 4

Enter element b21 : 5

Enter element b22 : 6

Enter element b23 : 8

Enter element b31 : 9

Enter element b32 : 3

Enter element b33 : 1

Sum of two matrix is:

2 4 7

8 9 12

14 5 4

C++ Program to Display Largest Element of an array

```
#include <iostream>
using namespace std;
int main()
{
   int i, n;
   float arr[100];
   cout << "Enter total number of elements(1 to 100): ";
   cin >> n;
   cout << endl;
   // Store number entered by the user
   for(i = 0; i < n; ++i)
   {
     cout << "Enter Number " << i + 1 << " : ";
     cin >> arr[i];
   }
   // Loop to store largest number to arr[0]
   for(i = 1;i < n; ++i)
```

```
    {
      // Change < to > if you want to find the smallest element
      if(arr[0] < arr[i])
          arr[0] = arr[i];
    }
    cout << "Largest element = " << arr[0];
    return 0;
}
```

STRUCTURE

C wouldn't have been so popular had it been able to handle only all **int**s, or all **float**s or all **char**s at a time. In fact when we handle real world data, we don't usually deal with little atoms of information by themselves—things like integers, characters and such. Instead we deal with entities that are collections of things, each thing having its own attributes, just as the entity we call a 'book' is a collection of things such as title, author, call number, publisher, number of pages, date of publication, etc. As you can see all this data is dissimilar, for example author is a string, whereas number of pages is an integer.
For dealing with such collections, C provides a data type called 'structure'. A structure gathers together, different atoms of information that comprise a given entity.

Need of Structure :

We have seen earlier how ordinary variables can hold one piece of information and how arrays can hold a number of pieces of information of the same data type. These two data types can handle a great variety of situations. But quite often we deal with entities that are collection of dissimilar data types.

For example, suppose you want to store data about a book. You might want to store its name (a string), its price (a float) and number of pages in it (an int). If data about say 3 such books is to be stored, then we can follow two approaches:

1. Construct individual arrays, one for storing names, another for storing prices and still another for storing number of pages.
2. Use a structure variable.

So better option is option number 2.

Lets consider that we have been given a task where our program takes roll_no, marks of two subjects (dbms and maths), calculates total of marks (dbms and maths) and finds average of marks of five students. Such program will require five variables to store roll_no having variable name roll_no 1 to roll_no 5, five variables to store marks of dbms having variable name dbms1 to dbms5, five variables to store marks of maths having variable name variable maths1 to maths5, five variables to store total of marks of subject having variable name total1 to total5 and five variables to store average of marks

having name variable avg1 to avg5. This program will require to declare total 25 variables. Just imagine that if we have 100 students, then number of variable required will be more. Declaring this much amount of variable is difficult one.

column1	column2	column3	column4	Column5
roll_no1	dbms1	maths1	total1	avg1
roll_no2	dbms2	maths2	total2	avg2
roll_no3	dbms3	maths3	total3	avg3
roll_no4	dbms4	maths4	total4	avg4
roll_no5	dbms5	maths5	total5	avg5

Now, if we go column-wise, then we have to declare the array. Because array is collection of variables which have common characteristics and data type. But if we go row-wise, then you will find that it contains information regarding the student as roll_no, marks in subjects, total and avg. This information consists of five columns which may have different data types. As roll_no is int and avg is float. Every row contains one record.

To define one record, which contains different information of different data type, one has to use structure.

4.11 DEFINING A STRUCTURE

Structure declarations are somewhat difficult than array declarations, since a structure must be defined in terms of its individual members.

In general terms, the structure may be defined as

```
struct structure_name
   {
     member1;
    member2;
     ……..
    member_n;
  };
```

In the above declaration, **struct** is a reserved keyword, struct_name is the name given to the structure and member1, member2 to member_n are the members of the structure.

The member of the structure can be variable, array, pointer or any other structure. Every member of the array should have name and that should be different from other members.

The member of the structure can be variables, which can be declared as

data type variable name;

So, general form of structure can be rewritten as

```
struct structure_name
{
  data type variable name1;
  data type variable name2;
```

```
   ……..
  data type variable namen;
};
```

Example 4.63 : Declaration of structure student

```
struct student
{
 int roll_no;
 int dbms;
 int maths;
 int total;
 float avg;
};
```

The above example creates a structure whose name is student, which contains the five members: roll_no, dbms, maths, total and avg, out of which first four are of type int and last is of type float.

Once the structure is defined, the variable of the structure can be created by using the following syntax.

struct structure_name var1,var2;

where var1 and var2 are variables of type structure structure_name.

Example :

We have already created the structure student. Now we will create the variable of structure student.

struct student a,b;

The above statement will create variables a and b, both are of type student. Both a and b contain five members individually as roll_no, dbms, maths, total and avg as shown.

a

roll_no	dbms	maths	total	avg
2 bytes	2 bytes	2 bytes	2 bytes	4 bytes

b

roll_no	dbms	maths	total	avg
2 bytes	2 bytes	2 bytes	2 bytes	4 bytes

Notice in the above diagram that variables **a** and **b** consist of five members. Under a and b, it reserves 12 bytes of memory for each (addition of memory required by each variable as per their data type), and collectively it has been given name **a** and **b**.

Example 4.64 : Declare structure employee :

```
struct Employee
{
int emp_id;
char Emp_name[20];
float salary;
};
```

A variable of the structure can be declared at the end of structure declaration.

So Structure can be rewritten as

```
struct structure_name
  {
   data type variable name1;
   data type variable name2;
    ……..
   data type variable namen;
}var1,var2….varn;
```

Example 4.65 : Create some structures

```
struct student
{
  int roll_no;
  int dbms;
  int maths;
  int total;
  float avg;
}a, b;

struct account
{
  int account_number;
  char name[20];
  float balance;
}oldcustomer, newcustomer;
```

The above structure defines a structure of account having members as account number, name and balance, and creates two variables - oldcustomer and newcustomer of it. Here, oldcustomer and newcustomer have the same members individually defined by the structure.

Nested Structure :

A structure may be defined as a part of another structure as shown

Example 4.66 : This example illustrates the structures in one another structure.

```
struct date
{
  int dd;
  int mm;
  int yy;
};
struct account
{
  int account_number;
  char name[20];
  float balance;
  date dateofbirth;
}oldcustomer, newcustomer;
```

The second structure contains first structure as a part of it. The second structure defines one variable of structure date as a member of it.

Example 4.67 : Create structure student :

```
struct date
{
  int dd;
  int mm;
  int yy;
};

struct student
{
  int roll_no;
  char name[20];
   date dob;
};
```

Initialization of structure :

The member of the structure can be assigned with initial values as

struct structure_name var = {value1, value2, ……., valuen};

Example 4.68 : This example illustrates the assignment of initial value to structure.

```
struct date
```

```
{
  int day;
  int month;
  int year;
};

struct account
{
  int accountnumber;
  char name[20];
  float balance;
  date dateofbirth;
};
struct account oldcustomer={12009,”TANMAY”,12345.0,09,06,01}
```

The oldcustomer is a variable of type account, whose members are assigned initial values. The first member (account number) is initialized to integer value 12009, second member (name) is initialized with string value “TANMAY”. The third member (balance) is initialized with float value 12345.0 and the last member itself is a structure that contains three integer members (day, month, year). Therefore, the last member is assigned with value 09, 06, 01.

Example 4.69 : Define two structure date and account and initialize them :

```
struct date
{
  int day;
  int month;
  int year;
};

struct account
{
  int account_number;
  char name[20];
  float balance;
  date dateofbirth;
};
```

struct account a1={12,”Tanmay”,12000,09,06,01};

Rules for initialization

(1) We cannot initialize individual member inside structure. We have to initialize all member.

(2) The order of values enclosed in braces must match the order of members in the structure definition.

4.12 ARRAY OF STRUCTURE

It is also possible to declare the array of structure, i.e. an array in which each element is structure.

Example 4.70 : Following example declares the array of structure.

```
struct date
{
  int day;
  int month;
  int year;
};
struct account
{
  int account_number;
  char name[20];
  float balance;
  date dateofbirth;
} customer[100];
```

The above example declares an array having name customer of 100 elements. Here, each element of customer is a separate structure of type account.

It is also possible to declare array customer as separate declaration.

Example 4.71 : Following example declares the array of structure.

```
struct date
{
  int day;
  int month;
  int year;
};
struct account
{
  int account_number;
  char name[20];
  float balance;
  date dateofbirth;
};
void main( )
```

```
{
struct account customer[100];
}
```

An array of structure can be assigned initial values just as other array.

Example 4.72 :

```
struct date
{
  char name[20];
  int day;
  int month;
  int year;
};

struct date birthday []= {{“siddhi”,7, 10, 81},
                          {“tanmay”, 9, 6, 01},
                          {“Riddhi”, 27, 7, 01}
                         };
```

In this example, birthday is an array of structures whose size is unspecified. The initial values will define the size of the array, and the amount of memory required to store the array. Notice that each row in the variable declaration contains four constants. These constants represent the initial values, i.e., the name, month, day and year, for one array element. Since there are 3 rows (3 sets of constants), the array will contain 3 elements, numbered 0 to 2.

Remember that each structure is a self-contained entity with respect to member definitions. Thus, the same member name can be used in different structures to represent different data. In other words, the scope of a member name is limited to the particular structure within which it is defined.

Example 4.73 : Two different structures, called first and second, are declared below.

```
    struct first
      {
          float a;
          int b;
          char c;
       };
      struct second
       {
            char a;
            float b, c;
       };
```

Notice that the individual member names a, b and c appear in both structure declarations, but the associated data types are different. Thus, a represents a floating-point quantity in first and a character in second. Similarly, b represents an integer quantity in first and a floating-point quantity in second, whereas c represents a character in first and a floating-point quantity in second. This duplication of member names is permissible, since the scope of each set of member definitions is limited to its respective structure. Within each structure, the member names are distinct, as required.

The members of a structure are usually processed individually, as separate entities. Therefore, we must be able to access the individual structure members. A structure member can be accessed by writing

Variable.member

where variable refers to the name of a structure-type variable, and member refers to the name of a member within the structure. Notice the period (**.**) that separates the variable name from the member name. This period is an operator; it is a member of the highest precedence group, and its associativity is left to right.

Example 4.74 : Consider the following structure declarations.

```
struct account
        {
         int acct_no;
        char acct_type;
        char name[80];
        float balance;
        } oldcustomer;
```

In this example, oldcustomer is a structure variable of type account, we would write

oldcustomer.acct-no;

if we wanted to access the customer's account. Similarly, the customer's name and the customer's balance can be accessed by writing

oldcustomer.name;

and

oldcustomer.balance.

Since the period (.) is a member of the higher precedence group, this operator will take precedence over the unary operator as well as the various arithmetic, relational operators. Thus, the expression of the form **++ variable.member** is equivalent to ++ (variable. member); i.e. the ++ operator will apply to the structure member, not the entire structure variable. Similarly, the expression **&variable.member** is equivalent to &(variable. member), accesses the address of the memory, and uses to store data from keyboard into the member of variable.

Example 4.75 : Consider the structure declaration given below.

```
struct date
{
 int day;
```

```
  int month;
  int year;
};
struct account
{
  int account_number;
  char name[20];
  float balance;
  date dateofbirth;
}customer;
```

Expression	Interpretation
++customer.balance	Increment the value of customer.balance
−−customer.account number	Decrement the value of customer.account number
customer.balance++	Increment the value of customer.balance
customer.account number−−	Decrement the value of customer.account number

More complex expressions involving the repeated use of the period operator may also be written. For example, if a structure member itself is a structure, then a member of the embedded structure can be accessed by writing

variable.member. submember

where member refers to the name of the member within the outer structure, and submember refers to the name of the member within the embedded structure.

Summary

Array: a collection of a fixed number of components, all of the same data type

One-dimensional array: components are arranged in a list form

C++ does not allow functions to return a value of type array

Character array: an array whose components are of type char

C-strings are null-terminated ('\0‘) character arrays

Size of an array can be omitted if the array is initialized during declaration

Ways to process a two-dimensional array:

- Process entire array
- Row processing: process a single row at a time
- Column processing: process a single column at a time

Review Questions

1. Write a program to sort one dimensional array.
2. What is array?
3. How to declare and initialize single dimensional array?
4. Write a program to find greatest number from array.
5. Write a program for addition of 2×2 matrix.
6. Declare one dimensional, 5 element integer array and Initialize all values.
7. Explain declaration of string and initialization of string.
8. Declare one-dimensional, 5 elements integer array and initialize all values.
9. What is array? Explain how elements of array can be accessed.
10. Write a program to find smallest number in 5-element integer array.
11. What is Array? How to declare and initialize one dimensional array.
12. Give one example for declaring and initializing one dimensional array.
13. Write a program to sort elements of an array in alphabetical order.
14. What is two dimensional array? How is it declared and initialized? Give one Example.
15. Explain the declaration and initialization of 2D array with example.
16. Define two dimensional Array. Explain initialization of two dimensional Array.
17. What is structure? How to declare and initialize structure?
18. How to declare a structure and access the members of structures.
19. Explain structure declaration and initialization
20. With example Explain Array of Structure.
21. Write syntax to declare array of structure.
22. Explain Array of structure with an example.
23. Explain array of structure with an example.

Chap 01 Object oriented design methodology

1. **Introduction to Programming Languages**
2. **Generation of Programming Languages**
3. **Object-Oriented Design (OOD) Methodologies**
4. **Introduction to Object-Oriented Programming**

    4.1 An Object-Oriented Class

    4.2 An Object

    4.3 Encapsulation

    4.4 Inheritance

    4.5 Polymorphism

5. **Over-Loading**
6. **Over-Riding**
7. **Abstract class**
8. **Object-Oriented Analysis and Design**

1. Introduction to Programming Languages
What is Algorithm?

An algorithm (pronounced AL-go-rith-um) is a procedure or formula for solving a problem, based on conducting a sequence of specified actions. A computer program can be viewed as an elaborate algorithm. In mathematics and computer science, an algorithm usually means a small procedure that solves a recurrent problem.

Algorithms are widely used throughout all areas of IT (information technology). A search enginealgorithm, for example, takes search strings of keywords and operators as input, searches its associated database for relevant web pages, and returns results.

An encryption algorithm transforms data according to specified actions to protect it. A secret key algorithm such as the U.S. Department of Defense's Data Encryption Standard (DES), for example, uses the same key to encrypt and decrypt data. As long as the algorithm is sufficiently sophisticated, no one lacking the key can decrypt the data.

The word algorithm derives from the name of the mathematician, Mohammed ibn-Musa al-Khwarizmi, who was part of the royal court in Baghdad and who lived from about 780 to 850. Al-Khwarizmi's work is the likely source for the word algebra as well.

How Do Algorithms Work?

Let's take a closer look at an example.

A very simple example of an algorithm would be to find the largest number in an unsorted list of numbers. If you were given a list of five different numbers, you would have this figured out in no time, no computer needed. Now, how about five million different numbers? Clearly, you are going to need a computer to do this, and a computer needs an algorithm.

Below is what the algorithm could look like. Let's say the input consists of a list of numbers, and this list is called L. The number L1 would be the first number in the list, L2 the second number, etc. And we know the list is not sorted - otherwise, the answer would be really easy. So, the input to the algorithm is a list of numbers, and the output should be the largest number in the list.

The algorithm would look something like this:

Step 1: Let Largest = L1

This means you start by assuming that the first number is the largest number.

Step 2: For each item in the list:

This means you will go through the list of numbers one by one.

Step 3: If the item > Largest:

If you find a new largest number, move to step four. If not, go back to step two, which means you move on to the next number in the list.

Step 4: Then Largest = the item

This replaces the old largest number with the new largest number you just found. Once this is completed, return to step two until there are no more numbers left in the list.

Step 5: Return Largest

This produces the desired result.

Notice that the algorithm is described as a series of logical steps in a language that is easily understood. For a computer to actually use these instructions, they need to be written in a language that a computer can understand, known as a programming language.

In programming, algorithm is a set of well defined instructions in sequence to solve the problem.

Qualities of a good algorithm

1. Input and output should be defined precisely.
2. Each steps in algorithm should be clear and unambiguous.
3. Algorithm should be most effective among many different ways to solve a problem.
4. An algorithm shouldn't have computer code. Instead, the algorithm should be written in such a way that, it can be used in similar programming languages.

Algorithm Examples
Algorithm to add two numbers
Algorithm to find the largest among three numbers

Algorithm Examples
Algorithm to find all the roots of quadratic equation
Algorithm to find the factorial
Algorithm to check prime number
Algorithm of Fibonacci series

Examples Of Algorithms In Programming

Write an algorithm to add two numbers entered by user.

Step 1: Start

Step 2: Declare variables num1, num2 and sum.

Step 3: Read values num1 and num2.

Step 4: Add num1 and num2 and assign the result to sum.

sum←num1+num2

Step 5: Display sum

Step 6: Stop

Write an algorithm to find the largest among three different numbers entered by user.

Step 1: Start

Step 2: Declare variables a,b and c.

Step 3: Read variables a,b and c.

Step 4: If a>b

If a>c

Display a is the largest number.

Else

Display c is the largest number.

Else

If b>c

Display b is the largest number.

Else

Display c is the greatest number.

Step 5: Stop

Write an algorithm to find all roots of a quadratic equation ax2+bx+c=0.

Step 1: Start

Step 2: Declare variables a, b, c, D, x1, x2, rp and ip;

Step 3: Calculate discriminant

D←b2-4ac

Step 4: If D≥0

r1←(-b+√D)/2a

r2←(-b-√D)/2a

Display r1 and r2 as roots.

Else

Calculate real part and imaginary part

rp←b/2a

ip←√(-D)/2a

Display rp+j(ip) and rp-j(ip) as roots

Step 5: Stop

Write an algorithm to find the factorial of a number entered by user.

Step 1: Start

Step 2: Declare variables n,factorial and i.

Step 3: Initialize variables

factorial←1

i←1

Step 4: Read value of n

Step 5: Repeat the steps until i=n

5.1: factorial←factorial*i

5.2: i←i+1

Step 6: Display factorial

Step 7: Stop

Write an algorithm to check whether a number entered by user is prime or not.

Step 1: Start

Step 2: Declare variables n,i,flag.

Step 3: Initialize variables

flag←1

i←2

Step 4: Read n from user.

Step 5: Repeat the steps until i<(n/2)

5.1 If remainder of n÷i equals 0

flag←0

Go to step 6

5.2 i←i+1

Step 6: If flag=0

Display n is not prime

else

Display n is prime

Step 7: Stop

Write an algorithm to find the Fibonacci series till term≤1000.

Step 1: Start

Step 2: Declare variables first_term,second_term and temp.

Step 3: Initialize variables first_term←0 second_term←1

Step 4: Display first_term and second_term

Step 5: Repeat the steps until second_term≤1000

5.1: temp←second_term

5.2: second_term←second_term+first term

5.3: first_term←temp

5.4: Display second_term

Step 6: Stop

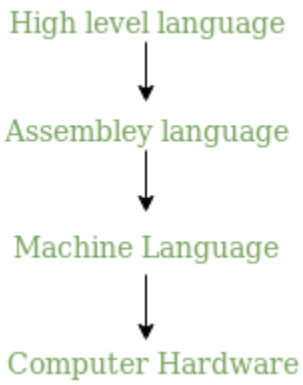
Algorithm is not the computer code. Algorithm are just the instructions which gives clear idea to you idea to write the computer code.

A computer is a computational device which is used to process the data under the control of a computer program. Program is a sequence of instruction along with data. While executing the program, raw data is processed into a desired output format. These computer programs are written in a programming language which are high level languages. High level languages are nearly human languages which are more

complex then the computer understandable language which are called machine language, or low level language.

Between high-level language and machine language there are assembly language also called symbolic machine code. Assembly language are particularly computer architecture specific. Utility program (**Assembler**) is used to convert assembly code into executable machine code. High Level Programming Language are portable but require Interpretation or compiling toconvert it into a machine language which is computer understood.

Hierarchy of Computer language –



Characteristics of a programming Language –

- A programming language must be simple, easy to learn and use, have good readability and human recognizable.
- Abstraction is a must-have Characteristics for a programming language in which ability to define the complex structure and then its degree of usability comes.
- A portable programming language is always preferred.
- Programming language's efficiency must be high so that it can be easily converted into a machine code and executed consumes little space in memory.
- A programming language should be well structured and documented so that it is suitable for application development.
- Necessary tools for development, debugging, testing, maintenance of a program must be provided by a programming language.
- A programming language should provide single environment known as Integrated Development Environment(IDE).
- A programming language must be consistent in terms of syntax and semantics.

2. Generation of Programming Languages

There are five generation of Programming languages.They are:
First Generation Languages :
These are low-level languages like machine language.
Second Generation Languages :

These are low-level assembly languages used in kernels and hardware drives.

Third Generation Languages :

These are high-level languages like C, C++, Java, Visual Basic and JavaScript.

Fourth Generation Languages :

These are languages that consist of statements that are similar to statements in the human language. These are used mainly in database programming and scripting. Example of these languages include Perl, Python, Ruby, SQL, MatLab(MatrixLaboratory).

Fifth Generation Languages :

These are the programming languages that have visual tools to develop a program. Examples of fifth generation language include Mercury, OPS5, and Prolog.

The first two generations are called low level languages. The next three generations are called high level languages.

3. Object-Oriented Design (OOD) Methodologies

the object-oriented approach combines data and processes (called methods) into single entities called objects. Objects usually correspond to the real things a system deals with, such as customers, suppliers, contracts, and invoices. Object-oriented models are able to thoroughly represent complex relationships and to represent data and data processing with a reliable notation, which allows an easier mix of analysis and design in a growth process. The aim of the Object-Oriented approach is to make system elements more modular, thus improving system quality and the efficiency of systems analysis and design.

In the Object-Oriented approach we tend to focus more on the behaviour of the system. The main feature we document is the Object or Class.

What is an Object?

An Object, as we have already stated, is something we hold data about. This is usually in the form of a noun, eg an apple is an object. It has attributes such as size and colour and taste.

What is a Class?

A Class is the description of a set of common objects, eg the apple would belong to the class Fruits, which all have similar group characteristics but also wide differences between them: ie another object, Orange, would also be part of the Fruit class but has different attributes of taste, colour and size from an apple.

A class will not occupy any memory space. Hence to work with the data represented by the class you must create a variable for the class, that is called an object.

When an object is created using the new operator, memory is allocated for the class in the heap, the object is called an instance and its starting address will be stored in the object in stack memory.

When an object is created without the new operator, memory will not be allocated in the heap, in other words an instance will not be created and the object in the stack contains the value **null**.

When an object contains null, then it is not possible to access the members of the class using that object.

4. Introduction to Object-Oriented Programming

Object-Oriented Programming (OOP) is the term used to describe a programming approach based on **objects** and **classes**. The object-oriented paradigm allows us to organise software as a collection of objects that consist of both data and behaviour. This is in contrast to conventional functional programming practice that only loosely connects data and behaviour.

Since the 1980s the word 'object' has appeared in relation to programming languages, with almost all languages developed since 1990 having object-oriented features. Some languages have even had object-oriented features retro-fitted. It is widely accepted that object-oriented programming is the most important and powerful way of creating software.

The object-oriented programming approach encourages:

- Modularisation: where the application can be decomposed into modules.
- Software re-use: where an application can be composed from existing and new modules.

4.1 An Object-Oriented Class

If we think of a real-world object, such as a television (as in Figure 1.1), it will have several features and properties:

- We do not have to open the case to use it.
- We have some controls to use it (buttons on the box, or a remote control).
- We can still understand the concept of a television, even if it is connected to a DVD player.
- It is complete when we purchase it, with any external requirements well documented.
- The TV will not crash!

In many ways this compares very well to the notion of a class.

Figure 1.1. The concept of a class - television example.

A class should:

- Provide a well-defined interface - such as the remote control of the television.
- Represent a clear concept - such as the concept of a television.
- Be complete and well-documented - the television should have a plug and should have a manual that documents all features.
- The code should be robust - it should not crash, like the television.

With a functional programming language (like C) we would have the component parts of the television scattered everywhere and we would be responsible for making them work correctly - there would be no case surrounding the electronic components.

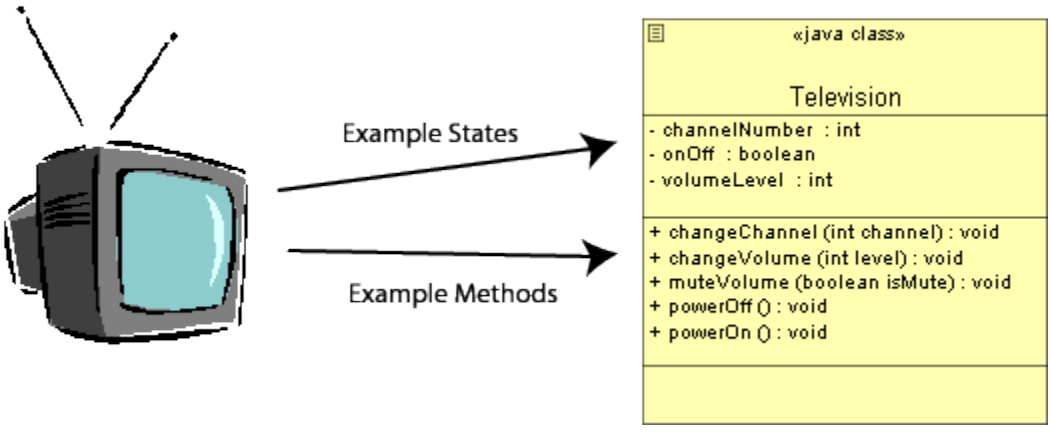
Humans use class based descriptions all the time - what is a duck? (Think about this, we will discuss it soon.)

Classes allow us a way to represent complex structures within a programming language. They have two components:

- **States** - (or data) are the values that the object has.
- **Methods** - (or behaviour) are the ways in which the object can interact with its data, the actions.

The notation used in Figure 1.2 on the right hand side is a Unified Modelling Language (UML) representation of the Television class for object-oriented modelling and programming.

Figure 1.2. The Television class example.



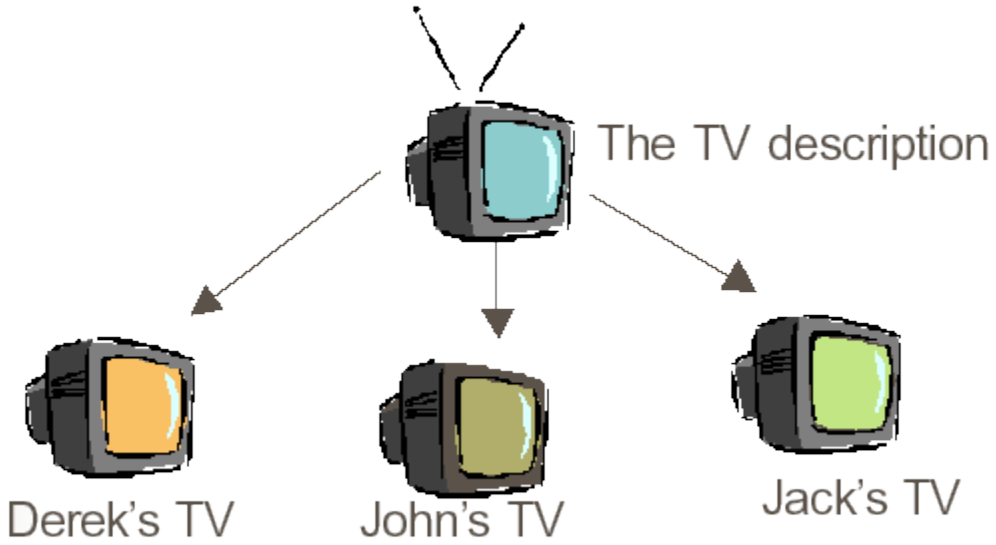
An instance of a class is called an **object**.

4.2 An Object

An **object** is an instance of a class. You could think of a class as the description of a concept, and an object as the realisation of this description to create an independent distinguishable entity. For example, in the case of the Television, the class is the set of plans (or blueprints) for a generic television, whereas a television object is the realisation of these plans into a real-world physical television. So there would be one set of plans (the class), but there could be thousands of real-world televisions (objects).

Objects can be concrete (a real-world object, a file on a computer) or could be conceptual (such as a database structure) each with its own individual identity. Figure 1.3 shows an example where the Television class description is realised into several television objects. These objects should have their own identity and are independent from each other. For example, if the channel is changed on one television it will not change on the other televisions.

Figure 1.3. The Television objects example.



4.3 Encapsulation

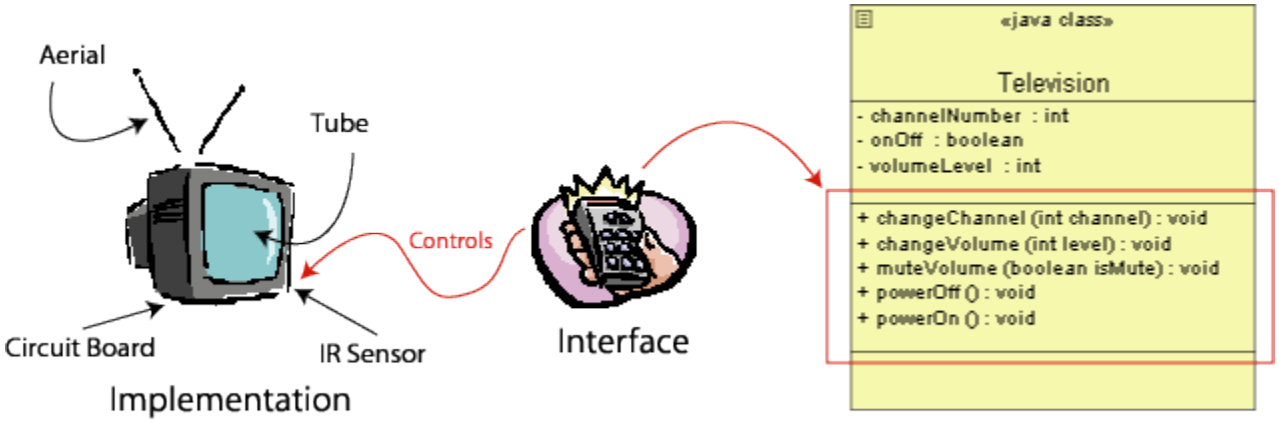
The object-oriented paradigm encourages encapsulation. Encapsulation is used to hide the mechanics of the object, allowing the actual implementation of the object to be hidden, so that we don't need to understand how the object works. All we need to understand is the interface that is provided for us.

You can think of this in the case of the Television class, where the functionality of the television is hidden from us, but we are provided with a remote control, or set of controls for interacting with the television, providing a high level of abstraction. So, as in Figure 1.4 there is no requirement to understand how the signal is decoded from the aerial and converted into a picture to be displayed on the screen before you can use the television.

There is a sub-set of functionality that the user is allowed to call, termed the interface. In the case of the television, this would be the functionality that we could use through the remote control or buttons on the front of the television.

The full implemenation of a class is the sum of the public interface plus the private implementation.

Figure 1.4. The Television interface example.

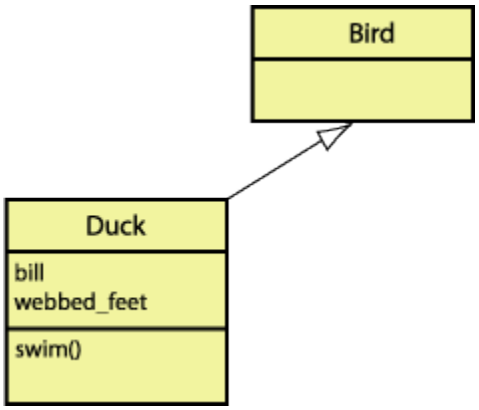


4.4 Inheritance

If we have several descriptions with some commonality between these descriptions, we can group the descriptions and their commonality using inheritance to provide a compact representation of these descriptions. The object-oriented programming approach allows us to group the commonalities and create classes that can describe their differences from other classes.

Humans use this concept in categorising objects and descriptions. For example you may have answered the question - "What is a duck?", with "a bird that swims", or even more accurately, "a bird that swims, with webbed feet, and a bill instead of a beak". So we could say that a Duck is a Bird that swims, so we could describe this as in Figure 1.6. This figure illustrates the inheritance relationship between a Duck and a Bird. In effect we can say that a Duck is a special type of Bird.

Figure 1.6. The Duck class showing inheritance.



4.5 Polymorphism

When a class inherits from another class it inherits both the states and methods of that class, so in the case of the Car class inheriting from the Vehicle class the Car class inherits the methods of the Vehicle class, such as

engineStart(), gearChange(), lightsOn() etc. The Car class will also inherit the states of the Vehicle class, such as isEngineOn, isLightsOn, numberWheels etc.

Polymorphism means "multiple forms". In OOP these multiple forms refer to multiple forms of the same method, where the exact same method name can be used in different classes, or the same method name can be used in the same class with slightly different paramaters. There are two forms of polymorphism, **over-riding** and **over-loading**.

5. Over-Loading

Over-Loading is the second form of polymorphism. The same method name can be used, but the number of parameters or the types of parameters can differ, allowing the correct method to be chosen by the compiler. For example:

```
add (int x, int y)
add (String x, String y)
```

are two different methods that have the same name and the same number of parameters. However, when we pass two String objects instead of two int variables then we expect different functionality. When we add two int values we expect an intresult - for example 6 + 7 = 13. However, if we passed two String objects we would expect a result of "6" + "7" = "67". In other words the strings should be concatenated.

The number of arguments can also determine which method should be run. For example:

```
channel()
channel(int x)
```

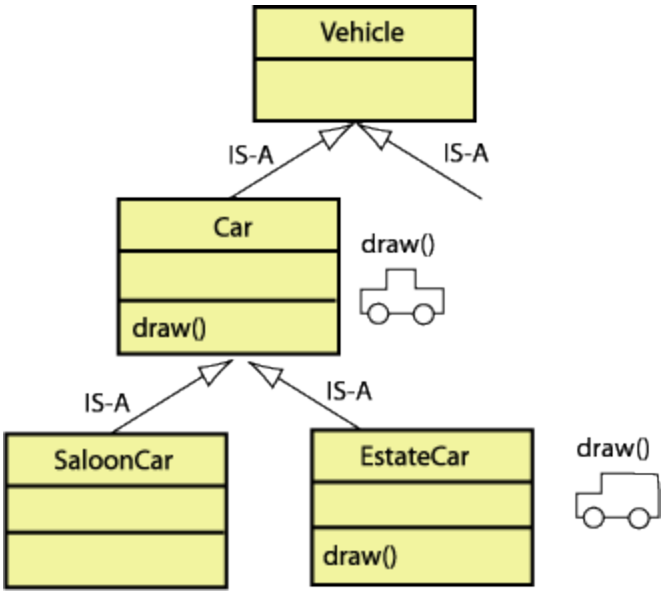
will provide different functionality where the first method may simply display the current channel number, but the second method will set the channel number to the number passed.

6. Over-Riding

As discussed, a derived class inherits its methods from the base class. It may be necessary to redefine an inherited method to provide specific behaviour for a derived class - and so alter the implementation. So, over-riding is the term used to describe the situation where the same method name is called on two different objects and each object responds differently.

Over-riding allows different kinds of objects that share a common behaviour to be used in code that only requires that common behaviour.

Figure 1.10. The over-ridden draw() method.



7. Abstract Classes

An abstract class is a class that is incomplete, in that it describes a set of operations, but is missing the actual implementation of these operations. Abstract classes:

- Cannot be instantiated.
- So, can only be used through inheritance.

For example: In the Vehicle class example previously the draw() method may be defined as abstract as it is not really possible to draw a generic vehicle. By doing this we are forcing all derived classes to write a draw() method if they are to be instantiated.

As discussed previously, a class is like a set of plans from which you can create objects. In relation to this analogy, an abstract class is like a set of plans with some part of the plans missing. E.g. it could be a car with no engine - you would not be able to make complete car objects without the missing parts of the plan.

Figure 1.11. The abstract draw() method in the Vehicle class.

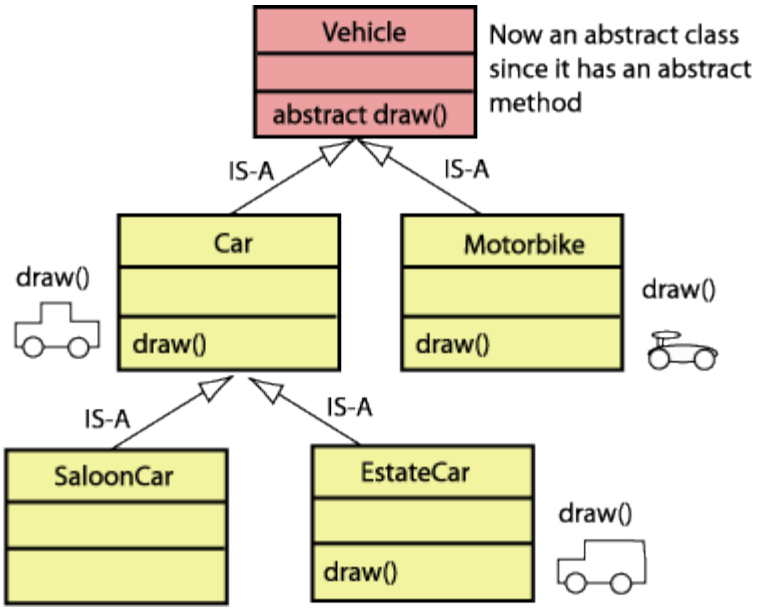


Figure 1.11 illustrates this example. The draw() has been written in all of the classes and has some functionality. The draw() in the Vehicle has been tagged as abstract and so this class cannot be instantiated - i.e. we cannot create an object of the Vehicle class, as it is incomplete. In Figure 1.11 the SaloonCar has no draw() method, but it does inherit a draw() method from the parent Car class. Therefore, it is possible to create objects of SaloonCar.

8. Object-Oriented Analysis and Design

As discussed previously, object-oriented programming has been around since the 1990s. Formal design processes when using objects involves many complex stages and are the debate of much research and development.

Why use the object-oriented approach?

Consider the general cycle that a programmer goes through to solve a programming problem:

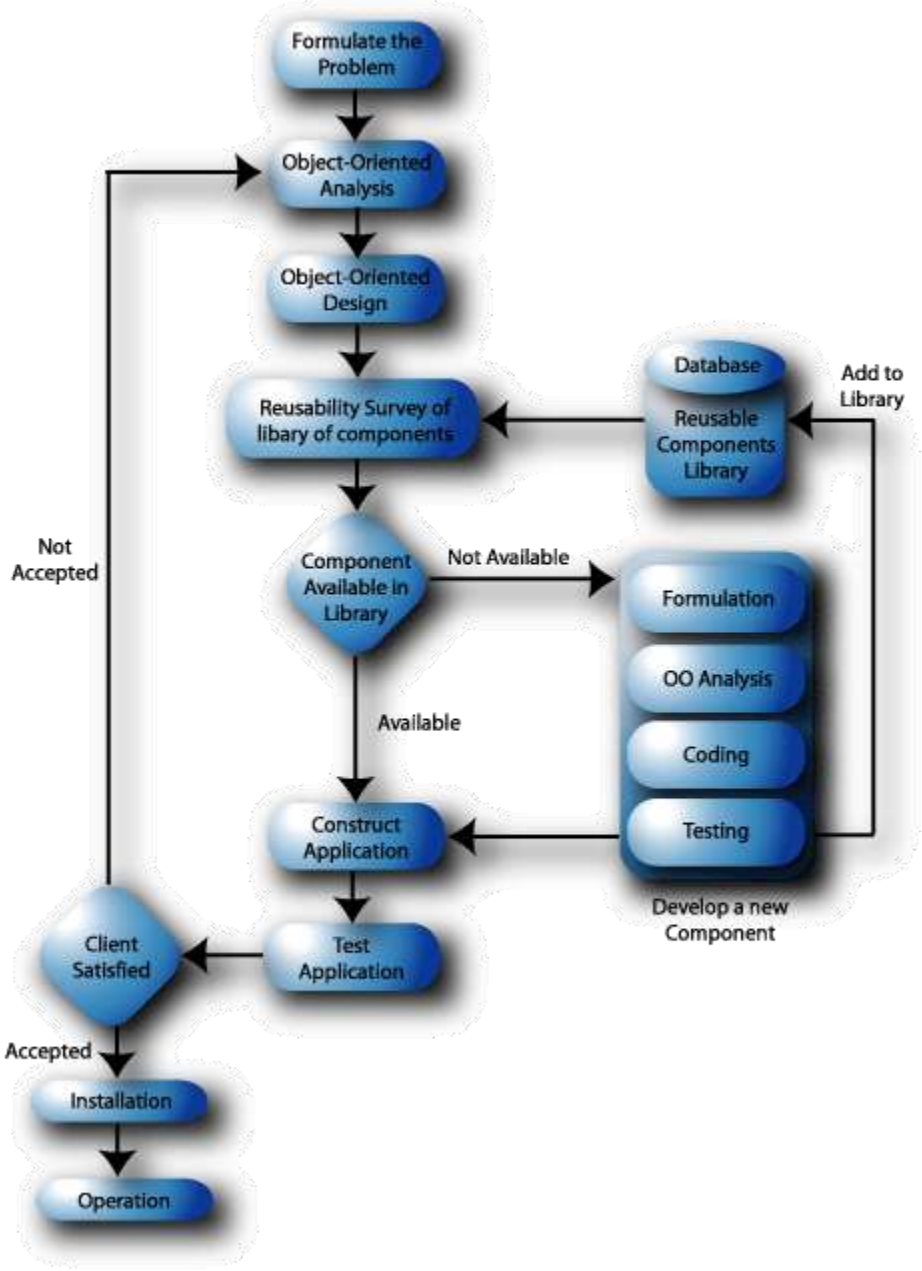
- **Formulate the problem** - The programmer must completely understand the problem.
- **Analyse the problem** - The programmer must find the important concepts of the problem.
- **Design** - The programmer must design a solution based on the analysis.
- **Code** - Finally the programmer writes the code to implement the design.

The Object-Oriented Design Model

One object-oriented methodology is based around the re-use of development modules and components. As such, a new development model is required that takes this re-use into account. The object-oriented model as shown in Figure 1.15 builds integration of existing software modules into the system development. A database of reusable components supplies the components for re-use. The object-oriented model starts with the formulation and analysis of the problem. The design phase is followed by a survey of the component library to see if any of the components can be re-used in the system development. If the component is not available in the library then a new component must be developed, involving formulation, analysis, coding and testing of the module. The new component is added to the library and used to construct the new application.

This model aims to reduce costs by integrating existing modules into development. These modules are usually of a higher quality as they have been tested in the field by other clients and should have been debugged. The development time using this model should be lower as there is less code to write.

Figure 1.15. The Object-Oriented Design Model



The object-oriented model should provide advantages over the other models, especially as the library of components that is developed grows over time.

9. Quiz

Reference

- http://ee402.eeng.dcu.ie/introduction/chapter-1---introduction-to-object-oriented-programming
- https://www.sqa.org.uk/e-learning/SDM01CD/page_05.htm

Chap 04. C++ for Loop

1. **Difference between Entry Controlled and Exit Controlled loop in C**
2. C++ for Loop Syntax
3. C++ while and do...while Loop
4. C++ break and continue Statement
5. C++ continue Statement
6. C++ if, if...else and Nested if...else**(Control Statements)**
7. C++ switch..case Statement
8. C++ goto Statement

Loops are used in programming to repeat a specific block of code. In this tutorial, you will learn to create a for loop in C++ programming (with examples).

1.What is the difference between Entry Controlled and Exit Controlled loop in C?

Entry and Exit Controlled Loop in C

Loops are the technique to repeat set of statements until given condition is true. C programming language has three types of loops - **1) while loop**, **2) do while loop** and **3) for loop**.

These loops controlled either at entry level or at exit level hence loops can be controlled two ways

1. Entry Controlled Loop
2. Exit Controlled Loop

Entry Controlled Loop

Loop, **where test condition is checked before entering the loop body**, known as **Entry Controlled Loop**.

Example: while loop, for loop

Exit Controlled Loop

Loop, **where test condition is checked after executing the loop body**, known as **Exit Controlled Loop**.

Example: do while loop

Consider the code snippets

Using while loop

```
1int count=100;
2while(count<50)
3printf("%d",count++);
```

Using for loop

```
1int count;
2for(count=100; count<50; count++)
3printf("%d",count);
```

In both code snippets **value of count is 100** and condition is **count<50**, which will check first, hence there is no output.

Using do while loop

```
1int count=100;
2do
3{
4printf("%d",count++);
5}while(count<50);
```

In this code snippet **value of count is 100** and test condition is **count<50** which is false yet loop body will be executed first then condition will be checked after that. Hence output of this program will be 100.

Differences Table

Entry Controlled Loop	Exit Controlled Loop
Test condition is checked first, and then loop body will be executed.	Loop body will be executed first, and then condition is checked.

If Test condition is false, loop body will not be executed.

for loop and while loop are the examples of Entry Controlled Loop.

Entry Controlled Loops are used when checking of test condition is mandatory before executing loop body.

If Test condition is false, loop body will be executed once.

do while loop is the example of Exit controlled loop.

Exit Controlled Loop is used when checking of test condition is mandatory after executing the loop body.

2.C++ for Loop Syntax

```
for(initializationStatement; testExpression; updateStatement) {
   // codes
}
```

where, only *testExpression* is mandatory.

How for loop works?

1. The initialization statement is executed only once at the beginning.
2. Then, the test expression is evaluated.
3. If the test expression is false, for loop is terminated. But if the test expression is true, codes inside body of for loop is executed and update expression is updated.
4. Again, the test expression is evaluated and this process repeats until the test expression is false.

Flowchart of for Loop in C++

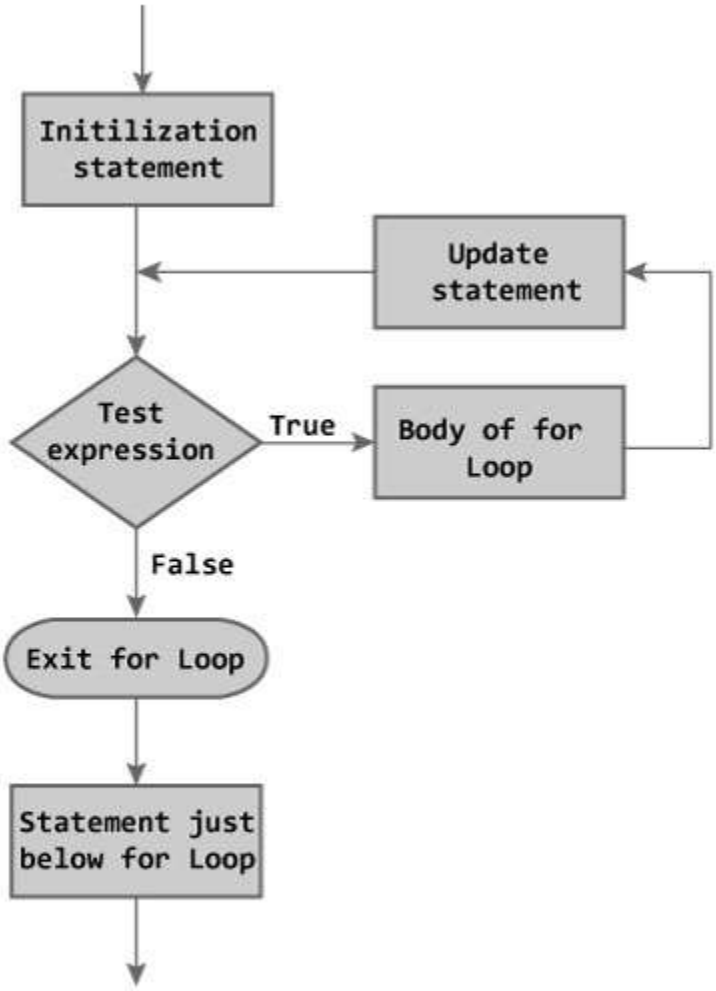


Figure: Flowchart of for Loop

Example 1: C++ for Loop

```
// C++ Program to find factorial of a number
// Factorial on n = 1*2*3*...*n

#include <iostream>
using namespace std;

int main()
{
   int i, n, factorial = 1;

   cout << "Enter a positive integer: ";
   cin >> n;
```

```
    for (i = 1; i <= n; ++i) {
        factorial *= i;   // factorial = factorial * i;
    }

    cout<< "Factorial of "<<n<<" = "<<factorial;
    return 0;
}
```

Output

```
Enter a positive integer: 5
Factorial of 5 = 120
```

In the program, user is asked to enter a positive integer which is stored in variable *n* (suppose user entered 5). Here is the working of for loop:

1. Initially, *i* is equal to 1, test expression is true, factorial becomes 1.
2. *i* is updated to 2, test expression is true, factorial becomes 2.
3. *i* is updated to 3, test expression is true, factorial becomes 6.
4. *i* is updated to 4, test expression is true, factorial becomes 24.
5. *i* is updated to 5, test expression is true, factorial becomes 120.
6. *i* is updated to 6, test expression is false, for loop is terminated.

In the above program, variable *i* is not used outside of the for loop. In such cases, it is better to declare the variable in for loop (at initialization statement).

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

int main()
{
    int n, factorial = 1;

    cout << "Enter a positive integer: ";
    cin >> n;

    for (int i = 1; i <= n; ++i) {
        factorial *= i;   // factorial = factorial * i;
    }

    cout<< "Factorial of "<<n<<" = "<<factorial;
    return 0;
}
```

Infinite for loop in C++

A loop is said to be infinite when it executes repeatedly and never stops. This usually happens by mistake. When you set the condition in for loop in such a way that it never return false, it becomes infinite loop.

For example:

```
#include <iostream>
using namespace std;
int main(){
   for(int i=1; i>=1; i++){
      cout<<"Value of variable i is: "<<i<<endl;
   }
   return 0;
}
```

This is an infinite loop as we are incrementing the value of i so it would always satisfy the condition i>=1, the condition would never return false.

Here is another example of infinite for loop:

```
// infinite loop
for ( ; ; ) {
   // statement(s)
}
```

Example: display elements of array using for loop

```
#include <iostream>
using namespace std;
int main(){
   int arr[]={21,9,56,99, 202};
   /* We have set the value of variable i
    * to 0 as the array index starts with 0
    * which means the first element of array
    * starts with zero index.
    */
   for(int i=0; i<5; i++){
      cout<<arr[i]<<endl;
   }
   return 0;
}
```

Output:

```
21
9
56
99
202
```

3.C++ while and do...while Loop

C++ while Loop

The syntax of a while loop is:

```
while (testExpression)
{
```

```
    // codes
}
```

where, *testExpression* is checked on each entry of the while loop.

How while loop works?

- The while loop evaluates the test expression.
- If the test expression is true, codes inside the body of while loop is evaluated.
- Then, the test expression is evaluated again. This process goes on until the test expression is false.
- When the test expression is false, while loop is terminated.

Flowchart of while Loop

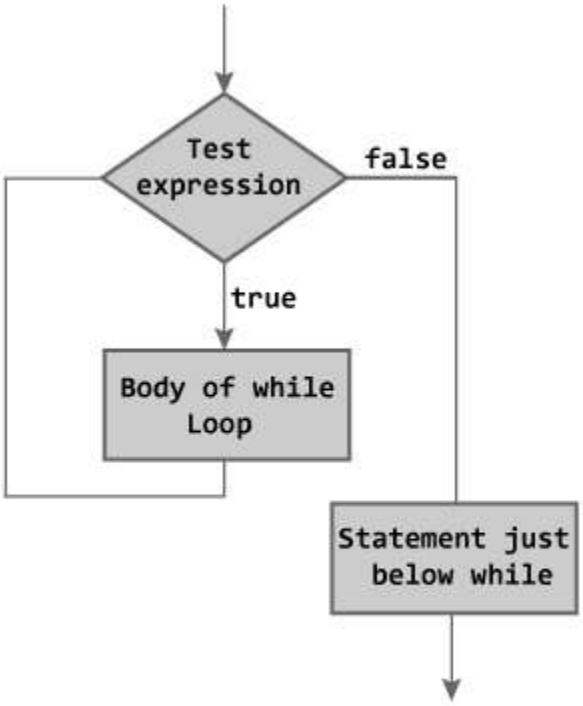


Figure: Flowchart of while Loop

Example 1: C++ while Loop

```
// C++ Program to compute factorial of a number
// Factorial of n = 1*2*3...*n

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

```
int main()
{
   int number, i = 1, factorial = 1;

   cout << "Enter a positive integer: ";
   cin >> number;

   while ( i <= number) {
      factorial *= i;      //factorial = factorial * i;
      ++i;
   }

   cout<<"Factorial of "<< number <<" = "<< factorial;
   return 0;
}
```

Output

```
Enter a positive integer: 4
Factorial of 4 = 24
```

In this program, user is asked to enter a positive integer which is stored in variable *number*. Let's suppose, user entered 4.

Then, the while loop starts executing the code. Here's how while loop works:

1. Initially, `i = 1`, test expression `i <= number` is true and factorial becomes 1.
2. Variable *i* is updated to 2, test expression is true, factorial becomes 2.
3. Variable *i* is updated to 3, test expression is true, factorial becomes 6.
4. Variable *i* is updated to 4, test expression is true, factorial becomes 24.
5. Variable *i* is updated to 5, test expression is false and while loop is terminated.

Infinite While loop

A while loop that never stops is said to be the infinite while loop, when we give the condition in such a way so that it never returns false, then the loops becomes infinite and repeats itself indefinitely.

An example of infinite while loop:

This loop would never end as I'm decrementing the value of i which is 1 so the condition i<=6 would never return false.

```
#include <iostream>
using namespace std;
int main(){
  int i=1; while(i<=6) {
    cout<<"Value of variable i is: "<<i<<endl; i--;
  }
}
```

Example: Displaying the elements of array using while loop

```
#include <iostream>
using namespace std;
int main(){
   int arr[]={21,87,15,99, -12};
   /* The array index starts with 0, the
    * first element of array has 0 index
    * and represented as arr[0]
    */
   int i=0;
   while(i<5){
      cout<<arr[i]<<endl;
      i++;
   }
}
```

Output:

```
21
87
15
99
-12
```

C++ do...while Loop

The do...while loop is a variant of the while loop with one important difference. The body of do...while loop is executed once before the test expression is checked.

The syntax of do..while loop is:

```
do {
   // codes;
}
while (testExpression);
```

How do...while loop works?

- The codes inside the body of loop is executed at least once. Then, only the test expression is checked.
- If the test expression is true, the body of loop is executed. This process continues until the test expression becomes false.
- When the test expression is false, do...while loop is terminated.

Flowchart of do...while Loop

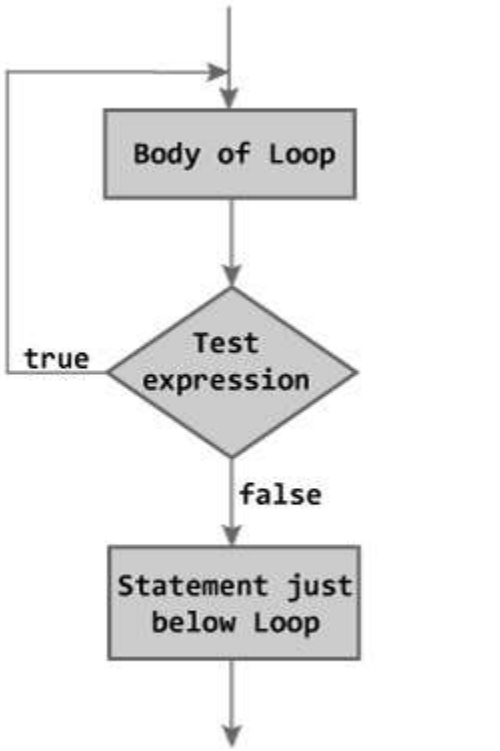


Figure: Flowchart of do...while Loop

Example 2: C++ do...while Loop

```
// C++ program to add numbers until user enters 0

#include <iostream>
using namespace std;

int main()
{
   float number, sum = 0.0;

   do {
      cout<<"Enter a number: ";
      cin>>number;
      sum += number;
   }
   while(number != 0.0);

   cout<<"Total sum = "<<sum;

   return 0;
}
```

Output

```
Enter a number: 2
Enter a number: 3
Enter a number: 4
Enter a number: -4
Enter a number: 2
Enter a number: 4.4
Enter a number: 2
Enter a number: 0
```

4.C++ break and continue Statement

In C++, there are two statements break; and continue; specifically to alter the normal flow of a program.

Sometimes, it is desirable to skip the execution of a loop for a certain test condition or terminate it immediately without checking the condition.

For example: You want to loop through data of all aged people except people aged 65. Or, you want to find the first person aged 20.

In scenarios like these, continue; or a break; statement is used.

C++ break Statement

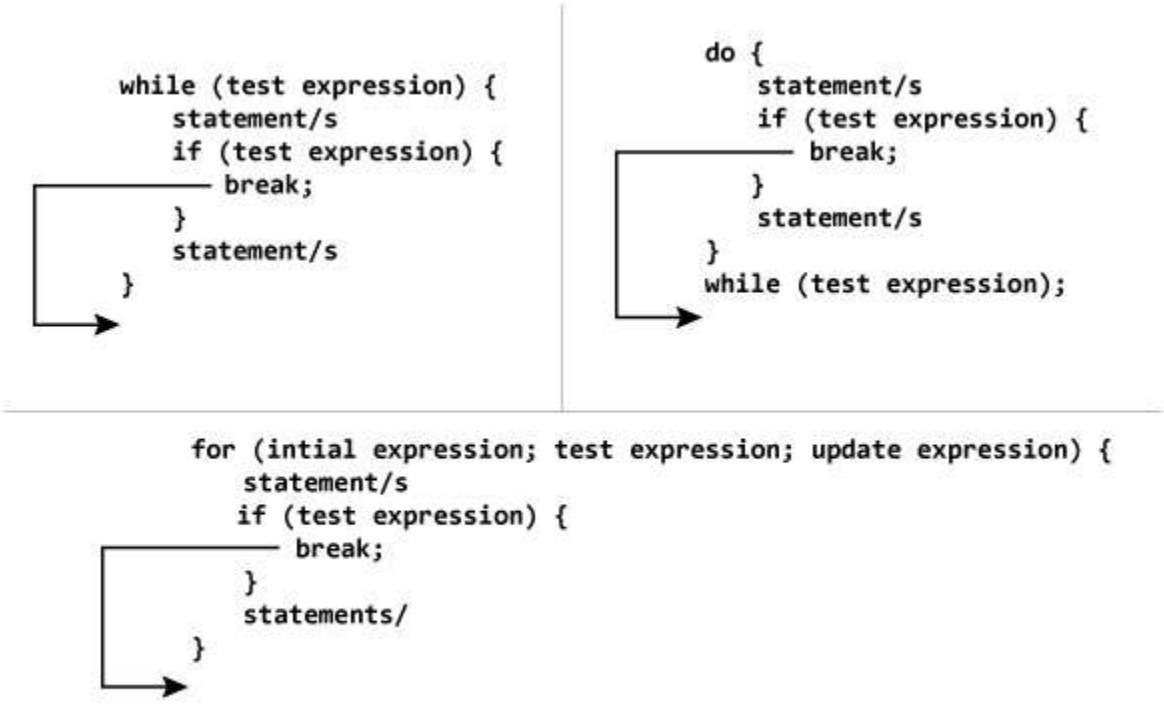
The break; statement terminates a loop (for, while and do..while loop) and a switch statement immediately when it appears.

Syntax of break

```
break;
```

In real practice, break statement is almost always used inside the body of conditional statement (if...else) inside the loop.

How break statement works?



NOTE: The break statment may also be used inside body of else statement.

Example 1: C++ break

C++ program to add all number entered by user until user enters 0.

```
// C++ Program to demonstrate working of break statement

#include <iostream>
using namespace std;
int main() {
   float number, sum = 0.0;

   // test expression is always true
   while (true)
   {
      cout << "Enter a number: ";
      cin >> number;

      if (number != 0.0)
      {
         sum += number;
      }
      else
```

```
        {
            // terminates the loop if number equals 0.0
            break;
        }

    }
    cout << "Sum = " << sum;

    return 0;
}
```

Output

```
Enter a number: 4
Enter a number: 3.4
Enter a number: 6.7
Enter a number: -4.5
Enter a number: 0
Sum = 9.6
```

In the above program, the test expression is always true.

The user is asked to enter a number which is stored in the variable *number*. If the user enters any number other than 0, the number is added to *sum* and stored to it.

Again, the user is asked to enter another number. When user enters 0, the test expression inside if statement is false and body of else is executed which terminates the loop.

Finally, the sum is displayed.

Example – Use of break statement in a while loop

In the example below, we have a while loop running from 10 to 200 but since we have a break statement that gets encountered when the loop counter variable value reaches 12, the loop gets terminated and the control jumps to the next statement in program after the loop body.

```
#include <iostream>
using namespace std;
int main(){
   int num =10;
   while(num<=200) {
      cout<<"Value of num is: "<<num<<endl;
      if (num==12) {
         break;
      }
      num++;
   }
   cout<<"Hey, I'm out of the loop";
   return 0;
}
```

Output:

Value of num is: 10
Value of num is: 11
Value of num is: 12
Hey, I'm out of the loop

5.C++ continue Statement

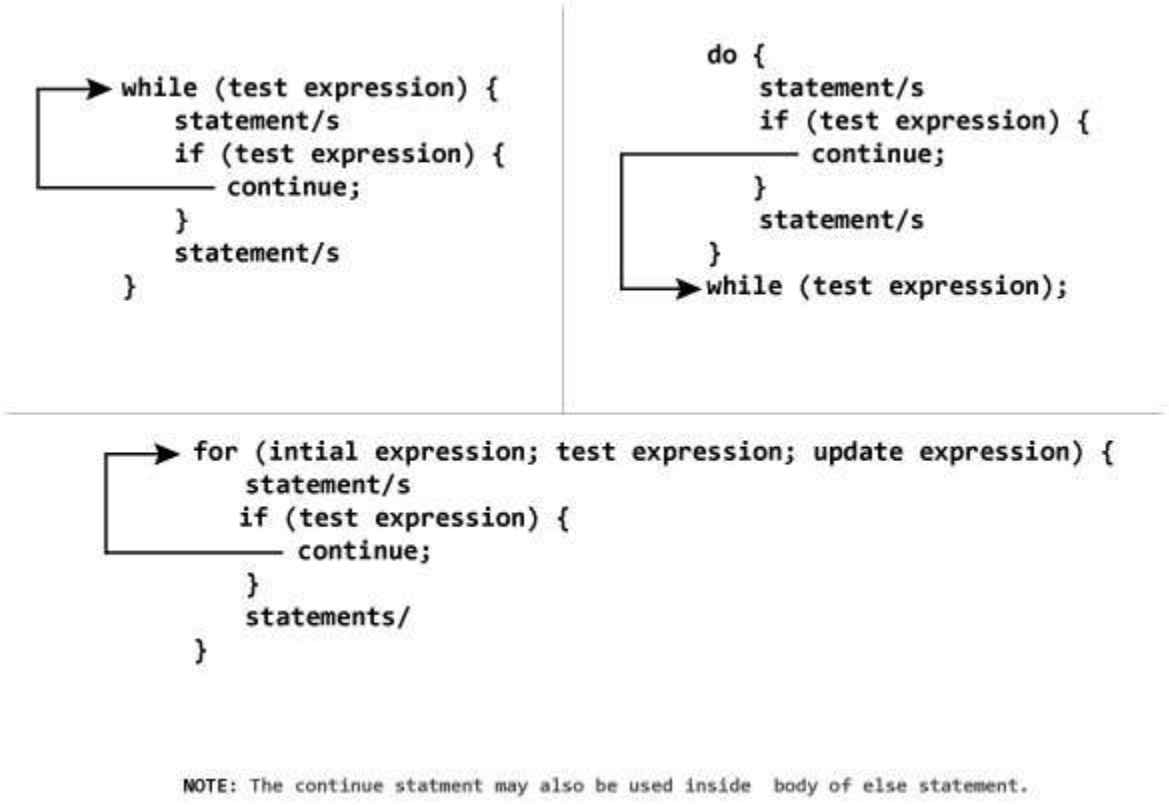
It is sometimes necessary to skip a certain test condition within a loop. In such case, continue; statement is used in C++ programming.

Syntax of continue

continue;

In practice, continue; statement is almost always used inside a conditional statement.

Working of continue Statement



NOTE: The continue statment may also be used inside body of else statement.

Example 2: C++ continue

C++ program to display integer from 1 to 10 except 6 and 9.

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

int main()
{
   for (int i = 1; i <= 10; ++i)
   {
      if ( i == 6 || i == 9)
      {
         continue;
      }
      cout << i << "\t";
   }
   return 0;
}
```

Output

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

In above program, when *i* is 6 or 9, execution of statement `cout << i << "\t";` is skipped inside the loop using `continue;` statement.

Example: continue statement inside for loop

As you can see that the output is missing the value 3, however the for loop iterate though the num value 0 to 6. This is because we have set a condition inside loop in such a way, that the continue statement is encountered when the num value is equal to 3. So for this iteration the loop skipped the cout statement and started the next iteration of loop.

```
#include <iostream>
using namespace std;
int main(){
  for (int num=0; num<=6; num++) {
    /* This means that when the value of
     * num is equal to 3 this continue statement
     * would be encountered, which would make the
     * control to jump to the beginning of loop for
     * next iteration, skipping the current iteration
     */

    if (num==3) {
       continue;
    }
    cout<<num<<" ";
  }
```

```
  return 0;
}
```

6.C++ if, if...else and Nested if...else(Control Statement)

C++ if Statement

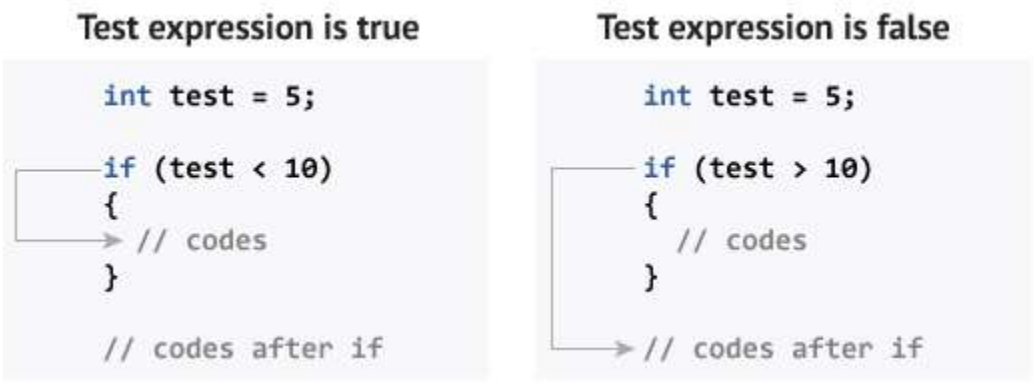
```
if (testExpression)
{
  // statements
}
```

The if statement evaluates the test expression inside parenthesis.

If test expression is evaluated to true, statements inside the body of if is executed.

If test expression is evaluated to false, statements inside the body of if is skipped.

How if statement works?



Flowchart of if Statement

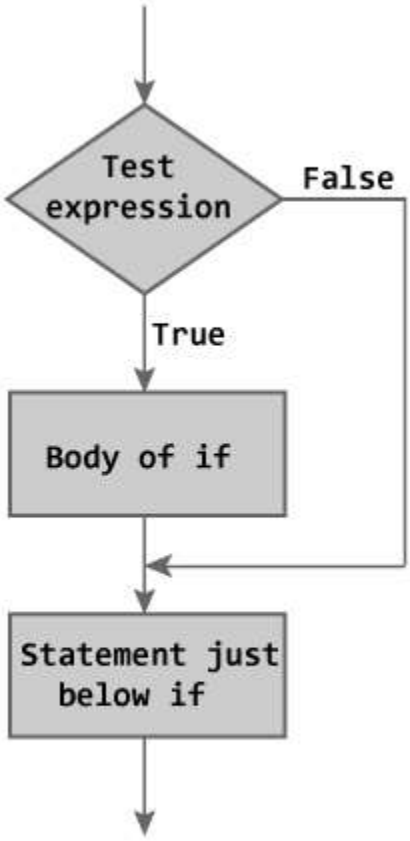


Figure: Flowchart of if Statement

Above figure describes the working of an if statement.

Example 1: C++ if Statement

```
// Program to print positive number entered by the user
// If user enters negative number, it is skipped

#include <iostream>
using namespace std;

int main()
{
   int number;
   cout << "Enter an integer: ";
   cin >> number;

   // checks if the number is positive
   if ( number > 0)
   {
      cout << "You entered a positive integer: " << number << endl;
```

```
    }

    cout << "This statement is always executed.";
    return 0;

}
```

Output 1

```
Enter an integer: 5
You entered a positive number: 5
This statement is always executed.
```

Output 2

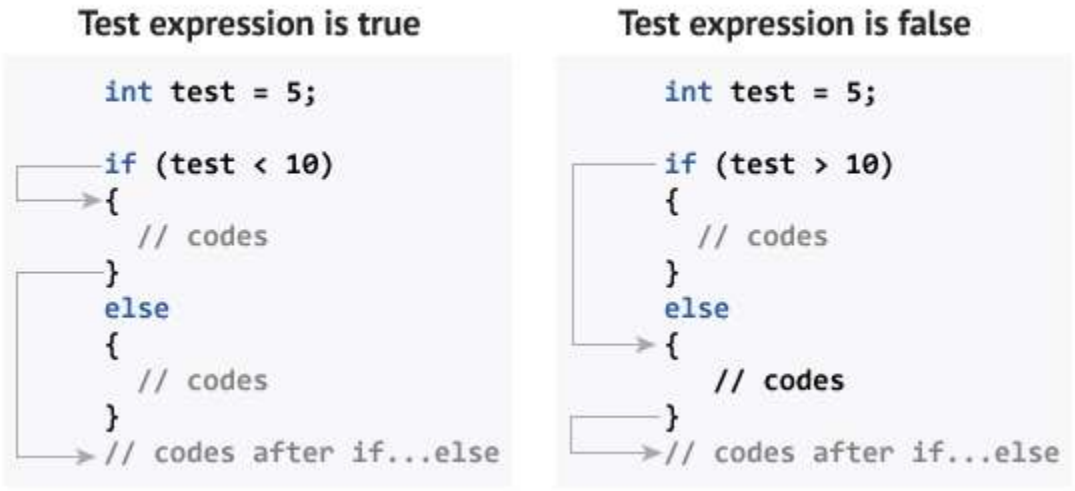
```
Enter a number: -5
This statement is always executed.
```

C++ if...else

The if else executes the codes inside the body of if statement if the test expression is true and skips the codes inside the body of else.

If the test expression is false, it executes the codes inside the body of else statement and skips the codes inside the body of if.

How if...else statement works?



Flowchart of if...else

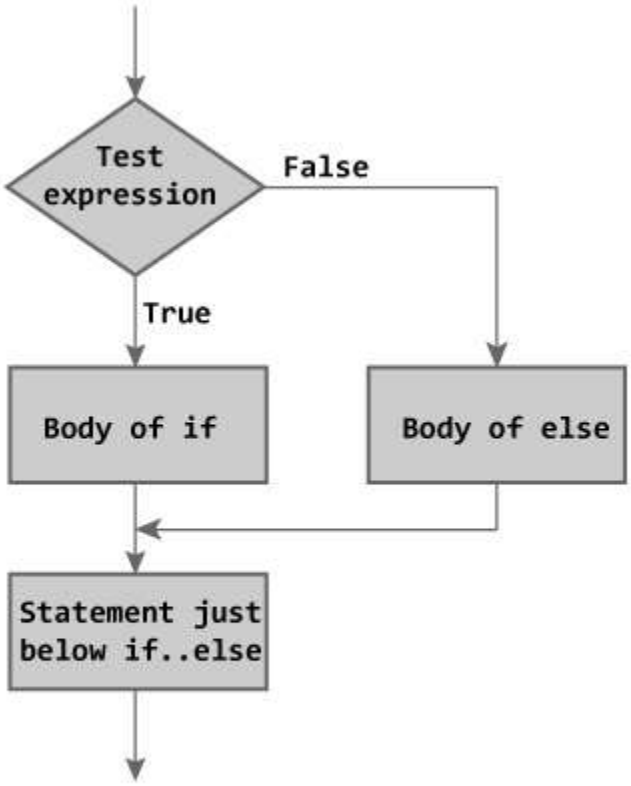


Figure: Flowchart of if...else Statement

Example 2: C++ if...else Statement

```
// Program to check whether an integer is positive or negative
// This program considers 0 as positive number

#include <iostream>
using namespace std;

int main()
{
   int number;
   cout << "Enter an integer: ";
   cin >> number;

   if ( number >= 0)
   {
      cout << "You entered a positive integer: " << number << endl;
   }

   else
```

```
    {
        cout << "You entered a negative integer: " << number << endl;
    }

    cout << "This line is always printed.";
    return 0;
}
```

Output

```
Enter an integer: -4
You entered a negative integer: -4.
This line is always printed.
```

C++ Nested if...else

The if...else statement executes two different codes depending upon whether the test expression is true or false. Sometimes, a choice has to be made from more than 2 possibilities.

The nested if...else statement allows you to check for multiple test expressions and execute different codes for more than two conditions.

Syntax of Nested if...else

```
if (testExpression1)
{
  // statements to be executed if testExpression1 is true
}
else if(testExpression2)
{
  // statements to be executed if testExpression1 is false and testExpression2 is true
}
else if (testExpression 3)
{
  // statements to be executed if testExpression1 and testExpression2 is false and testExpression3 is true
}
.
.
else
{
  // statements to be executed if all test expressions are false
}
```

Example 3: C++ Nested if...else

```
// Program to check whether an integer is positive, negative or zero

#include <iostream>
using namespace std;

int main()
```

```
{
   int number;
   cout << "Enter an integer: ";
   cin >> number;

   if ( number > 0)
   {
      cout << "You entered a positive integer: " << number << endl;
   }
   else if (number < 0)
   {
      cout<<"You entered a negative integer: " << number << endl;
   }
   else
   {
      cout << "You entered 0." << endl;
   }

   cout << "This line is always printed.";
   return 0;
}
```

Output

```
Enter an integer: 0
You entered 0.
This line is always printed.
```

Conditional/Ternary Operator ?:

A ternary operator operates on 3 operands which can be used instead of a if...else statement.

Consider this code:

```
if ( a < b ) {
   a = b;
}
else {
   a = -b;
}
```

You can replace the above code with:

```
a = (a < b) ? b : -b;
```

The ternary operator is more readable than a if...else statement for short conditions.

7.C++ switch..case Statement

The ladder if..else..if statement allows you to execute a block code among many alternatives. If you are checking on the value of a single variable in ladder if..else..if, it is better to use switch statement.

The switch statement is often faster than if...else (not always). Also, the syntax of switch statement is cleaner and easier to understand.

C++ switch...case syntax

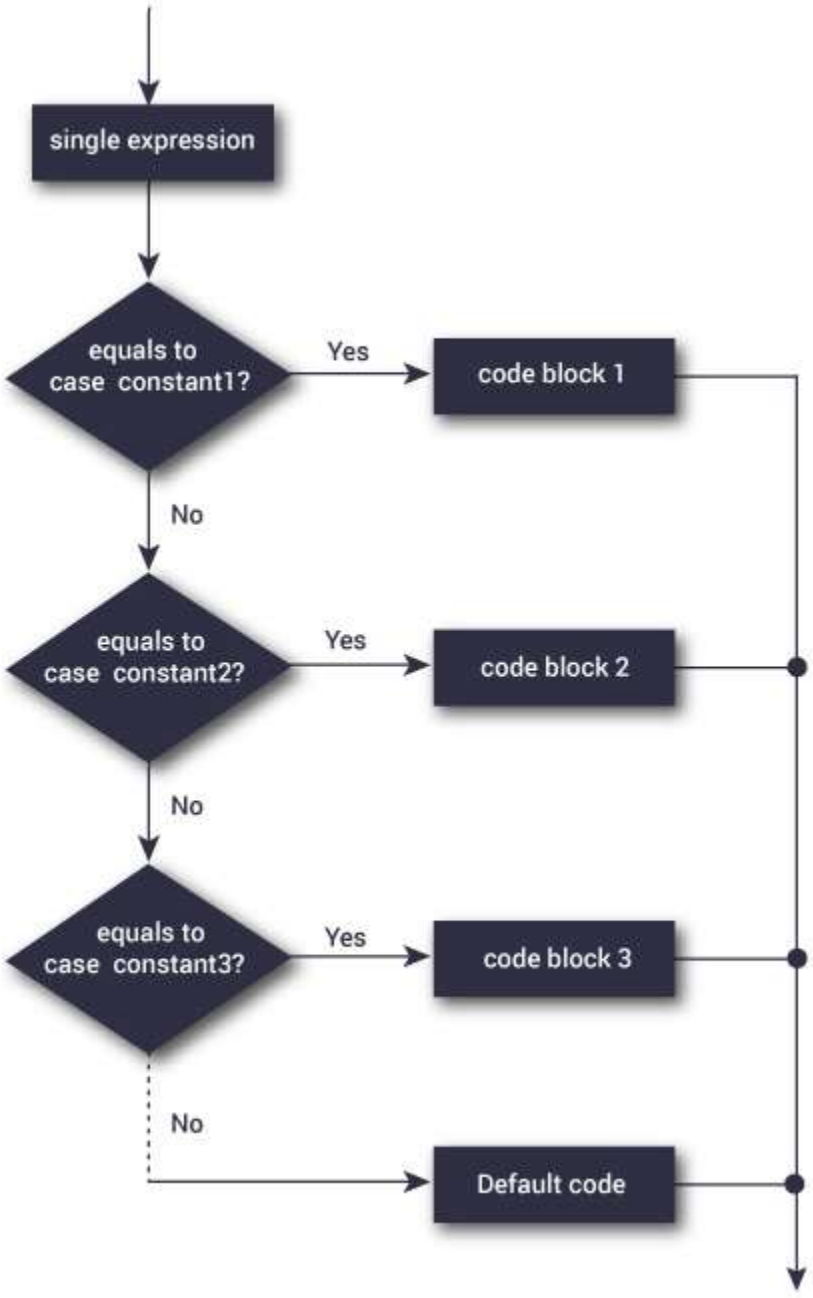
```
switch (n)
{
   case constant1:
      // code to be executed if n is equal to constant1;
      break;

   case constant2:
      // code to be executed if n is equal to constant2;
      break;
      .
      .
      .
   default:
      // code to be executed if n doesn't match any constant
}
```

When a case constant is found that matches the switch expression, control of the program passes to the block of code associated with that case.

In the above pseudocode, suppose the value of *n* is equal to *constant2*. The compiler will execute the block of code associated with the case statement until the end of switch block, or until the break statement is encountered.

The break statement is used to prevent the code running into the next case.

Flowchart of switch Statement



The above figure shows how a switch statement works and conditions are checked within the switch case clause.

Example: C++ switch Statement

```
// Program to built a simple calculator using switch Statement
```

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

int main()
{
   char o;
   float num1, num2;

   cout << "Enter an operator (+, -, *, /): ";
   cin >> o;

   cout << "Enter two operands: ";
   cin >> num1 >> num2;

   switch (o)
   {
      case '+':
         cout << num1 << " + " << num2 << " = " << num1+num2;
         break;
      case '-':
         cout << num1 << " - " << num2 << " = " << num1-num2;
         break;
      case '*':
         cout << num1 << " * " << num2 << " = " << num1*num2;
         break;
      case '/':
         cout << num1 << " / " << num2 << " = " << num1/num2;
         break;
      default:
         // operator is doesn't match any case constant (+, -, *, /)
         cout << "Error! operator is not correct";
         break;
   }

   return 0;
}
```

Output

```
Enter an operator (+, -, *, /): +
-
Enter two operands: 2.3
4.5
2.3 - 4.5 = -2.2
```

The - operator entered by the user is stored in *o* variable. And, two operands 2.3 and 4.5 are stored in variables *num1* and *num2* respectively.

Then, the control of the program jumps to

```
cout << num1 << " - " << num2 << " = " << num1-num2;
```

Finally, the break statement ends the switch statement.

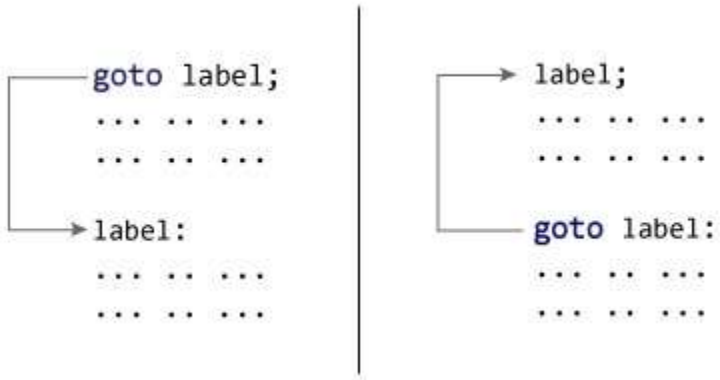
If break statement is not used, all cases after the correct case is executed.

8.C++ goto Statement

Syntax of goto Statement

```
goto label;
... .. ...
... .. ...
... .. ...
label:
statement;
... .. ...
```

In the syntax above, *label* is an identifier. When goto label; is encountered, the control of program jumps to label: and executes the code below it.



Example: goto Statement

```
// This program calculates the average of numbers entered by user.
// If user enters negative number, it ignores the number and
// calculates the average of number entered before it.

# include <iostream>
using namespace std;

int main()
{
   float num, average, sum = 0.0;
   int i, n;

   cout << "Maximum number of inputs: ";
   cin >> n;

   for(i = 1; i <= n; ++i)
   {
      cout << "Enter n" << i << ": ";
      cin >> num;

      if(num < 0.0)
```

```
        {
          // Control of the program move to jump:
           goto jump;
        }
        sum += num;
    }

jump:
    average = sum / (i - 1);
    cout << "\nAverage = " << average;
    return 0;
}
```

Output

```
Maximum number of inputs: 10
Enter n1: 2.3
Enter n2: 5.6
Enter n3: -5.6

Average = 3.95
```

You can write any C++ program without the use of goto statement and is generally considered a good idea not to use them.

Reason to Avoid goto Statement

The goto statement gives power to jump to any part of program but, makes the logic of the program complex and tangled.

In modern programming, goto statement is considered a harmful construct and a bad programming practice.

The goto statement can be replaced in most of C++ program with the use of break and continue statements.

Chap 07. Function in C++

1. Introduction
2. Types of function
3. Function Implementation
    - 3.1 Defining a Function
    - 3.2 Function Declarations
    - 3.3 Calling a Function
4. Function Arguments
5. Predefined Functions
    - 5.1 Rules of default arguments
    - 5.2 Common mistakes when using Default argument
6. C++ Recursion with example
7. C++ Function Overloading

1.Introduction

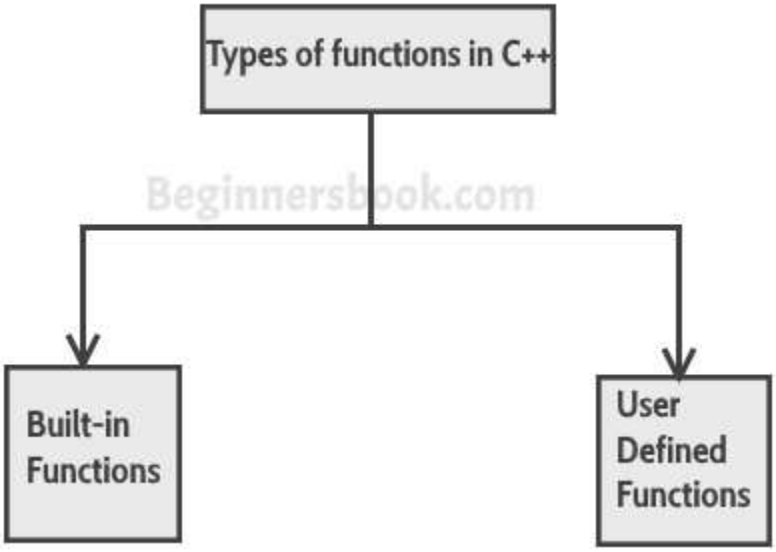
A function is block of code which is used to perform a particular task, for example let's say you are writing a large C++ program and in that program you want to do a particular task several number of times, like displaying value from 1 to 10, in order to do that you have to write few lines of code and you need to repeat these lines every time you display values. Another way of doing this is that you write these lines inside a function and call that function every time you want to display values. This would make you code simple, readable and reusable.

Why use Functions?

- Divide and Conquer
    - Allow complicated programs to be divided into manageable components
    - Programmer can focus on just the function: develop it, debug it, and test it
    - Various developers can work on different functions simultaneously
- Reusability:
    - Can be used in more than one place in a program--or in different programs
    - Avoids repetition of code, thus simplifying code maintenance
    - Can be called multiple times from anywhere in the program
- Components:
    - Custom functions and classes that you write

2. Types of function

We have two types of function in C++:



1) Built-in functions
2) User-defined functions

1) Build-it functions

Built-in functions are also known as library functions. We need not to declare and define these functions as they are already written in the C++ libraries such as iostream, cmath etc. We can directly call them when we need.

Example: C++ built-in function example

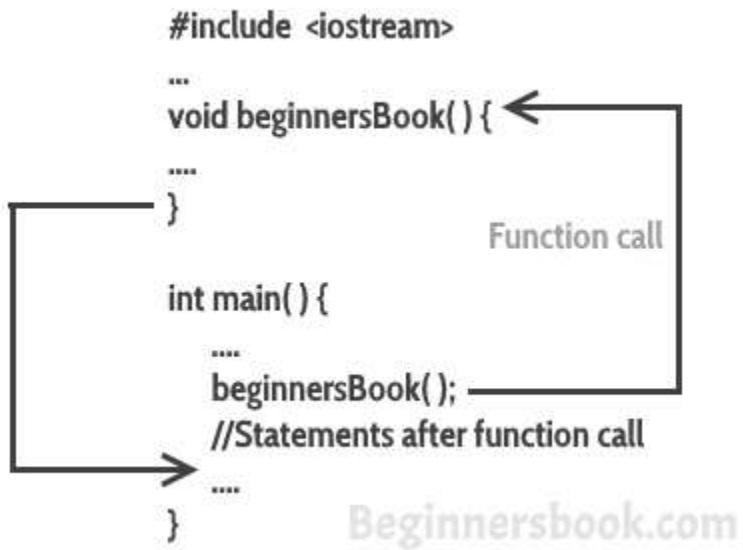
Here we are using built-in function pow(x,y) which is x to the power y. This function is declared in cmath header file so we have included the file in our program using #include directive.

```
#include <iostream>
#include <cmath>
using namespace std;
int main(){
   /* Calling the built-in function
    * pow(x, y) which is x to the power y
    * We are directly calling this function
    */
   cout<<pow(2,5);
   return 0;
}
```

Output:

32

2) User-defined functions



We have already seen user-defined functions, the example we have given at the beginning of this tutorial is an example of user-defined function. The functions that we declare and write in our programs are user-defined functions. Lets see another example of user-defined functions.

User-defined functions

```
#include <iostream>
#include <cmath>
using namespace std;
//Declaring the function sum
int sum(int,int);
```

```
int main(){
   int x, y;
   cout<<"enter first number: ";
   cin>> x;

   cout<<"enter second number: ";
   cin>>y;

   cout<<"Sum of these two :"<<sum(x,y);
   return 0;
}
//Defining the function sum
int sum(int a, int b) {
   int c = a+b;
   return c;
}
```

Output:

```
enter first number: 22
enter second number: 19
Sum of these two :41
```

Types of User-defined Functions in C++

For better understanding of arguments and return in functions, user-defined functions can be categorised as:

- Function with no argument and no return value
- Function with no argument but return value
- Function with argument but no return value
- Function with argument and return value

Consider a situation in which you have to check prime number. This problem is solved below by making user-defined function in 4 different ways as mentioned above.

Example 1: No arguments passed and no return value

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

void prime();

int main()
{
    // No argument is passed to prime()
    prime();
    return 0;
```

```
}


// Return type of function is void because value is not returned.
void prime()
{

   int num, i, flag = 0;

   cout << "Enter a positive integer enter to check: ";
   cin >> num;

   for(i = 2; i <= num/2; ++i)
   {
      if(num % i == 0)
      {
         flag = 1;
         break;
      }
   }

   if (flag == 1)
   {
      cout << num << " is not a prime number.";
   }
   else
   {
      cout << num << " is a prime number.";
   }
}
```

In the above program, prime() is called from the main() with no arguments.

prime() takes the positive number from the user and checks whether the number is a prime number or not.

Since, return type of prime() is void, no value is returned from the function.

Example 2: No arguments passed but a return value

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

int prime();

int main()
{
   int num, i, flag = 0;

   // No argument is passed to prime()
   num = prime();
```

```
    for (i = 2; i <= num/2; ++i)
    {
        if (num%i == 0)
        {
            flag = 1;
            break;
        }
    }

    if (flag == 1)
    {
        cout<<num<<" is not a prime number.";
    }
    else
    {
        cout<<num<<" is a prime number.";
    }
    return 0;
}

// Return type of function is int
int prime()
{
    int n;

    printf("Enter a positive integer to check: ");
    cin >> n;

    return n;
}
```

In the above program, prime() function is called from the main() with no arguments.

prime() takes a positive integer from the user. Since, return type of the function is an int, it returns the inputted number from the user back to the calling main() function.

Then, whether the number is prime or not is checked in the main() itself and printed onto the screen.

Example 3: Arguments passed but no return value

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

void prime(int n);

int main()
{
    int num;
    cout << "Enter a positive integer to check: ";
    cin >> num;
```

```
    // Argument num is passed to the function prime()
    prime(num);
    return 0;
}

// There is no return value to calling function. Hence, return type of function is void. */
void prime(int n)
{
    int i, flag = 0;
    for (i = 2; i <= n/2; ++i)
    {
        if (n%i == 0)
        {
            flag = 1;
            break;
        }
    }

    if (flag == 1)
    {
        cout << n << " is not a prime number.";
    }
    else {
        cout << n << " is a prime number.";
    }
}
```

In the above program, positive number is first asked from the user which is stored in the variable *num*.

Then, *num* is passed to the prime() function where, whether the number is prime or not is checked and printed.

Since, the return type of prime() is a void, no value is returned from the function.

Example 4: Arguments passed and a return value.

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

int prime(int n);

int main()
{
    int num, flag = 0;
    cout << "Enter positive integer to check: ";
    cin >> num;

    // Argument num is passed to check() function
    flag = prime(num);
```

```
    if(flag == 1)
        cout << num << " is not a prime number.";
    else
        cout<< num << " is a prime number.";
    return 0;
}

/* This function returns integer value.  */
int prime(int n)
{
    int i;
    for(i = 2; i <= n/2; ++i)
    {
        if(n % i == 0)
            return 1;
    }

    return 0;
}
```

In the above program, a positive integer is asked from the user and stored in the variable num.

Then, num is passed to the function prime() where, whether the number is prime or not is checked.

Since, the return type of prime() is an int, 1 or 0 is returned to the main() calling function. If the number is a prime number, 1 is returned. If not, 0 is returned.

Back in the main() function, the returned 1 or 0 is stored in the variable *flag*, and the corresponding text is printed onto the screen.

Which method is better?

All four programs above gives the same output and all are technically correct program.

There is no hard and fast rule on which method should be chosen.

The particular method is chosen depending upon the situation and how you want to solve a problem.

3. Function Implementation

3.1Defining a Function

The general form of a C++ function definition is as follows −

```
return_type function_name( parameter list ) {
   body of the function
}
```

A C++ function definition consists of a function header and a function body. Here are all the parts of a function −

- **Return Type** − A function may return a value. The **return_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword **void**.
- **Function Name** − This is the actual name of the function. The function name and the parameter list together constitute the function signature.
- **Parameters** − A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
- **Function Body** − The function body contains a collection of statements that define what the function does.

Example

Following is the source code for a function called **max()**. This function takes two parameters num1 and num2 and return the biggest of both −

```
// function returning the max between two numbers

int max(int num1, int num2) {
   // local variable declaration
   int result;

   if (num1 > num2)
      result = num1;
   else
      result = num2;

   return result;
}
```

3.2Function Declarations

A function **declaration** tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.

A function declaration has the following parts −

```
return_type function_name( parameter list );
```

For the above defined function max(), following is the function declaration −

```
int max(int num1, int num2);
```

Parameter names are not important in function declaration only their type is required, so following is also valid declaration −

```
int max(int, int);
```

Function declaration is required when you define a function in one source file and you call that function in another file. In such case, you should declare the function at the top of the file calling the function.

3.3 Calling a Function

While creating a C++ function, you give a definition of what the function has to do. To use a function, you will have to call or invoke that function.

When a program calls a function, program control is transferred to the called function. A called function performs defined task and when it's return statement is executed or when its function-ending closing brace is reached, it returns program control back to the main program.

To call a function, you simply need to pass the required parameters along with function name, and if function returns a value, then you can store returned value. For example −

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

// function declaration
int max(int num1, int num2);

int main () {
  // local variable declaration:
  int a = 100;
  int b = 200;
  int ret;

  // calling a function to get max value.
  ret = max(a, b);
  cout << "Max value is : " << ret << endl;

  return 0;
}

// function returning the max between two numbers
int max(int num1, int num2) {
  // local variable declaration
  int result;

  if (num1 > num2)
```

```
    result = num1;
  else
    result = num2;

  return result;
}
```

I kept max() function along with main() function and compiled the source code. While running final executable, it would produce the following result −

```
Max value is : 200
```

4. Function Arguments

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the **formal parameters** of the function.

The formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

While calling a function, there are two ways that arguments can be passed to a function −

Sr.No	Call Type & Description
1	Call by Value This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
2	Call by Pointer This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.
3	Call by Reference This method copies the reference of an argument into the formal parameter. Inside the function, the reference is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

By default, C++ uses **call by value** to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function and above mentioned example while calling max() function used the same method.

5. Predefined Functions

Using predefined functions:

- C++ Standard Library contains many predefined functions to perform various operations
- Predefined functions are organized into separate libraries
- I/O functions are in iostream header
- Math functions are in cmath header
- Some predefined C++ mathematical functions:
    - pow(x,y)
    - sqrt(x)
    - floor(x)
- **Power Function - pow(x,y):**
    - Power function pow(x,y) has two parameters
    - pow(x,y) returns value of type double
    - pow(x,y) calculates x to the power of y: pow(2,3) = 8.0
    - x and y called parameters (or arguments) of function pow
- **Square Root Function - sqrt(x):**
    - Square root function sqrt(x) has only one parameter
    - sqrt(x) returns value of type double
    - sqrt(x) calculates non-negative square root of x, for x >= 0.0: sqrt(2.25) = 1.5
- **Floor Function - floor(x):**
    - Floor function floor(x) has only one parameter
    - floor(x) returns value of type double
    - floor(x) calculates largest whole number not greater than x: floor(48.79) = 48.0

Function	Header File	Purpose	Parameter(s) Type	Result
`abs(x)`	`<cstdlib>`	Returns the absolute value of its argument: `abs(-7) = 7`	`int`	`int`
`ceil(x)`	`<cmath>`	Returns the smallest whole number that is not less than `x`: `ceil(56.34) = 57.0`	`double`	`doubl`
`cos(x)`	`<cmath>`	Returns the cosine of angle `x`: `cos(0.0) = 1.0`	`double` (`radians`)	`doubl`
`exp(x)`	`<cmath>`	Returns e^x, where `e = 2.718`: `exp(1.0) = 2.71828`	`double`	`doubl`
`fabs(x)`	`<cmath>`	Returns the absolute value of its argument: `fabs(-5.67) = 5.67`	`double`	`doubl`

Function	Header File	Purpose	Parameter(s) Type	Result
`floor(x)`	`<cmath>`	Returns the largest whole number that is not greater than `x`:`floor(45.67) = 45.00`	`double`	`double`
`pow(x, y)`	`<cmath>`	Returns x^y; If `x` is negative, `y` must be a whole number: `pow(0.16, 0.5) = 0.4`	`double`	`double`
`tolower(x)`	`<cctype>`	Returns the lowercase value of `x` if `x` is uppercase; otherwise, returns `x`	`int`	`int`
`toupper(x)`	`<cctype>`	Returns the uppercase value of `x` if `x` is lowercase; otherwise, returns `x`	`int`	`int`

Example:

- In general, function arguments may be constants, variables or more complex expressions
- The pre-defined math function sqrt listed above takes one input value (of type double) and returns its square root. Sample calls:

```
double x = 9.0, y = 16.0, z;

z = sqrt(36.0); // sqrt returns 6.0 (gets stored in z)
z = sqrt(x);// sqrt returns 3.0 (gets stored in z)
z = sqrt(x + y);// sqrt returns 5.0 (gets stored in z)

cout << sqrt(100.0);// sqrt returns 10.0, which gets printed

cout << sqrt(49);     // because of automatic type conversion rules
//  we can send an int where a double is expected
//  this call returns 7.0

// in this last one, sqrt(625.0) returns 25.0, which gets sent as the
//  argument to the outer sqrt call.  This one returns 5.0, which gets
//  printed

cout << sqrt(sqrt(625.0));
```

5.Default Arguments in C++ Functions

The default arguments are used when you provide no arguments or only few arguments while calling a function. The default arguments are used during compilation of program. For example, lets say you have a user-defined function sum declared like this: int sum(int a=10, int b=20), now while calling this function you do not provide any arguments, simply called sum(); then in this case the result would be 30, compiler used the default values 10 and 20 declared in function signature. If you pass only one argument like this: sum(80) then the result would be 100, using the passed argument 80 as first value and 20 taken from the default argument.

Example: Default arguments in C++

```
#include <iostream>
using namespace std;
int sum(int a, int b=10, int c=20);

int main(){
  /* In this case a value is passed as
   * 1 and b and c values are taken from
   * default arguments.
   */
  cout<<sum(1)<<endl;

  /* In this case a value is passed as
```

```
    * 1 and b value as 2, value of c values is
    * taken from default arguments.
    */
   cout<<sum(1, 2)<<endl;

   /* In this case all the three values are
    * passed during function call, hence no
    * default arguments have been used.
    */
   cout<<sum(1, 2, 3)<<endl;
   return 0;
}
int sum(int a, int b, int c){
   int z;
   z = a+b+c;
   return z;
}
```

Output:

```
31
23
6
```

5.1Rules of default arguments

As you have seen in the above example that I have assigned the default values for only two arguments b and c during function declaration. It is up to you to assign default values to all arguments or only selected arguments but remember the following rule while assigning default values to only some of the arguments:

If you assign default value to an argument, the subsequent arguments must have default values assigned to them, else you will get compilation error.

For example: Lets see some valid and invalid cases.
Valid: Following function declarations are valid –

```
int sum(int a=10, int b=20, int c=30);
int sum(int a, int b=20, int c=30);
int sum(int a, int b, int c=30);
```

Invalid: Following function declarations are invalid –

```
/* Since a has default value assigned, all the
 * arguments after a (in this case b and c) must have
 * default values assigned
 */
int sum(int a=10, int b, int c=30);
```

```
/* Since b has default value assigned, all the
 * arguments after b (in this case c) must have
 * default values assigned
 */
int sum(int a, int b=20, int c);

/* Since a has default value assigned, all the
 * arguments after a (in this case b and c) must have
 * default values assigned, b has default value but
 * c doesn't have, thats why this is also invalid
 */
int sum(int a=10, int b=20, int c);
```

5.2Common mistakes when using Default argument

1. `void add(int a, int b = 3, int c, int d = 4);`

   The above function will not compile. You cannot miss a default argument in between two arguments.
   In this case, *c* should also be assigned a default value.

2. `void add(int a, int b = 3, int c, int d);`

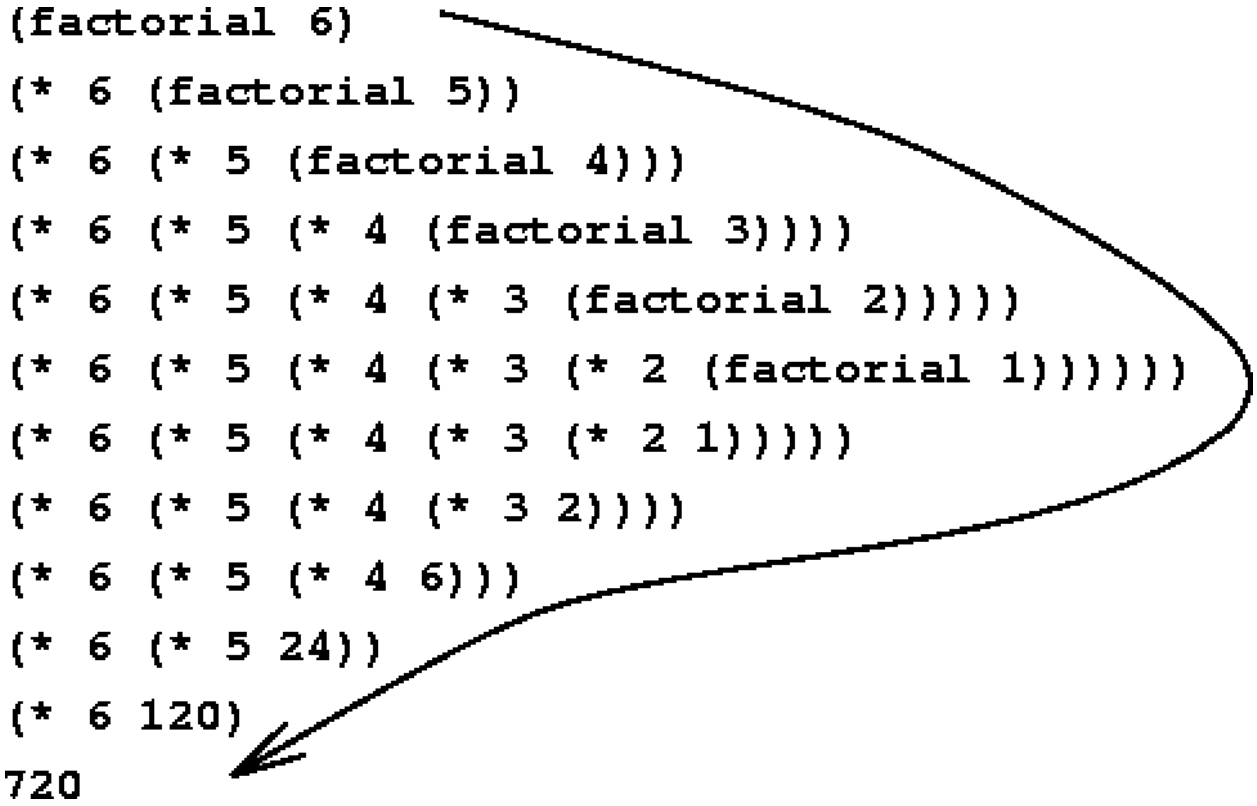
   The above function will not compile as well. You must provide default values for each argument after *b*.
   In this case, *c* and *d* should also be assigned default values.
   If you want a single default argument, make sure the argument is the last one. `void add(int a, int b, int c, int d = 4);`

3. No matter how you use default arguments, a function should always be written so that it serves only one purpose.
   If your function does more than one thing or the logic seems too complicated, you can use Function overloading to separate the logic better.

6. C++ Recursion with example

The process in which a function calls itself is known as recursion and the corresponding function is called the **recursive function**. The popular example to understand the recursion is factorial function.

Factorial function: f(n) = n*f(n-1), base condition: if n<=1 then f(n) = 1. Don’t worry we wil discuss what is base condition and why it is important.



C++ recursion example: Factorial

```
#include <iostream>
using namespace std;
//Factorial function
int f(int n){
   /* This is called the base condition, it is
    * very important to specify the base condition
    * in recursion, otherwise your program will throw
    * stack overflow error.
    */
   if (n <= 1)
        return 1;
   else
       return n*f(n-1);
}
int main(){
   int num;
   cout<<"Enter a number: ";
   cin>>num;
   cout<<"Factorial of entered number: "<<f(num);
   return 0;
}
```

Output:

```
Enter a number: 5
```

Factorial of entered number: 120

Base condition

In the above program, you can see that I have provided a base condition in the recursive function. The condition is:

```
if (n <= 1)
    return 1;
```

The purpose of recursion is to divide the problem into smaller problems till the base condition is reached. For example in the above factorial program I am solving the factorial function f(n) by calling a smaller factorial function f(n-1), this happens repeatedly until the n value reaches base condition(f(1)=1). If you do not define the base condition in the recursive function then you will get stack overflow error.

A function is a group of statements that together perform a task. Every C++ program has at least one function, which is **main()**, and all the most trivial programs can define additional functions.

You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division usually is such that each function performs a specific task.

A function **declaration** tells the compiler about a function's name, return type, and parameters. A function **definition** provides the actual body of the function.

The C++ standard library provides numerous built-in functions that your program can call. For example, function **strcat()** to concatenate two strings, function **memcpy()** to copy one memory location to another location and many more functions.

A function is known with various names like a method or a sub-routine or a procedure etc.

https://www.programiz.com/cpp-programming/function

http://www.cplusplus.com/doc/tutorial/functions/

Passing Arguments to Function

In programming, argument (parameter) refers to the data which is passed to a function (function definition) while calling it.

In the above example, two variables, *num1* and *num2* are passed to function during function call. These arguments are known as actual arguments.

The value of *num1* and *num2* are initialized to variables *a* and *b* respectively. These arguments *a* and *b* are called formal arguments.

This is demonstrated in figure below:

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

int add(int, int);

int main() {
     ... .. ...
     sum = add(num1, num2);   // Actual parameters: num1 and num2
    ... .. ...
}

int add(int a, int b) {      // Formal parameters: a and b
     ... .. ...
      add = a+b;
    ... .. ...
}
```

Notes on passing arguments

- The numbers of actual arguments and formals argument should be the same. (Exception: Function Overloading)
- The type of first actual argument should match the type of first formal argument. Similarly, type of second actual argument should match the type of second formal argument and so on.
- You may call function a without passing any argument. The number(s) of argument passed to a function depends on how programmer want to solve the problem.
- You may assign default values to the argument. These arguments are known as default arguments.
- In the above program, both arguments are of int type. But it's not necessary to have both arguments of same type.

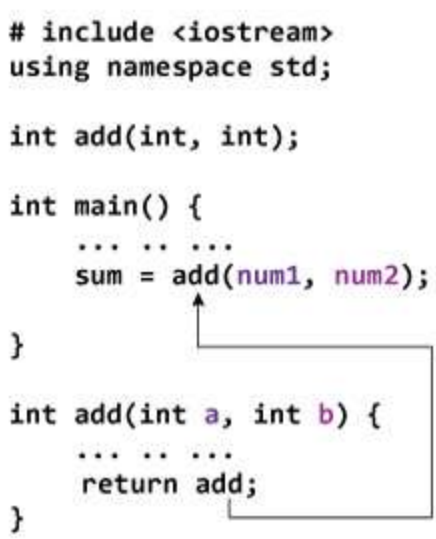
Return Statement

A function can return a single value to the calling program using return statement.

In the above program, the value of *add* is returned from user-defined function to the calling program using statement below:

return add;

The figure below demonstrates the working of return statement.



In the above program, the value of *add* inside user-defined function is returned to the calling function. The value is then stored to a *variable* sum.

Notice that the variable returned, i.e., *add* is of type int and *sum* is also of *int* type.

Also, notice that the return type of a function is defined in function declarator int add(int a, int b). The int before add(int a, int b) means the function should return a value of type int.

If no value is returned to the calling function then, void should be used.

https://www.programiz.com/cpp-programming/function

7.C++ Function Overloading

Function refers to a segment that groups code to perform a specific task.

In C++ programming, two functions can have same name if number and/or type of arguments passed are different.

These functions having different number or type (or both) of parameters are known as overloaded functions. For example:

```
int test() { }
int test(int a) { }
float test(double a) { }
int test(int a, double b) { }
```

Here, all 4 functions are overloaded functions because argument(s) passed to these functions are different.

Notice that, the return type of all these 4 functions are not same. Overloaded functions may or may not have different return type but it should have different argument(s).

```
// Error code
int test(int a) { }
double test(int b){ }
```

The number and type of arguments passed to these two functions are same even though the return type is different. Hence, the compiler will throw error.

Example 1: Function Overloading

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

void display(int);
void display(float);
void display(int, float);

int main() {

   int a = 5;
   float b = 5.5;

   display(a);
   display(b);
   display(a, b);

   return 0;
}

void display(int var) {
   cout << "Integer number: " << var << endl;
}

void display(float var) {
   cout << "Float number: " << var << endl;
}

void display(int var1, float var2) {
   cout << "Integer number: " << var1;
   cout << " and float number:" << var2;
}
```

Output

```
Integer number: 5
Float number: 5.5
Integer number: 5 and float number: 5.5
```

Here, the display() function is called three times with different type or number of arguments.

The return type of all these functions are same but it's not necessary.

Reference

- https://www.programiz.com/cpp-programming/function
- https://www.programiz.com/cpp-programming/function
- http://www.cplusplus.com/doc/tutorial/functions/

Chap 11. String in C++

1. Introduction
2. Concatenation of C-Style Strings
3. C-Style vs. C++ Style
4. The String Class in C++
5. Applications

1.Introduction

C++ provides following two types of string representations −

- The C-style character string.
- The string class type introduced with Standard C++.

What is a C-Style String?

In the last lesson we learned about C++ strings. There is an additional, different type of string—the C-style string.

C-style strings are the primary string type of the C language. Since C++ is a superset of C, it too has use C-style strings. In previous lessons we examined characters and arrays—a C-style string is essentially an array of characters with the special character null ('\0') at the end.

Since C-style strings are arrays, they have a fixed maximum length. "Dynamic" strings are possible, but they require the programmer to manually manage memory—a process that is full of pitfalls and gotchas. Many security problems and bugs refer to buffer overflows—which are typically errors with C-style strings that result in memory corruption.

There are many who believe that you should never have a C-style string in a C++ program. Unfortunately "should" and the "real world" are two quite different things. There is still a lot of legacy code out there that uses C-style strings. As a programmer in the real world, you will see this type of string and have to deal with it.

So let's start with an experimental C-style string program. Create a project named **c-string**, assign it to your **C++1_Lessons** working set, and create a program **c-string.cpp** containing the following:

Code to Type: c-string.cpp

```
#include <iostream>

int main()
{
   char name[] = {'S', 'a', 'm', '\0'};    // The name for this example

   std::cout << "The name is " << name << std::endl;
   return(0);
}
```

This assigns the value "Sam" to a C string name, which can be output to **std::cout**.

OBSERVE:
```
The name is Sam
```

Now what would happen if you didn't end the string with '\0'? Let's see. Change your program as shown:

Code to Edit: c-string.cpp
```
#include <iostream>
int main()
{
   char name[] = {'S', 'a', 'm', '\0'};    // The name for this example
   char other[] = {'J', 'o', 'e'};   // Another name with an error

   std::cout << "The name is " << name << " and other is " << other  << std::endl;
   return(0);
}
```
OBSERVE:
```
The name is Sam and other is Joew
```

Your actual output might differ from this example—in fact, it might even look normal. This is another situation where the behavior is undefined. If you do not put an end-of-string marker (\0) in your C-style string, it is anybody's guess what will occur.

Remember when we accidentally used an array element we weren't supposed to use? What happened? Because there was no end-of-string marker, C++ did not stop at the end of other. It continued to write out characters from random memory until it actually found an end-of-string character.

C++ allows for even more compact initialization, using double quotes ("). Let's use it to fix the error:

Code to Edit: c-string.cpp
```
#include <iostream>
int main()
{
   char name[] = {'S', 'a', 'm', '\0'};    // The name for this example
   char other[] = "Joe";   // Another name with an error

   std::cout << "The name is " << name << " and other is " << other  << std::endl;
   return(0);
}
```

This form of initialization creates an array *four* characters long and assigns it four character values, the fourth and last being the end-of-string character.

OBSERVE:
```
The name is Sam and other is Joe
```

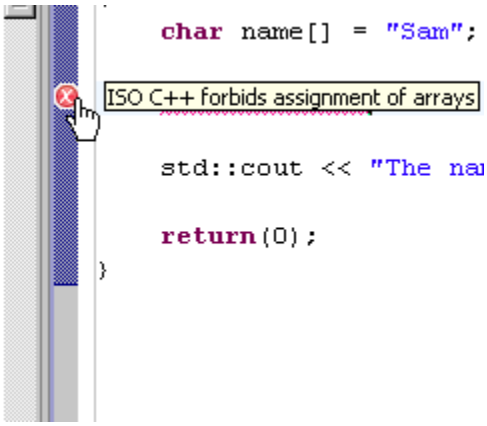
You cannot assign a value to an existing C-style string. This is because a C string is an array and you cannot change its value like you can with other variables. Try it:

Code to Edit: c-string.cpp

```
#include <iostream>
int main()
{
   char name[] = "Sam";    // The name for this example
   char other[] = "Joe";   // Another name with an error
   name = "Joe"; // Will this work?

   std::cout << "The name is " << name << " and other is " << other  << std::endl;

   return(0);
}
```

When you save this file you will see an error:

```
**** Build of configuration Debug for project
c-string ****

**** Internal Builder is used for build
****
g++ -O0 -g3 -Wall -c -fmessage-length=0
-oc-string.o ..\c-string.cpp
..\c-string.cpp: In function `int main()':
..\c-string.cpp:6: error: ISO C++ forbids
assignment of arrays
Build error occurred, build is stopped
Time consumed: 625  ms.
```



This error is generated because C strings require the programmer to worry about memory and do extra work when copying them. Think of the mailbox analogy from the array lesson—here we created a mailbox with three slots, containing the values "S," "a," and "m." Using **name = "Joe";** is like buying a new mailbox with three slots and trying to shove the new mailbox with slots "J", "o" and "e" where the old mailbox is still hanging.

Instead of trying to smash one mailbox in place of another, we must manually open each slot and copy the contents from the new to the old. This is done using the strncpy() function (see the reference information at cplusplus.com). Note this reference uses an old header file (string.h) instead of the current one (cstring).

This function has three parameters:

strncpy() Syntax

```
std::strncpy( destination , source , size );
```

destination is where the data is to be put, source is the source of the data, and size is the maximum number of characters to put in the destination.

Before we can use **strncpy**, we need to get out our tape measure—an operator named sizeof(). sizeof() returns the size of something in "char" units. If we use it on a C-style string, it will return the maximum number of characters that can be stored in the string.

This is very important because our strings (mailboxes) are fixed in size. If we try to copy too many characters from our new mailbox to the old, bad things could happen. Copying too many characters is yet another way to overflow the buffer and possibly cause the program to crash.

Let's see an example. Edit **c-string.cpp** as shown:

Code to Edit: c-string.cpp

```
#include <cstring>
#include <iostream>

int main()
{
   char name[4];// Short name

   std::cout << "Size is " << sizeof(name) << std::endl;
   std::strncpy(name, "Joe", sizeof(name));
   std::cout << "Name is now " << name << std::endl;

   return (0);
}
```

OBSERVE:

```
Size is 4
Name is now Joe
```

You might be wondering why you see **Size is 4**—after all, **Joe** is only three letters long! The fourth character is the end of string ('\0') null character. Remember: this character is required, so you must take it into account.

What happens when we try to copy a larger string into a smaller variable? Let's try it:

Code to Edit: c-string.cpp

```
#include <cstring>
#include <iostream>

int main()
{
   char name[4]; // Short name

   std::cout << "Size is " << sizeof(name) << std::endl;
   std::strncpy(name, "Joe", sizeof(name));
   std::cout << "Name is now " << name << std::endl;

   std::strncpy(name, "Steve", sizeof(name));
   std::cout << "Name is now " << name << std::endl;

   return (0);
}
```

OBSERVE:

```
Size is 4
Name is now Joe
Name is now Stev a"
```

The output from your program may be slightly different, see how "Steve" wasn't copied correctly, and the output contains extra characters?

We used the sizeof operator to compute the maximum number of characters that can be stored in the variable name. *This included the null character*. "Steve" is six characters—"Steve" plus the end-of-string character.

If the source string has fewer characters than the size of the array, then std::strncpy() copies the string and adds an end-of-string ('\0') to the end. But since the source is bigger here, it copies *size's* number of characters and does not append the end-of-string. So in order to make things work, we must do so ourselves:

Code to Edit: c-string.cpp

```
#include <cstring>
#include <iostream>

int main()
{
   char name[4]; // Short name

   std::cout << "Size is " << sizeof(name) << std::endl;
   std::strncpy(name, "Joe", sizeof(name));
   std::cout << "Name is now " << name << std::endl;

   std::strncpy(name, "Steve", sizeof(name));
   name[sizeof(name)-1] = '\0';
   std::cout << "Name is now " << name << std::endl;

   return (0);
}
```

Since sizeof(name) is 4 in our example, sizeof(name)-1 will be 3. Remember arrays (and C-style strings, which are arrays of characters) are zero-based, so 3 is the last mailbox in the name string.

OBSERVE:

```
Size is 4
Name is now Joe
Name is now Ste
```

We only see the first three characters of "Steve" because name is only big enough to contain four characters—"Ste" plus the null character. Believe it or not, changing the size of C-style strings is not as straightforward as you might expect. That topic will be covered in a future course.

2.Concatenation of C-Style Strings

In the last lesson, we covered the length() function for C++-style strings. The function std::strlen() returns the length of a C-style string. In other words, it returns the number of characters actually in the string, as opposed to sizeof(), which returns the *capacity*.

The function to perform concatenation of C-style strings is std::strncat(). Unlike concatenation of C++ strings, concatenation of C-style strings requires some planning to make sure you don't overflow any buffers. The function takes three parameters:

OBSERVE:
```
std::strncat( destination , source, size );
```

You must carefully calculate size in order to make sure you don't overflow the destination. The easiest way to do this is to always use the following code:

Concatenation Design Pattern
```
std::strncat( destination , source, sizeof(dest) - std::strlen(dest) - 1 );
destination[sizeof(destination)-1] = '\0';
```

The calculation for the size parameter for std::strncat() works like this:

Code	Description
sizeof(destination)	Start with the size of the destination string in characters.
- std::strlen(destination)	Subtract the number of characters already in the string.
- 1	Subtract one more for the end-of-string ('\0') character.

Let's try an example! Edit **c-string.cpp** as shown:

Code to Edit: c-string.cpp
```
#include <cstring>
#include <iostream>

int main()
{
   char name[25]; // Short name with plenty of space

   std::cout << "Size is " << sizeof(name) << std::endl;
   std::strncpy(name, "Joe", sizeof(name));
   std::cout << "Name is now " << name << std::endl;
```

```
    std::strncat(name, " Smith", sizeof(name) - std::strlen(name) - 1);
    name[sizeof(name)-1] = '\0';
    std::cout << "Name is now " << name << std::endl;

    return (0);
}
```

OBSERVE:

```
Size is 25
Name is now Joe
Name is now Joe Smith
```

Success!

Comparing Strings

The C-style string comparison function is std::strcmp(). It takes two parameters:

OBSERVE:

```
std::strcmp( string1, string2)
```

It returns:

- 0 if the strings are equal
- A positive value if the first character that does not match has a greater value in string1 than in string2
- A negative value if the first character that does not match has a greater value in string2 than in string1

Let's see how it works. Create a **compare-c** project and assign it to your **C++1_Lessons** working set. Then, create a program named **compare-c.cpp** as shown:

Code to Type: compare-c.cpp

```
#include <cstring>
#include <iostream>

int main()
{
    char str1[] = "Steve";
    char str2[] = "Steven";

    int result = std::strcmp(str1, str2);
    std::cout << "Result is " << result << std::endl;
    return (0);
}
```

OBSERVE:

```
Result is -1
```

You might be asking yourself why we can't use the == equality operator to check to see if two strings are the same. Let's try it to see how it might work:

Code to Edit: compare-c.cpp

```
#include <cstring>
#include <iostream>

int main()
{
   char str1[] = "Steve";
   char str2[] = "Steve";

   if (str1 == str2)
   {
      std::cout << "Same!" << std::endl;
   }
   else
   {
      std::cout << "Not the same!" << std::endl;
   }

   return (0);
}
```

OBSERVE:

```
Not the same!
```

Why is this? The two strings obviously have the same value.

Using our mailbox metaphor, the == equality operator checks to see if the two mailboxes are the same (like if they have the same serial number)—it doesn't check their contents. This fails because the mailboxes are not the same.

Tips

Converting C++ Strings to C-Style Strings

You can convert from C-style strings to C++ style strings, and vice versa. To get a C-style string from a C++-style string, use the c_str() function. You can assign a C-style string to a C++-style string. Try it:

Code to Edit: c-string.cpp

```
#include <iostream>
int main()
{
   char name[] = "Sam";    // The name for this example
   char other[] = "Joe";   // Another name
   char c_style[4];        // New variable to hold C++ string

   std::string cpp_style;

   cpp_style = name;  // Assigning to a C++-style string

   std::strncpy(c_style, cpp_style.c_str(), 4);   // Copy C-string to a variable

   std::cout << "The name is " << name << " and other is " << other  << std::endl;
   std::cout << "cpp_style as a C-style string is " << c_style << std::endl;
```

```
    return(0);
}
```

In this example we:

1. Copied a C-style string to a C++ style string, using direct assignment: cpp_style = name
2. Copied a C++ string to a C-style string, using c_str() and std::strncpy

OBSERVE:

```
The name is Sam and other is Joe
cpp_style as a C-style string is Sam
```

Unsafe String Functions

Many people don't use the string copy design pattern we provided, and thus buffer overflow problems occur in many programs. Instead, they use the function std::strcpy(). The standard form of this function is:

Unsafe Use of strcpy()

```
// Unsafe.  Do not use.
std::strcpy(destination, source);
```

Where destination is where the data will be copied into and source is the string to copy.

Paranoid programmers will ask themselves "What happens if the source is bigger that the destination?" The strcpy() function does not check length and if the source is too big, it will happily write random memory, corrupting your program.

Tip Paranoia, in programmers, is actually a *good* quality.

In the style guide, we recommend that you *not* use the strcpy() function.

In the real world, you might encounter legacy code containing strcpy()—a lot of code still uses this function. So, what should you do when you see it? Ideally, to make the program safe, replace all strcpy() functions with strncpy(), but any time you change a program there is risk—for example, you may not make the change correctly. If the code is working, even if it is messy, the best thing to do is to *leave it alone*. Unless there is a bug in the program that forces you to rewrite the code, *leave working code alone*.

There are two times you would want to upgrade strcpy() to the C-string strncpy() design pattern. The first is if you are changing the code anyway. (Always leave code better than when you found it.) The second is when you are trying to track down a memory corruption bug—in this case, the change might fix the bug.

Another unsafe function is std::strcat(). It performs much the same function as std::strcpy(), except that it does concatenation instead of copying.

std::strcat (unsafe)

```
// Unsafe.  Do not use.
std::strcat(destination, source);
```

This function adds the source string to the end of the destination *with no regard for the size of the destination*.

The future of strcpy() and strcat()

People have done all sorts of things to get around the limitations of strcpy() and strcat() for years. Currently, the OpenBSD folks have devised new functions, strlcpy() and strlcat(), designed to overcome the safety problems with strcpy() and strcat(), respectively. However, their effort has not made it into the standard yet.

Comparisons to other types

C Strings vs. Arrays of Characters

C-style strings and arrays of characters are two different things. An array of characters is of fixed length and contains no markers or other special characters. In other words, char name[50] contains 50 characters of any type, no more or less.

Note In a character array, the null character ('\0') need not be present; it's just another character, with no special meaning.

A C-style string is built on the character array type. It states that you have an array of characters, with an end-of-string marker ('\0') to end it. So, null ('\0') cannot be part of the string.

So all C-style strings are arrays of characters, but all arrays of characters are not C-style strings. Let's try an example. Edit your **compare-c.cpp** program as shown:

Code to Edit: compare-c.cpp

```
#include <cstring>
#include <iostream>

int main()
{
   char state[2] = {'C','A'};

   std::cout << "This probably looks funny: " << std::endl;
   std::cout << state << std::endl;

   std::cout << "This looks better: " << std::endl;
   std::cout << state[0] << state[1] << std::endl;

   return (0);
}
```

You might look at the state declaration and think it is an error, that it should be char state[3] in order to reserve one character for the end of string.

But this is not a mistake. state is *not* a C-style string; it's a character array. It holds two characters. No more, no less (there are no one-character state name abbreviations).

OBSERVE:
This probably looks funny:
CA ÿ"
This looks better:
CA

When you use **std::cout << state <<**, C++ will assumes that state is a C-style string (it's not). It will then look for the end-of-string marker (there is none) and write the state abbreviation followed by some random garbage.

You must write a character array one character at a time:

Writing a Character Array
```
std::cout << state[0] << state[1] << std::endl;
```

The bottom line: You can write out a C-style string using << but not a character array.

3.C-Style vs. C++ Style

There are advantages and disadvantages to using each type of string.

Size	C++ strings can store any length string automatically. You must explicitly declare the maximum size of C-style strings.
Memory	The memory used by C-style strings is precisely controlled. They do not grow or shrink depending on what data you put into them. C++ style strings manage their own memory. They can grow and shrink. They can also use memory in surprising ways if you are not careful.
Operations	Almost all of the operations you can use on C++ style strings are safe. Almost all of the operations you can use on C-style strings can be dangerous. You must be very careful about safety so you do not cause any buffer overflows.
Efficiency	C-style strings are more efficient that C++ style strings. However, as a practical matter, most programs are not CPU limited so efficiency makes little difference in a program. Safety does, and that's where C++ style strings win.
OS Interaction	If you are interacting directly with an operating system like Windows or Linux, you will find that many of the lower-level operating system functions (raw read and others) use C-style strings as arguments. This means that if you pass data from one part of the OS to another, C-style strings are more efficient.

Have you had enough of strings yet? In our next lesson we'll finally move on from output to input, and we'll start learning how to make decisions in our programs. See you there!

4.The String Class in C++

The standard C++ library provides a **string** class type that supports all the operations mentioned above, additionally much more functionality. Let us check the following example −

Live Demo

```
#include <iostream>
#include <string>

using namespace std;

int main () {

   string str1 = "Hello";
   string str2 = "World";
   string str3;
   int  len ;

   // copy str1 into str3
   str3 = str1;
   cout << "str3 : " << str3 << endl;

   // concatenates str1 and str2
   str3 = str1 + str2;
   cout << "str1 + str2 : " << str3 << endl;

   // total length of str3 after concatenation
   len = str3.size();
   cout << "str3.size() :  " << len << endl;

   return 0;
}
```

When the above code is compiled and executed, it produces result something as follows −

```
str3 : Hello
str1 + str2 : HelloWorld
str3.size() :  10
```

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 append, 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 characters 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 characters from a index by str
str6.replace(2, 7, "ese are test");

cout << str6 << endl;

return 0;
```

```
}
```

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.

5.Applications :

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

filter_none

edit

play_arrow

brightness_4

```
// 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;
}
```

Output :

```
Floating part is : 342
string contains only digit
```

google%20com%20in

Chap 12. File Handling

1. Introduction
2. Opening and Closing a File in C++
3. General functions used for File handling
4. Reading from and writing to a File
5. File Position Pointers
6. Binary files
7. Special operations in a File

1.Introduction

Many real-life scenarios are there that handle a large number of data, and in such situations, you need to use some secondary storage to store the data. The data are stored in the secondary device using the concept of files. Files are the collection of related data stored in a particular area on the disk. Programs can be written to perform read and write operations on these files.

In Standard C++, you can do I/O to and from disk files very much like the ordinary console I/O streams cin and cout. The object cin is a global object in the class istream (input stream), and the global object cout is a membe of the class ostream (output stream). File streams come in two flavors also: the class ifstream (input file stream) inherits from istream, and the class ofstream (output file stream) inherits from ostream. Thus all of the member functions and operators that you can apply to an istream or ostream object can also be applied to ifstream and ofstream objects. However, file streams have some additional member functions and internal information reflecting how they are connected to files on the disk.

Working with files generally requires the following kinds of data communication methodologies:

- Data transfer between console units
- Data transfer between the program and the disk file

So far we have learned about *iostream* standard library which provides cin and cout methods for reading from standard input and writing to standard output respectively. In this chapter, you will get to know how files are handled using C++ program and what are the functions and syntax used to handle files in C++.

Basics of using file streams

Let's get a quick overview, and then get into some details. First, you declare a file stream object for each file you need to simultaneously access. In this example, we will use one input file, and output file. But your program can have and be using as many files simultaneously as you wish. You just declare a stream object for each file:

```
#include <iostream>
#include <fstream>! // the class declarations for file stream objects
using namespace std;
```

```
...
int main ()
{
! ifstream my_input_file;! // an input file stream object
! ofstream my_output_file;! // an output file stream object
...
}
```

The above example code declares two objects, an input file stream object, and an output file stream object. Of course, they can be named whatever you wish, like any other C++ variable.

Here are the lists of standard file handling classes

1. **Ofstream**: This file handling class in C++ signifies the output file stream and is applied to create files for writing information to files
2. **Ifstream**: This file handling class in C++ signifies the input file stream and is applied for reading information from files
3. **Fstream**: This file handling class in C++ signifies the file stream generally, and has the capabilities for representing both ofstream and ifstream

All the above three classes are derived from fstreambase and from the corresponding iostream class and they are designed specifically to manage disk files.

2.Opening and Closing a File in C++

If programmers want to use a disk file for storing data, they need to decide about the following things about the file and its intended use. These points that are to be noted are:

- A name for the file
- Data type and structure of the file
- Purpose (reading, writing data)
- Opening method
- Closing the file (after use)

Files can be opened in two ways. They are:

1. Using constructor function of the class
2. Using member function open of the class

Opening a File

The first operation generally performed on an object of one of these classes to use a file is the procedure known as to opening a file. An open file is represented within a program by a stream and any input or output task performed on this stream will be applied to the physical file associated with it. The syntax of opening a file in C++ is:

```
open (filename, mode);
```

There are some mode flags used for file opening. These are:

- **ios::app**: append mode
- **ios::ate**: open a file in this mode for output and read/write controlling to the end of the file
- **ios::in**: open file in this mode for reading
- **ios::out**: open file in this mode for writing
- **ios::trunk**: when any file already exists, its contents will be truncated before file opening

Closing a file in C++

When any C++ program terminates, it automatically flushes out all the streams releases all the allocated memory and closes all the opened files. But it is good to use the close() function to close the file related streams and it is a member of ifsream, ofstream and fstream objects.

The structure of using this function is:

```
void close();
```

3.General functions used for File handling

1. **open()**: To create a file
2. **close()**: To close an existing file
3. **get()**: to read a single character from the file
4. **put()**: to write a single character in the file
5. **read()**: to read data from a file
6. **write()**: to write data into a file

4.Reading from and writing to a File

While doing C++ program, programmers write information to a file from the program using the stream insertion operator (<<) and reads information using the stream extraction operator (>>). The only difference is that for files programmers need to use an ofstream or fstream object instead of the cout object and ifstream or fstream object instead of the cin object.

Example:

```
#include <iostream>
#include <fstream.h>

void main() {
  ofstream file;
  file.open("egone.txt");
  file & lt; & lt;
  "Writing to a file in C++....";
  file.close();
}
```

Another Program for File Handling in C++

Example:

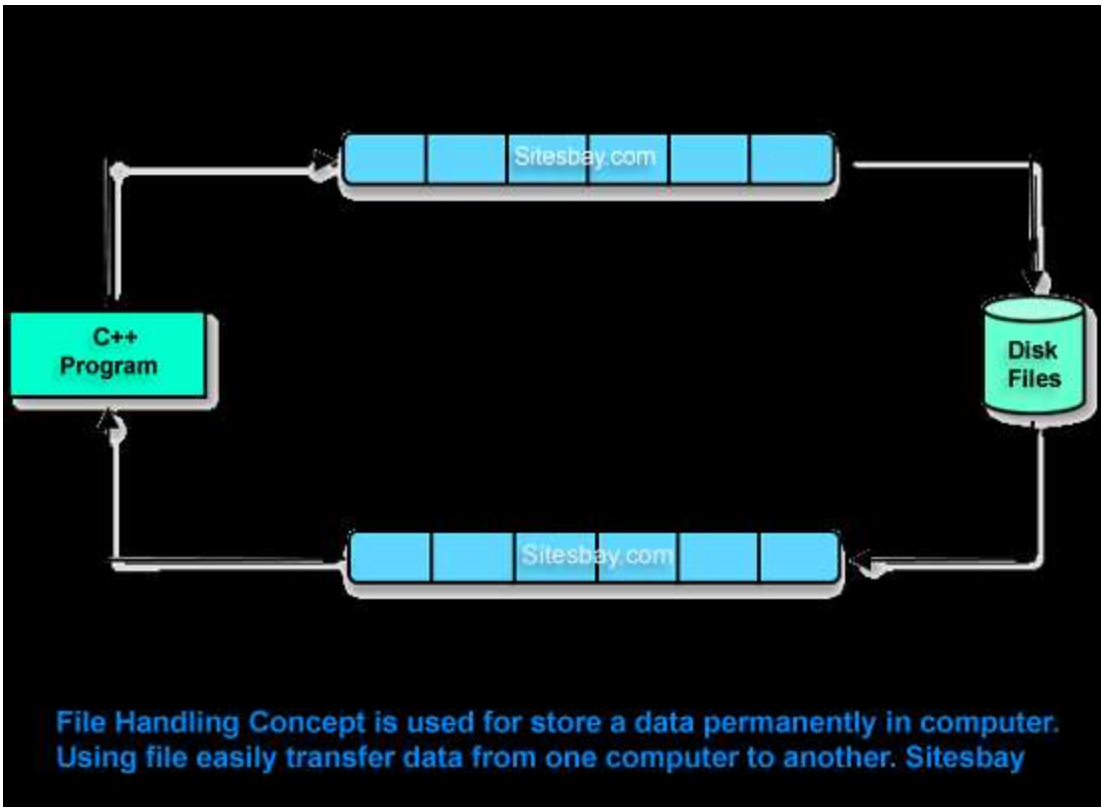
```
#include <iostream>
#include<fstream.h>
```

```
void main() {
  char c, fn[10];
  cout & lt; & lt;
  "Enter the file name....:";
  cin & gt; & gt;
  fn;
  ifstream in (fn);
  if (! in ) {
    cout & lt; & lt;
    "Error! File Does not Exist";
    getch();
    return;
  }
  cout & lt; & lt;
  endl & lt; & lt;
  endl;
  while ( in .eof() == 0) { in .get(c);
    cout & lt; & lt;
    c;
  }
}
```

Another C++ Program to Print Hello GS to the Console

Example:

```
#include <iostream>
#include<fstream.h>
#include<math.h>

void main() {
  ofstream fileo("Filethree");
  fileo & lt; & lt;
  "Hello GS";
  fileo.close();
  ifstream fin("Filethree");
  char ch;
  while (fin) {
    fin.get(ch);
    cout & lt; & lt;
    ch;
  }
  fin.close();
}
```



Writing to a File

While doing C++ programming, you write information to a file from your program using the stream insertion operator (<<) just as you use that operator to output information to the screen. The only difference is that you use an **ofstream** or **fstream** object instead of the **cout** object.

Reading from a File

You read information from a file into your program using the stream extraction operator (>>) just as you use that operator to input information from the keyboard. The only difference is that you use an **ifstream** or **fstream** object instead of the **cin** object.

Read and Write Example

Following is the C++ program which opens a file in reading and writing mode. After writing information entered by the user to a file named afile.dat, the program reads information from the file and outputs it onto the screen −

Live Demo

```
#include <fstream>
```

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

int main () {
   char data[100];

   // open a file in write mode.
   ofstream outfile;
   outfile.open("afile.dat");

   cout << "Writing to the file" << endl;
   cout << "Enter your name: ";
   cin.getline(data, 100);

   // write inputted data into the file.
   outfile << data << endl;

   cout << "Enter your age: ";
   cin >> data;
   cin.ignore();

   // again write inputted data into the file.
   outfile << data << endl;

   // close the opened file.
   outfile.close();

   // open a file in read mode.
   ifstream infile;
   infile.open("afile.dat");

   cout << "Reading from the file" << endl;
   infile >> data;

   // write the data at the screen.
   cout << data << endl;

   // again read the data from the file and display it.
   infile >> data;
   cout << data << endl;

   // close the opened file.
   infile.close();

   return 0;
}
```

When the above code is compiled and executed, it produces the following sample input and output −

```
$./a.out
Writing to the file
Enter your name: Zara
Enter your age: 9
Reading from the file
```

Zara
9

Above examples make use of additional functions from cin object, like getline() function to read the line from outside and ignore() function to ignore the extra characters left by previous read statement.

5.File Position Pointers

Both **istream** and **ostream** provide member functions for repositioning the file-position pointer. These member functions are **seekg** ("seek get") for istream and **seekp** ("seek put") for ostream.

The argument to seekg and seekp normally is a long integer. A second argument can be specified to indicate the seek direction. The seek direction can be **ios::beg** (the default) for positioning relative to the beginning of a stream, **ios::cur** for positioning relative to the current position in a stream or **ios::end** for positioning relative to the end of a stream.

The file-position pointer is an integer value that specifies the location in the file as a number of bytes from the file's starting location. Some examples of positioning the "get" file-position pointer are −

```
// position to the nth byte of fileObject (assumes ios::beg)
fileObject.seekg( n );

// position n bytes forward in fileObject
fileObject.seekg( n, ios::cur );

// position n bytes back from end of fileObject
fileObject.seekg( n, ios::end );

// position at end of fileObject
fileObject.seekg( 0, ios::end );
```

Text files

Text file streams are those where the ios::binary flag is not included in their opening mode. These files are designed to store text and thus all values that are input or output from/to them can suffer some formatting transformations, which do not necessarily correspond to their literal binary value.

Writing operations on text files are performed in the same way we operated with cout:

```
1  // writing on a text file
2  #include <iostream>
3  #include <fstream>
4  using namespace std;
5
6  int main () {
7    ofstream myfile ("example.txt");
8    if (myfile.is_open())
9    {
10     myfile << "This is a line.\n";
11     myfile << "This is another line.\n";
```

```
[file example.txt]
This is a line.
This is another line.
```

Edit & Run

```
12    myfile.close();
13   }
14   else cout << "Unable to open file";
15   return 0;
16 }
```

Reading from a file can also be performed in the same way that we did with cin:

```
1   // reading a text file
2   #include <iostream>
3   #include <fstream>
4   #include <string>
5   using namespace std;
6
7   int main () {
8     string line;
9     ifstream myfile ("example.txt");
10    if (myfile.is_open())
11    {                                   This is a line.          Edit & Run
12      while ( getline (myfile,line) )   This is another line.
13      {
14        cout << line << '\n';
15      }
16      myfile.close();
17    }
18
19    else cout << "Unable to open file";
20
21    return 0;
22  }
```

This last example reads a text file and prints out its content on the screen. We have created a while loop that reads the file line by line, using getline. The value returned by getline is a reference to the stream object itself, which when evaluated as a boolean expression (as in this while-loop) is true if the stream is ready for more operations, and false if either the end of the file has been reached or if some other error occurred.

Checking state flags

The following member functions exist to check for specific states of a stream (all of them return a bool value):

bad()

Returns true if a reading or writing operation fails. For example, in the case that we try to write to a file that is not open for writing or if the device where we try to write has no space left.

fail()

Returns true in the same cases as bad(), but also in the case that a format error happens, like when an alphabetical character is extracted when we are trying to read an integer number.

eof()

Returns true if a file open for reading has reached the end.

good()

It is the most generic state flag: it returns false in the same cases in which calling any of the previous functions would return true. Note that good and bad are not exact opposites (good checks more state flags at once).

The member function clear() can be used to reset the state flags.

get and put stream positioning

All i/o streams objects keep internally -at least- one internal position:

ifstream, like istream, keeps an internal *get position* with the location of the element to be read in the next input operation.

ofstream, like ostream, keeps an internal *put position* with the location where the next element has to be written.

Finally, fstream, keeps both, the *get* and the *put position*, like iostream.

These internal stream positions point to the locations within the stream where the next reading or writing operation is performed. These positions can be observed and modified using the following member functions:

tellg() and tellp()

These two member functions with no parameters return a value of the member type streampos, which is a type representing the current *get position* (in the case of tellg) or the *put position* (in the case of tellp).

seekg() and seekp()

These functions allow to change the location of the *get* and *put positions*. Both functions are overloaded with two different prototypes. The first form is:

```
seekg ( position );
seekp ( position );
```

Using this prototype, the stream pointer is changed to the absolute position position (counting from the beginning of the file). The type for this parameter is streampos, which is the same type as returned by functions tellg and tellp.

The other form for these functions is:

```
seekg ( offset, direction );
seekp ( offset, direction );
```

Using this prototype, the *get* or *put position* is set to an offset value relative to some specific point determined by the parameter direction. offset is of type streamoff. And direction is of type seekdir, which is an *enumerated type* that determines the point from where offset is counted from, and that can take any of the following values:

ios::beg offset counted from the beginning of the stream

ios::cur offset counted from the current position

ios::end offset counted from the end of the stream

The following example uses the member functions we have just seen to obtain the size of a file:

```
1  // obtaining file size
2  #include <iostream>
3  #include <fstream>
4  using namespace std;
5
6  int main () {
7    streampos begin,end;
8    ifstream myfile ("example.bin", ios::binary);
9    begin = myfile.tellg();
10   myfile.seekg (0, ios::end);
11   end = myfile.tellg();
12   myfile.close();
13   cout << "size is: " << (end-begin) << " bytes.\n";
14   return 0;
15 }
```

size is: 40 bytes. Edit & Run

Notice the type we have used for variables begin and end:

```
streampos size;
```

streampos is a specific type used for buffer and file positioning and is the type returned by file.tellg(). Values of this type can safely be subtracted from other values of the same type, and can also be converted to an integer type large enough to contain the size of the file.

These stream positioning functions use two particular types: streampos and streamoff. These types are also defined as member types of the stream class:

Type	Member type	Description

	Defined as fpos<mbstate_t>.
streampos ios::pos_type	It can be converted to/from streamoff and can be added or subtracted values of these types.
streamoff ios::off_type	It is an alias of one of the fundamental integral types (such as int or long long).

Each of the member types above is an alias of its non-member equivalent (they are the exact same type). It does not matter which one is used. The member types are more generic, because they are the same on all stream objects (even on streams using exotic types of characters), but the non-member types are widely used in existing code for historical reasons.

6.Binary files

For binary files, reading and writing data with the extraction and insertion operators (<< and >>) and functions like getline is not efficient, since we do not need to format any data and data is likely not formatted in lines.

File streams include two member functions specifically designed to read and write binary data sequentially: write and read. The first one (write) is a member function of ostream (inherited by ofstream). And read is a member function of istream (inherited by ifstream). Objects of class fstream have both. Their prototypes are:

```
write ( memory_block, size );
read ( memory_block, size );
```

Where memory_block is of type char* (pointer to char), and represents the address of an array of bytes where the read data elements are stored or from where the data elements to be written are taken. The size parameter is an integer value that specifies the number of characters to be read or written from/to the memory block.

```
1  // reading an entire binary file
2  #include <iostream>
3  #include <fstream>
4  using namespace std;
5
6  int main () {
7    streampos size;
8    char * memblock;
9                                                    the entire file content is in memory Edit & Run
10   ifstream file ("example.bin", ios::in|ios::binary|ios::ate);
11   if (file.is_open())
12   {
13     size = file.tellg();
14     memblock = new char [size];
15     file.seekg (0, ios::beg);
16     file.read (memblock, size);
17     file.close();
```

```
18
19    cout << "the entire file content is in memory";
20
21    delete[] memblock;
22  }
23  else cout << "Unable to open file";
24  return 0;
25 }
```

In this example, the entire file is read and stored in a memory block. Let's examine how this is done:

First, the file is open with the ios::ate flag, which means that the get pointer will be positioned at the end of the file. This way, when we call to member tellg(), we will directly obtain the size of the file.

Once we have obtained the size of the file, we request the allocation of a memory block large enough to hold the entire file:

```
memblock = new char[size];
```

Right after that, we proceed to set the *get position* at the beginning of the file (remember that we opened the file with this pointer at the end), then we read the entire file, and finally close it:

```
1 file.seekg (0, ios::beg);
2 file.read (memblock, size);
3 file.close();
```

At this point we could operate with the data obtained from the file. But our program simply announces that the content of the file is in memory and then finishes.

Buffers and Synchronization

When we operate with file streams, these are associated to an internal buffer object of type streambuf. This buffer object may represent a memory block that acts as an intermediary between the stream and the physical file. For example, with an ofstream, each time the member function put (which writes a single character) is called, the character may be inserted in this intermediate buffer instead of being written directly to the physical file with which the stream is associated.

The operating system may also define other layers of buffering for reading and writing to files.

When the buffer is flushed, all the data contained in it is written to the physical medium (if it is an output stream). This process is called *synchronization* and takes place under any of the following circumstances:

- **When the file is closed:** before closing a file, all buffers that have not yet been flushed are synchronized and all pending data is written or read to the physical medium.

- **When the buffer is full:** Buffers have a certain size. When the buffer is full it is automatically synchronized.
- **Explicitly, with manipulators:** When certain manipulators are used on streams, an explicit synchronization takes place. These manipulators are: flush and endl.
- **Explicitly, with member function sync():** Calling the stream's member function sync() causes an immediate synchronization. This function returns an int value equal to -1 if the stream has no associated buffer or in case of failure. Otherwise (if the stream buffer was successfully synchronized) it returns 0.

7.Special operations in a File

There are few important functions to be used with file streams like:

- tellp() - It tells the current position of the put pointer.

  Syntax: filepointer.tellp()

- tellg() - It tells the current position of the get pointer.

  Syntax: filepointer.tellg()

- seekp() - It moves the put pointer to mentioned location.

  Syntax: filepointer.seekp(no of bytes,reference mode)

- seekg() - It moves get pointer(input) to a specified location.

  Syntax: filepointer.seekg((no of bytes,reference point)

- put() - It writes a single character to file.
- get() - It reads a single character from file.

Note: *For seekp and seekg three reference points are passed:*
ios::beg *- beginning of the file*
ios::cur *- current position in the file*
ios::end *- end of the file*

Below is a program to show importance of tellp, tellg, seekp and seekg:

```
#include <iostream>
#include<conio>
#include <fstream>

using namespace std;

int main()
{
   fstream st; // Creating object of fstream class
```

```
   st.open("E:\studytonight.txt",ios::out);  // Creating new file
   if(!st) // Checking whether file exist
   {
      cout<<"File creation failed";
   }
   else
   {
      cout<<"New file created"<<endl;
      st<<"Hello Friends"; //Writing to file

      // Checking the file pointer position
      cout<<"File Pointer Position is "<<st.tellp()<<endl;

      st.seekp(-1, ios::cur); // Go one position back from current position

      //Checking the file pointer position
      cout<<"As per tellp File Pointer Position is "<<st.tellp()<<endl;

      st.close(); // closing file
   }
   st.open("E:\studytonight.txt",ios::in);   // Opening file in read mode
   if(!st) //Checking whether file exist
   {
      cout<<"No such file";
   }
   else
   {
      char ch;
      st.seekg(5, ios::beg);  // Go to position 5 from begning.
      cout<<"As per tellg File Pointer Position is "<<st.tellg()<<endl; //Checking file pointer position
      cout<<endl;
      st.seekg(1, ios::cur); //Go to position 1 from beginning.
      cout<<"As per tellg File Pointer Position is "<<st.tellg()<<endl; //Checking file pointer position
      st.close(); //Closing file
   }
   getch();
   return 0;
}
```

New file created File Pointer Position is 13 As per tellp File Pointer Position is 12 As per tellg File Pointer Position is 5 As per tellg File Pointer Position is 6

UNIT-15[libraries]

Index

15.1 C++ library

The C++ library supplied by IBM and this manual is based on the Dinkum C++ Library and the Dinkum C++ Library Reference. Use of this Dinkum C++ Library Reference is subject to limitations.. A C++ program can call on a large number of functions from the Dinkum C++ Library, a conforming implementation of the Standard C++ library. These functions perform essential services such as input and output. They also provide efficient implementations of frequently used operations. Numerous function and class definitions accompany these functions to help you to make better use of the library. Most of the information about the Standard C++ library can be found in the descriptions of the C++ library headers that declare or define library entities for the program. The Standard C++ library consists of 53 headers. Of these 53 headers, 13 constitute the Standard Template Library, or STL.

Other information on the Standard C++ library includes:

C++ Library Overview — how to use the Standard C++ library

Characters — how to write character constants and string literals , and how to convert between multibyte characters and wide characters .

Files and Streams — how to read and write data between the program and files .

Formatted Output — how to generate text under control of a format string.

Formatted Input — how to scan and parse text under control of a format string.

STL Conventions — how to read the descriptions of STL.

Template classes and functions Containers — how to use an arbitrary STL container template class.

The C++ library headers have two broader subdivisions, iostreams headers and STL headers.

Using C++ Library Headers You include the contents of a standard header by naming it in an include directive, as in:

#include /* include I/O facilities */

You can include the standard headers in any order, a standard header more than once, or two or more standard headers that define the same macro or the same type. Do not include a standard header within a declaration. Do not define macros that have the same names as keywords before you include a standard header. A C++ library header includes any other C++ library headers it needs to define needed types. (Always include explicitly any C++ library headers needed in a translation unit, however, lest you guess wrong about its actual dependencies.) A Standard C

header never includes another standard header. A standard header declares or defines only the entities described for it in this document.

Every function in the library is declared in a standard header. Unlike in Standard C, the standard header never provides a masking macro, with the same name as the function, that masks the function declaration and achieves the same effect. All names other than operator delete and operator new in the C++ library headers are defined in the std namespace, or in a namespace nested within the std namespace. Including a C++ library header does not introduce any library names into the current namespace.

You refer to the name cin ,

for example, as std::cin.

Alternatively, you can write the declaration: using namespace std; which promotes all library names into the current namespace. If you write this declaration immediately after all include directives, you can otherwise ignore namespace considerations in the remainder of the translation unit. Note that macro names are not subject to the rules for nesting namespaces.

Iostreams Conventions

The iostreams headers support conversions between text and encoded forms, and input and output to external files. The simplest use of iostreams requires only that you include the header . You can then extract values from cin , to read the standard input. The rules for doing so are outlined in the description of the class basic_istream . You can also insert values to cout , to write the standard output. The rules for doing so are outlined in the description of the class basic_ostream. Format control common to both extractors and insertors is managed by the class basic_ios . Manipulating this format information in the guise of extracting and inserting objects is the province of several manipulators . You can perform the same iostreams operations on files that you open by name, using the classes declared in . To convert between iostreams and objects of class basic_string, use the classes declared in . And to do the same with C strings, use the classes declared in . The remaining headers provide support services, typically of direct interest to only the most advanced users of the iostreams classes.

15.2 Contents of the standard C headers:

<assert.h>	Conditionally compiled macro that compares its argument to zero
<complex.h>	Complex number arithmetic
<ctype.h>	Functions to determine the type contained in character data

<errno.h>	Macros reporting error conditions
<fenv.h>	Floating-point environment
<float.h>	Limits of float types
<inttypes.h>	Format conversion of integer types
<iso646.h>	Alternative operator spellings
<limits.h>	Sizes of basic types
<locale.h>	Localization utilities
<math.h>	Common mathematics functions
<setjmp.h>	Nonlocal jumps
<signal.h>	Signal handling
<stdalign.h>	alignas and alignof convenience macros
<stdarg.h>	Variable arguments
<stdatomic.h>	Atomic types
<stdbool.h>	Boolean type
<stddef.h>	Common macro definitions
<stdint.h>	Fixed-width integer types
<stdio.h>	Input/output
<stdlib.h>	General utilities: memory management, program utilities, string conversions, random numbers
<stdnoreturn.h>	noreturn convenience macros

<string.h>	String handling
<tgmath.h>	Type-generic math (macros wrapping math.h and complex.h)
<threads.h>	Thread library
<time.h>	Time/date utilities
<uchar.h> (since C11)	UTF-16 and UTF-32 character utilities

15.3 String Streams

C++ provides a <sstream> header, which uses the same public interface to support IO between a program and string object (buffer).
The string streams is based on ostringstream (subclass of ostream), istringstream (subclass of istream) and bi-directional stringstream (subclass of iostream).

```
typedef basic_istringstream<char> istringstream;
typedef basic_ostringstream<char> ostringstream;
```

Stream input can be used to validate input data; stream output can be used to format the output.

ostringstream

```
explicit ostringstream (ios::openmode mode = ios::out);  // default with empty string
explicit ostringstream (const string & buf,
                        ios::openmode mode = ios::out);  // with initial str

string str () const;          // Get contents
void str (const string & str); // Set contents
```

For example,

```
// construct output string stream (buffer) - need <sstream> header
ostringstream sout;

// Write into string buffer
sout << "apple" << endl;
sout << "orange" << endl;
sout << "banana" << endl;

// Get contents
cout << sout.str() << endl;
```

The ostringstream is responsible for dynamic memory allocation and management.

istringstream

```
explicit istringstream (ios::openmode mode = ios::in);  // default with empty string
explicit istringstream (const string & buf,
                        ios::openmode mode = ios::in);  // with initial string
```

For example,

```
// construct input string stream (buffer) - need <sstream> header
istringstream sin("123 12.34 hello");

// Read from buffer
int i;
double d;
string s;
sin >> i >> d >> s;
cout << i << "," << d << "," << s << endl;
```

15.4 File Processing: File Input/Output (Header <fstream>)

C++ handles file IO similar to standard IO. In header <fstream>, the class ofstream is a subclass of ostream; ifstream is a subclass of istream; and fstream is a subclass of iostream for bi-directional IO. You need to include both <iostream> and <fstream> headers in your program for file IO.

To write to a file, you construct a ofsteam object connecting to the output file, and use the ostream functions such as stream insertion <<, put() and write(). Similarly, to read from an input file, construct an ifstream object connecting to the input file, and use the istream functions such as stream extraction >>, get(), getline() and read().

File IO requires an additional step to connect the file to the stream (i.e., file open) and disconnect from the stream (i.e., file close).

File Output

The steps are:

1. Construct an ostream object.
2. Connect it to a file (i.e., file open) and set the mode of file operation (e.g, truncate, append).
3. Perform output operation via insertion >> operator or write(), put() functions.
4. Disconnect (close the file which flushes the output buffer) and free the ostream object.

```
#include <fstream>

.....

ofstream fout;
```

```
fout.open(filename, mode);
.....
fout.close();


// OR combine declaration and open()
ofstream fout(filename, mode);
```

By default, opening an output file creates a new file if the filename does not exist; or truncates it (clear its content) and starts writing as an empty file.

open(), close() and is_open()

```
void open (const char* filename,
        ios::openmode mode = ios::in | ios::out);
  // open() accepts only C-string. For string object, need to use c_str() to get the C-string

void close ();    // Closes the file, flush the buffer and disconnect from stream object

bool is_open ();  // Returns true if the file is successfully opened
```

File Modes

File modes are defined as static public member in ios_base superclass. They can be referenced from ios_base or its subclasses - we typically use subclass ios. The available file mode flags are:

1. ios::**in** - open file for input operation
2. ios::**out** - open file for output operation
3. ios::**app** - output appends at the end of the file.
4. ios::**trunc** - truncate the file and discard old contents.
5. ios::**binary** - for binary (raw byte) IO operation, instead of character-based.
6. ios::**ate** - position the file pointer "at the end" for input/output.

You can set multiple flags via bit-or (|) operator, e.g., ios::out | ios::app to append output at the end of the file.
For output, the default is ios::out | ios::trunc. For input, the default is ios::in.

File Input

The steps are:

1. Construct an istream object.
2. Connect it to a file (i.e., file open) and set the mode of file operation.
3. Perform output operation via extraction << operator or read(), get(), getline() functions.
4. Disconnect (close the file) and free the istream object.

```
#include <fstream>

.......
```

```
ifstream fin;
fin.open(filename, mode);
.....
fin.close();


// OR combine declaration and open()
ifstream fin(filename, mode);
```

By default, opening an input file ....

Example on Simple File IO

```
1/* Testing Simple File IO (TestSimpleFileIO.cpp) */
2#include <iostream>
3#include <fstream>
4#include <cstdlib>
5#include <string>
6using namespace std;
7
8int main() {
 9  string filename = "test.txt";
10
11  // Write to File
12  ofstream fout(filename.c_str());  // default mode is ios::out | ios::trunc
13  if (!fout) {
14     cerr << "error: open file for output failed!" << endl;
15     abort();  // in <cstdlib> header
16  }
17  fout << "apple" << endl;
18  fout << "orange" << endl;
19  fout << "banana" << endl;
20  fout.close();
21
22  // Read from file
23  ifstream fin(filename.c_str());  // default mode ios::in
24  if (!fin) {
25     cerr << "error: open file for input failed!" << endl;
26     abort();
27  }
28  char ch;
29  while (fin.get(ch)) {  // till end-of-file
30     cout << ch;
31  }
```

```
32   fin.close();
33   return 0;
34}
```

Program Notes:

- Most of the <fstream> functions (such as constructors, open()) supports filename in C-string only. You may need to extract the C-string from string object via the c_str() member function.
- You could use is_open() to check if the file is opened successfully.
- The get(char &) function returns a null pointer (converted to false) when it reaches end-of-file.

Binary file, read() and write()

We need to use read() and write() member functions for binary file (file mode of ios::binary), which read/write raw bytes without interpreting the bytes.

```
 1/* Testing Binary File IO (TestBinaryFileIO.cpp) */
 2#include <iostream>
 3#include <fstream>
 4#include <cstdlib>
 5#include <string>
 6using namespace std;
 7
 8int main() {
 9   string filename = "test.bin";
10
11   // Write to File
12   ofstream fout(filename.c_str(), ios::out | ios::binary);
13   if (!fout.is_open()) {
14      cerr << "error: open file for output failed!" << endl;
15      abort();
16   }
17   int i = 1234;
18   double d = 12.34;
19   fout.write((char *)&i, sizeof(int));
20   fout.write((char *)&d, sizeof(double));
21   fout.close();
22
23   // Read from file
24   ifstream fin(filename.c_str(), ios::in | ios::binary);
25   if (!fin.is_open()) {
26      cerr << "error: open file for input failed!" << endl;
27      abort();
28   }
29   int i_in;
```

```
30   double d_in;
31   fin.read((char *)&i_in, sizeof(int));
32   cout << i_in << endl;
33   fin.read((char *)&d_in, sizeof(double));
34   cout << d_in << endl;
35   fin.close();
36   return 0;
37}
```

Random Access File

Random access file is associated with a file pointer, which can be moved directly to any location in the file. Random access is crucial in certain applications such as databases and indexes.

You can position the input pointer via seekg() and output pointer via seekp(). Each of them has two versions: absolute and relative positioning.

```
// Input file pointer (g for get)
istream & seekg (streampos pos);  // absolute position relative to beginning
istream & seekg (streamoff offset, ios::seekdir way);
    // with offset (positive or negative) relative to seekdir:
    // ios::beg (beginning), ios::cur (current), ios::end (end)
streampos tellg ();  // Returns the position of input pointer

// Output file pointer (p for put)
ostream & seekp (streampos pos);  // absolute
ostream & seekp (streamoff offset, ios::seekdir way);  // relative
streampos tellp ();   // Returns the position of output pointer
```

Random access file is typically process as binary file, in both input and output modes.

15.5 Standard Template Library

The STL (Standard Template Library) is a generic collection of class templates and algorithms that allow programmers to easily implement standard data structures like queues, lists, and stacks.

The STL contains several kinds of entities. The three most important are containers, algorithms, and iterators.

A **container** is a way that stored data is organized in memory. Two kinds of containers: stacks and linked lists. Another container, the array, is so common that it's built into C++ (and most other computer languages). However, there are many other kinds of containers, and the STL

includes the most useful. The STL containers are implemented by template classes, so they can be easily customized to hold different kinds of data.

Containers in the STL fall into two main categories:

sequence Container and associative Container.

The sequence containers are vector, list, and deque.

The associative containers are set, multiset, map, and multimap.

In addition, several specialized containers are derived from the sequence containers. These are stack, queue, and priority queue.

the pair, the vector and the map are used more often STL classes

The pair The STL pair is nothing more than a template struct with two fields.

```
template <class U, class V>
 struct  pair
{ U first;
 V second;
pair(const  U&  first = U(),  const  V&  second = V() )  :  first(first), second(second) {}
 };
 template<class U, class V>
pair<U,V>   make_pair(const  U&  first ,  const  V&   second);
```

Notice that the pair is a struct rather than a class. When handling a pair, you're free to directly access the first and second fields, since there are no private access modifiers in place to prevent you. (Had it been declared a class, the first and second fields would have been private by default. We don't need to encapsulate the fields of a pair, because it's hardly a secret what a pair really is.) The details of the pair are trivial, but they deserve specific mention up front, because many other STL classes depend on them. All of the associative containers (map, hash_map, and multimap) require a pair be used to insert new data.

```
Map< string, int> portfolio;
 portfolio.insert(make_pair(string("LU"), 400));
portfolio.insert(make_pair(string("AAPL"), 80));
portfolio.insert(make_pair(string("GOOG"), 6500));
```

The calls to insert do what you'd expect. After the third insertion, our stock portfolio consists of 400 shares of Lucent stock, 80 shares of Apple stock, and 6500 shares of Google. We'll talk more about the map in a few paragraphs. The point here is that the pair comes up at least as often as the map does.

Algorithms in the STL are procedures that are applied to containers to process their data in various ways. For example, there are algorithms to sort, copy, search, and merge data.

Algorithms are represented by template functions. These functions are not member functions of the container classes. Rather, they are standalone functions. Indeed, one of the striking characteristics of the STL is that its algorithms are so general. You can use them not only on STL containers, but on ordinary C++ arrays and on containers you create yourself. (Containers also include member functions for more specific tasks.)

Iterators are a generalization of the concept of pointers: they point to elements in a container. You can increment an iterator, as you can a pointer, so it points in turn to each element in a container. Iterators are a key part of the STL because they connect algorithms with containers.

15.6 Short Answer Questions

1.What is header file with example?

Ans. A **header file** is a **file** with extension .h which contains C function declarations and macro definitions to be shared between several source **files**. There are two types of**header files**: the **files** that the programmer writes and the **files** that comes with your compiler.

2. What is difference between header file and library file?

Ans. A **header file** describes how to call the functionality, a **library** contains the compiled code that implements this functionality. **LIBRARY FILE** is that in which definition of a particular function is written. MATH.H is a **HEADER FILE** while MATH.LIB is **library file**. Working of **HEADER File and LIBRARY in a** Program.

3. What does a header file do in C++?

Ans. **Header Files** in C++ **Header files** contain definitions of Functions and Variables, which is imported or used into any **C++** program by using the pre-processor #**include** statement. **Header file** have an extension ".h" which contains **C++**function declaration and macro definition.

4.What is meant by Vector in the container library contains?

Ans. C++ **Vectors** – std::**vector** – **Containers Library**. **Vectors** are sequence **container**(same as dynamic arrays) which resizes itself automatically. ... This **means** that a pointer to an element of a **vector** may be passed to any function that expects a pointer to an element of an array.

5.What is a static library in C++?
Ans. In computer science, a **static library** or statically-linked **library** is a set of routines, external functions and variables which are resolved in a caller at compile-time and copied into a target application by a compiler, linker, or binder, producing an object file and a stand-alone executable.

6.What are library functions in C++?

Ans. The C++ Standard Library provides a rich collection of functions for performing common mathematical calculations, **string** manipulations, character manipulations, input/output, error checking and many other useful **operations**.

UNIT-8 Operator Overloading

Index

8.1 Operator overloading

The operator overloading feature of C++ is one of the method of realizing polymorphism. The word polymorphism is derived from the greek words poly and morphism .poly refers to many or multiple and morphism refers to actions, i.e., performing many actions with a single operator.

Operator Category	Operators
Arithmatic	+ , - , *, / , %
Bit-Wise	& , | , ~ , ^
Logical	&& , || , !
Relational	>, < ,<= ,>= , ==, !==
Assignment or initialization	=
Arithmatic Assignment	+= ,-= ,*= , /= ,%= ,&= ,|= ,^=
Shift	<< , >> , <<= , >>=
Unary	++ , --
	+, -

C++ overloadable operators

8.2 Assignment Operator Overloading

The compiler copies all the members of a user-defined source object to a destination object in an assignment statement, when its members are statically allocated. The data members, which are dynamically allocated, must be copied to the destination object explicitly by overloading the assignment operator.

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

class Distance {
  private:
    int feet;          // 0 to infinite
```

```
   int inches;          // 0 to 12

  public:
    // required constructors
    Distance() {
      feet = 0;
      inches = 0;
    }
    Distance(int f, int i) {
      feet = f;
      inches = i;
    }
    void operator = (const Distance &D ) {
      feet = D.feet;
      inches = D.inches;
    }

    // method to display distance
    void displayDistance() {
      cout << "F: " << feet <<  " I:" <<  inches << endl;
    }
};

int main() {
  Distance D1(11, 10), D2(5, 11);

  cout << "First Distance : ";
  D1.displayDistance();
  cout << "Second Distance :";
  D2.displayDistance();

  // use assignment operator
  D1 = D2;
  cout << "First Distance :";
  D1.displayDistance();

  return 0;
}
```

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

```
First Distance : F: 11 I:10
Second Distance :F: 5 I:11
First Distance :F: 5 I:11
```

8.3 Overloading Arithmetic operator:

```
#include<iostream.h>
#include<conio.h>

class FLOAT

   float no;
   public:
   FLOAT(){}
   void getdata()
   {
      cout<<"\n ENTER AN FLOATING NUMBER :";
      cin>>no;
    }
    void putdata()
    {
      cout<<"\n\nANSWER IS                :"<<no;
    }
    FLOAT operator+(FLOAT);
    FLOAT operator*(FLOAT);
    FLOAT operator-(FLOAT);
    FLOAT operator/(FLOAT);
};
FLOAT FLOAT::operator+(FLOAT a)
{
   FLOAT temp;
   temp.no=no+a.no;
   return temp;
}
FLOAT FLOAT::operator*(FLOAT b)
{
   FLOAT temp;
   temp.no=no*b.no;
   return temp;
}
FLOAT FLOAT::operator-(FLOAT b)
{
   FLOAT temp;
   temp.no=no-b.no;
   return temp;
}
FLOAT FLOAT::operator/(FLOAT b)
{
   FLOAT temp;
```

```
   temp.no=no/b.no;
   return temp;
}


main()
{
  clrscr();
  FLOAT a,b,c;
  a.getdata();
  b.getdata();

  c=a+b;
  cout<<"\n\nAFTER ADDITION OF TWO OBJECTS";
  c.putdata();
  cout<<"\n\nAFTER MULTIPLICATION OF TWO OBJECTS";
  c=a*b;
  c.putdata();
  cout<<"\n\nAFTER SUBSTRACTION OF TWO OBJECTS";
  c=a-b;
  c.putdata();
  cout<<"\n\nAFTER DIVISION OF TWO OBJECTS";
  c=a/b;
  c.putdata();
  getch();
}
```

8.4 Arithmetic Assignment operator

C++ provides a short form when a variable is incremented, decremented etc. For example

j = j + 3; and j = j -3;

can also be written as

j += 3; and j -= 3;

and

j =j * 3;

can also be written as

j*=3;

+=, -=, *= are arithmetic assignment operators. The other arithmetic assignment operators are /=, %= .

Write C++ program illustrates the use of arithmetic assignment operators.

```
#include<iostream.h>
#include<conio.h>
void   main()
{
        int         count=10;
         clrscr();
         cout<<"Initialvalue of Count is: ";
          cout<<count<<"\n" ;
          count+=1;
          cout<<" count "<<count<<"\n";
          count-=2;
          cout<<"count "<<count<<"\n";
          count*=3;
          cout<<" count "<<count<<"\n";
          count/=2;
          cout<<"count "<<count<<"\n";
          getch();
}
```

Output:
Initial value of Count is:10
Count 11
Count 9
Count 27
Count 13

8.5 Operators overloading relational operator

In C++ Programming, the values stored in two variables can be compared using following operators and relation between them can be determined.

Various C++ relational operators available are-

Operator	Meaning
>	Greaterthan,
>=	Greaterthan or equal to,
==	Is equal to,
!=	Is not equal to,
< =	Less than or equal to,
<	Less than.

Program to demonstrate overloading of relational operator

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

class Distance {
  private:
    int feet;             // 0 to infinite
    int inches;           // 0 to 12
  public:
    // required constructors
    Distance(){
      feet = 0;
      inches = 0;
    }
    Distance(int f, int i){
      feet = f;
      inches = i;
    }
    // method to display distance
    void displayDistance() {
      cout << "F: " << feet << " I:" << inches <<endl;
    }
    // overloaded minus (-) operator
    Distance operator- () {
      feet = -feet;
      inches = -inches;
      return Distance(feet, inches);
    }
    // overloaded < operator
    bool operator <(const Distance& d) {
      if(feet < d.feet) {
        return true;
      }
      if(feet == d.feet && inches < d.inches) {
        return true;
      }
      return false;
    }
};

int main() {
  Distance D1(11, 10), D2(5, 11);

  if( D1 < D2 ) {
    cout << "D1 is less than D2 " << endl;
  } else {
    cout << "D2 is less than D1 " << endl;
```

```
    }

    return 0;
}
```

Output

```
D2 is less than D1
```

8.6 Overloading stream operators

In C++, stream insertion operator “<<” is used for output and extraction operator “>>” is used for input.

We must know following things before we start overloading these operators.
1) cout is an object of ostream class and cin is an object istream class
2) These operators must be overloaded as a global function. And if we want to allow them to access private data members of class, we must make them friend.

Why these operators must be overloaded as global?
In operator overloading, if an operator is overloaded as member, then it must be a member of the object on left side of the operator. For example, consider the statement “ob1 + ob2” (let ob1 and ob2 be objects of two different classes). To make this statement compile, we must overload ‘+’ in class of ‘ob1’ or make ‘+’ a global function.

The operators ‘<<' and '>>' are called like 'cout << ob1' and 'cin >> ob1'. So if we want to make them a member method, then they must be made members of ostream and istream classes, which is not a good option most of the time. Therefore, these operators are overloaded as global functions with two parameters, cout and object of user defined class.

C++ program to demonstrate overloading of <> operators.

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

class Complex
{
private:
    int real, imag;
public:
    Complex(int r = 0, int i =0)
    {  real = r;   imag = i; }
    friend ostream & operator << (ostream &out, const Complex &c);
    friend istream & operator >> (istream &in,  Complex &c);
};
```

```
ostream & operator << (ostream &out, const Complex &c)
{
   cout << c.real;
   cout << "+i" << c.imag << endl;
   return out;
}

istream & operator >> (istream &in,  Complex &c)
{
   cout << "Enter Real Part ";
   cin >> c.real;
   cout << "Enter Imagenory Part ";
   cin >> c.imag;
   return in;
}

int main()
{
  Complex c1;
  cin >> c1;
  cout << "The complex object is ";
  cout << c1;
  return 0;
}
```

Output:

```
Enter Real Part 10

Enter Imagenory Part 20

The complex object is 10+i20
```

8.7 Data Conversion

We know that the = operator will assign a value from one variable to another, in statements like intvar1 = intvar2; where intvar1 and intvar2 are integer variables. we may also have noticed that = assigns the value of one user-defined object to another, provided they are of the same type, in statements like dist3 = dist1 + dist2; where the result of the addition, which is type Distance, is assigned to another object of type Distance, dist3. Normally, when the value of one object is assigned to another of the same type, the values of all the member data items are simply copied into the new object. The compiler doesn't need any special instructions to use = for the assignment of user-defined objects such as Distance objects.

Data conversions include conversions between basic types and user-defined types, and conversions between different user-defined types.

Conversions Between Basic Types

When we write a statement like
intvar = floatvar;
where intvar is of type int and floatvar is of type float, we are assuming that the compiler will call a special routine to convert the value of floatvar, which is expressed in floating-point format, to an integer format so that it can be assigned to intvar. There are of course many such conversions: from float to double, char to float, and so on. Each such conversion has its own routine, built into the compiler and called up when the data types on different sides of the equal sign so dictate. We say such conversions are implicit because they aren't apparent in the listing.
For instance, to convert float to int, we can say
intvar = static_cast(floatvar);
Casting provides explicit conversion: It's obvious in the listing that static_cast() is intended to convert from float to int. However, such explicit conversions use the same built-in routines as implicit conversions.

Conversions Between Objects and Basic Types
When we want to convert between user-defined data types and basic types, we can't rely on built-in conversion routines, since the compiler doesn't know anything about user-defined types besides what we tell it.

Program for conversion between objects and basic types.

```
#include <iostream>
using namespace std;
const float MeterToFloat=3.280833;
class Distance {
        int feets;
        float inches;
        public:
        Distance()  //Distance Constructor {
                feets=0;
                inches=0.0;
        }

        Distance(float numofmeters)  //Single Parameter constructor {
                float feetsinfloat= MeterToFloat * numofmeters;
                feets=int(feetsinfloat);
                inches=12*(feetsinfloat-feets);
        }
        void displaydist()  // Method to display converted values {
                cout<<"Converted Value is: "<<feets<<"\' feets and "<<inches<<'\"'<<"
inches.";
        }
};
int main() {
        cout <<"Float to distance conversion.\n*********************************\n";
        float  meters;
        cout<<"Enter values in meter:";
```

```
        cin >>meters;
        Distance distance = meters;
        distance.displaydist();
}
```

Output:
Float to distance conversion.
****
Enter values in meter 3.3528
converted value is 11 feet and 5.79499 inches.

8.8 The increment and decrement operators

The increment operator '++' and decrement operator '- -' are examples of unary operators

/*C++ program for unary increment (++) and decrement (--) operator overloading.*/

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

class NUM
{
   private:
      int n;

   public:
      //function to get number
      void getNum(int x)
      {
         n=x;
      }
      //function to display number
      void dispNum(void)
      {
         cout << "value of n is: " << n;
      }
      //unary ++ operator overloading
      void operator ++ (void)
      {
         n=++n;
      }
      //unary -- operator overloading
      void operator -- (void)
      {
         n=--n;
```

```
        }
};
int main()
{
   NUM num;
   num.getNum(10);

   ++num;
   cout << "After increment - ";
   num.dispNum();
   cout << endl;

   --num;
   cout << "After decrement - ";
   num.dispNum();
   cout << endl;
   return 0;
}
```

Output:
After increment – value of n is: 11
After decrement - value of n is: 10

8.9 Overloading Subscript Operator [] :

The subscript operator, [], which is normally used to access array elements, can be overloaded. This is useful if you want to modify the way arrays work in C++. For example, to make a "safe" array: One that automatically checks the index numbers you use to access the array, to ensure that they are not out of bounds.

Subscript operator overload can be demonstrated by returning values from functions by reference

Three different approach to inserting and reading the array elements:
• Separate put() and get() functions.
• A single access() function using return by reference.
• The overloaded [] operator using return by reference.

Separate get() and put() Functions

The first program provides two functions to access the array elements: putel() to insert a value into the array, and getel() to find the value of an array element. Both functions check the value of the index number supplied to ensure it's not out of bounds; that is, less than 0 or larger than the array size (minus 1).
Here's the listing for ARROVER1:

```
// arrover1.cpp
// creates safe array (index values are checked before access)
```

```
 // uses separate put and get functions
 #include <iostream.h>
using namespace std;
 #include <process.h>// for exit()
const int LIMIT = 100;
class safearay
 {
 private: int arr[LIMIT];
 public:
 void putel(int n, int elvalue)
//set value of element
{
if( n< 0 || n>=LIMIT )
{
cout << "\nIndex out of bounds";
 exit(1);
}
arr[n] = elvalue;
 }
int getel(int n) const
//get value of element
{ if( n< 0 || n>=LIMIT )
 { cout << "\nIndex out of bounds";
 exit(1);
 }
return arr[n];
 }
};
 int main()
{
Int i;
 safearay sa1;
for( i=0;i<LIMIT;i++)   //insert elements
sa1.putel(i ,i*10);
for(i=9;i<LIMIT;i++)  //display elements
{
Int temp=sa1.getel(i);
Cout<<"element"<<i<<  " is  " <<temp<<endl;
}
return 0;
}
```

The data is inserted into the safe array with the putel() member function, and then displayed with getel(). This implements a safe array; you'll receive an error message if you attempt to use an out-of-bounds index.

Overloaded [] Operator Returning by Reference

```
// arrover3.cpp
// creates safe array (index values are checked before access)
// uses overloaded [] operator for both put and get

#include <iostream.h>
using namespace std;
 #include <process.h>// for exit()
const int LIMIT = 100;
class safearay
 {
 private: int arr[LIMIT];
 public:
   int& operator [](int n)    //note: return by reference
   {
    if( n< 0 || n>=LIMIT )
    { cout << "\nIndex out of bounds";
    exit(1);
    }
     return arr[n];
    }
   };
   int main()
     {
        Safearay   sa1;
       for(int j=0; j<LIMIT;j++)
        sa1[j]=j*10;
       for(int j=0; j<LIMIT;j++)
       {
          int temp=sa1[j];
         cout<< "Element " << j << " is " << temp << endl;
       }
     return 0;
}
```

In this program we can use the natural subscript expressions
sa1[j] = j*10; and
temp = sa1[j];
for input and output to the safe array

8.10 Rules for operator overloading

In C++, following are the general rules for operator overloading.

1) Only built-in operators can be overloaded. New operators can not be created.
2) Arity of the operators cannot be changed.
3) Precedence and associatively of the operators cannot be changed.

4) Overloaded operators cannot have default arguments except the function call operator () which can have default arguments.
5) Operators cannot be overloaded for built in types only. At least one operand must be used defined type.
6) Assignment (=), subscript ([]), function call ("()"), and member selection (->) operators must be defined as member functions
7) Except the operators specified in point 6, all other operators can be either member functions or a non member functions.
8) Some operators like (assignment)=, (address)& and comma (,) are by default overloaded.

8.10 Short Answer Questions:

1. What are all the operators that cannot be overloaded?

A. Following are the operators that cannot be overloaded -.

1. Scope Resolution (::)
2. Member Selection (.)
3. Member selection through a pointer to function (.*)

2. What is dynamic or run time polymorphism?

A. Dynamic or Run time polymorphism is also known as method overriding in which call to an overridden function is resolved during run time, not at the compile time. It means having two or more methods with the same name, same signature but with different implementation.

INDEX

6.1 Pointers

Pointers are an important feature of C++ (and C), while many other languages, such as Visual Basic and Java, have no pointers at all. Some operations that use pointers in C++ can be carried out in other ways. For example, array elements can be accessed with array notation rather than pointer notation and a function can modify arguments passed by reference, as well as those passed by pointers.

What are pointers for? Here are some common uses:
• Accessing array elements
• Passing arguments to a function when the function needs to modify the original argument
• Passing arrays and strings to functions
• Obtaining memory from the system
• Creating data structures such as linked lists

However, in some situations pointers provide an essential tool for increasing the power of C++.A notable example is the creation of data structures such as linked lists and binary trees. In fact, several key features of C++, such as virtual functions, the new operator, and the this pointer require the use of pointers.

Addresses and Pointers

the first key concept: Every byte in the computer's memory has an *address*. Addresses are numbers, just as they are for houses on a street. The numbers start at 0 and go up from there—1, 2, 3, and so on. If you have 1MB of memory, the highest address is 1,048,575. (Of course you have much more.)Your program, when it is loaded into memory, occupies a certain range of these addresses. That means that every variable and every function in your program starts at a particular address.

The Address-of Operator &

You can find the address occupied by a variable by using the *address-of* operator &.
Here's a program, VARADDR, that demonstrates how to do this:

```
// varaddr.cpp
// addresses of variables
#include <iostream.h>
using namespace std;
int main()
{
int var1 = 11; //define and initialize
int var2 = 22; //three variables
int var3 = 33;
cout << &var1 << endl //print the addresses
<< &var2 << endl //of these variables
<< &var3 << endl;
return 0;
}
```

This simple program defines three integer variables and initializes them to the values 11, 22,and 33. It then prints out the addresses of these variables.
The actual addresses occupied by the variables in a program depend on many factors, such as the computer the program is running on, the size of the operating system, and whether any other programs are currently in memory.

Accessing the Variable Pointed To

Suppose that we don't know the name of a variable but we do know its address. Can we access the contents of the variable?.

There is a special syntax to access the value of a variable using its address instead of its name.

Here's an example program, PTRACC, that shows how it's done:

```
// ptracc.cpp
// accessing the variable pointed to
#include <iostream.h>
using namespace std;
int main()
{
int var1 = 11; //two integer variables
int var2 = 22;
int* ptr; //pointer to integers
ptr = &var1; //pointer points to var1
cout << *ptr << endl; //print contents of pointer (11)
ptr = &var2; //pointer points to var2
cout << *ptr << endl; //print contents of pointer (22)
return 0;
}
```

This program is very similar to PTRVAR, except that instead of printing the address values in ptr, we print the integer value stored at the address that's stored in ptr. Here's the output:

```
11
22
```

The expression that accesses the variables var1 and var2 is *ptr, which occurs in each of the two cout statements.When an asterisk is used in front of a variable name, as it is in the *ptr expression, it is called the *dereference operator* (or sometimes the *indirection operator)*. It means *the value of thevariable pointed to by*.

6.2 Pointer to void

The address that you put in a pointer must be the same type as the pointer. You can't assign the address of a float variable to a pointer to int, for example:

```
float flovar = 98.6;
int* ptrint = &flovar; //ERROR: can't assign float* to int*
```

However, there is an exception to this. There is a sort of general-purpose pointer that can point to any data type. This is called a pointer to void, and is defined like this:

```
void* ptr; //ptr can point to any data type
```

Such pointers have certain specialized uses, such as passing pointers to functions that operate independently of the data type pointed to.

The next example uses a pointer to void and also shows that, if you don't use void, you must be careful to assign pointers an address of the same type as the pointer. Here's the listing for PTRVOID:

```
// ptrvoid.cpp
// pointers to type void
#include <iostream.h>
using namespace std;
int main()
{
```

```
int intvar; //integer variable
float flovar; //float variable
int* ptrint; //define pointer to int
float* ptrflo; //define pointer to float
void* ptrvoid; //define pointer to void
ptrint = &intvar; //ok, int* to int*
// ptrint = &flovar; //error, float* to int*
// ptrflo = &intvar; //error, int* to float*
ptrflo = &flovar; //ok, float* to float*
ptrvoid = &intvar; //ok, int* to void*
ptrvoid = &flovar; //ok, float* to void*
return 0;
}
```

You can assign the address of intvar to ptrint because they are both type int*, but you can't assign the address of flovar to ptrint because the first is type float* and the second is type int*. However, ptrvoid can be given any pointer value, such as int*, because it is a pointer to void.

If for some unusual reason you really need to assign one kind of pointer type to another, you can use the reinterpret_cast. For the lines commented out in PTRVOID, that would look like this:

```
ptrint = reinterpret_cast<int*>(flovar);
ptrflo = reinterpret_cast<float*>(intvar);
```

The use of reinterpret_cast in this way is not recommended, but occasionally it's the only way out of a difficult situation.

6.3 Pointer to arrays

There is a close association between pointers and arrays. Arrays and Strings, how array elements are accessed. The following program, ARRNOTE, provides a review.

```
// arrnote.cpp
// array accessed with array notation
#include <iostream.h>
using namespace std;
int main()
{ //array
int intarray[5] = { 31, 54, 77, 52, 93 };
for(int j=0; j<5; j++) //for each element,
cout << intarray[j] << endl; //print value
return 0;
}
```

The cout statement prints each array element in turn. For instance, when j is 3, the expression intarray[j] takes on the value intarray[3] and accesses the fourth array element, the integer 52. Here's the output of ARRNOTE:

Accessing the elements of Array using pointer

Similarly, array elements can be accessed using pointer notation as well as array notation. The next example, PTRNOTE, is similar to ARRNOTE except that it uses pointer notation.

```
// ptrnote.cpp
// array accessed with pointer notation
```

```
#include <iostream.h>
using namespace std;
int main()
{ //array
int intarray[5] = { 31, 54, 77, 52, 93 };
for(int j=0; j<5; j++) //for each element,
cout << *(intarray+j) << endl; //print value
return 0;
}
```

The expression *(intarray+j) in PTRNOTE has exactly the same effect as intarray[j] in ARRNOTE, and the output of the programs is identical. But how do we interpret the expression *(intarray+j)? Suppose j is 3, so the expression is equivalent to *(intarray+3). We want this to represent the contents of the fourth element of the array (52).

6.4 The new Operator

C++ provides a different approach to obtaining blocks of memory: the new operator. This operator obtains memory from the operating system and returns a pointer to its starting point. The NEWINTRO example shows how new is used:

```
// newintro.cpp
// introduces operator new
#include <iostream>
#include <cstring> //for strlen
using namespace std;
int main()
{
char* str = "Idle hands are the devil's workshop.";
int len = strlen(str); //get length of str
char* ptr; //make a pointer to char
ptr = new char[len+1]; //set aside memory: string + '\0'
strcpy(ptr, str); //copy str to new memory area ptr
cout << "ptr=" << ptr << endl; //show that ptr is now in str
delete[] ptr; //release ptr's memory
return 0;
}
```

The expression

```
ptr = new char[len+1];
```

returns a pointer to a section of memory just large enough to hold the string str, whose length len we found with the strlen() library function, plus an extra byte for the null character '\0' at the end of the string.Remember to use brackets around the size; the compiler won't object if you mistakenly use parentheses, but the results will be incorrect.

6.5 The delete Operator

If your program reserves many chunks of memory using new, eventually all the available memory will be reserved and the system will crash. To ensure safe and efficient use of memory, the new operator is matched by a corresponding delete operator that returns memory to the operating system. In NEWINTRO the statement

```
delete[] ptr;
```

returns to the system whatever memory was pointed to by ptr.

Actually, there is no need for this operator in NEWINTRO, since memory is automatically returned when the program terminates.

6.6 Pointers to Objects

Pointers can point to objects as well as to simple data types and arrays.
Consider an examples of objects defined and given a name, in statements like
Distance dist;
where an object called dist is defined to be of the Distance class.
Sometimes, however, we don't know, at the time that we write the program, how many objects we want to create. When this is the case we can use new to create objects while the program is running. As we've seen, new returns a pointer to an unnamed object. Let's look at a short example
program, ENGLPTR, that compares the two approaches to creating objects.

```
// englptr.cpp
// accessing member functions by pointer
#include <iostream.h>
using namespace std;
lass Distance   //English Distance class
{
private:
int feet;
float inches;
public:
void getdist()   //get length from user
{
cout << "\nEnter feet: "; cin >> feet;
cout << "Enter inches: "; cin >> inches;
}
void showdist() //display distance
{ cout << feet << "\'-" << inches << '\"'; }
};

int main()
{
Distance dist; //define a named Distance object
dist.getdist(); //access object members
dist.showdist(); // with dot operator
Distance* distptr; //pointer to Distance
distptr = new Distance; //points to new Distance object
distptr->getdist(); //access object members
distptr->showdist(); // with -> operator
cout << endl;
return 0;
}
```

Here's the output of the program:
Enter feet: 10 ←this object uses the dot operator
Enter inches: 6.25

10’-6.25”
Enter feet: 6 ← this object uses the -> operator
Enter inches: 4.75
6’-4.75”

6.7 Array of pointers to objects

An array of pointers to objects is often used to handle a group of object, which need not necessarily reside contiguously in memory. this approach is more flexible, in comparison with placing the objects themselves in an array, because objects can be dynamically created as and when required .the syntax for defining array of pointers to objects is same as any of the fundamental types.

Program to demonistrate Array of pointers to objects

```
#include<iostream.h>
#include<cstring.h>
using namespace std;
class city
{
 protected:
     char *name;
     int len;
public:
 city()
{
   len=0;
  name=new char[len+1];
}
void getname(void)
{
  char *s;
  s=new char[30];
  cout<< “  Enter city name: ”;
cin>>s;
len=strlen(s);
name=new  char[len+ 1];
strcpy(name,s);
}
void  printname(void)
{
 cout<< name  <<”\n”;}
};
Int main()
{
 city  *cptr[10];
int n=1;
int opt;
do
{
 cptr[n]= new city;
```

```
 cptr[n]->getname();
 n++;
cout<<"Do you want to enter one more name: \n";
cout<<"Enter 1 for  yes 0 for no:";
cin>>opt;
}while(opt);
cout<<"\n\n";
for(int i=1;i<=n;i++)
{
cptr[i]->printname();
}
return 0;
}
```

6.8 This POINTER

C++ uses a unique keyword called **this** to represent an object that invokes a member function. **this** is a pointer that points to object for which **this** function is called. For example address is same as the address of the first variable in the class structure.

This unique pointer is automatically passed to a member function when it is called. The pointer **this** act as an implicit argument to all the member functions.

```
class ClassName {

      private:
        int dataMember;

      public:
         method(int a) {

  // this pointer stores the address of object obj and access dataMember
              this->dataMember = a;
              ... .. ...
         }
}

int main() {

   ClassName obj;
   obj.method(5);
   ... .. ...
 }
```

Example 1: C++ program using this pointer to distinguish local members from parameters.

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

```
class sample
{
   int a,b;
   public:
      void input(int a,int b)
      {
         this->a=a+b;
         this->b=a-b;
      }
      void output()
      {
         cout<<"a = "<<a<<endl<<"b = "<<b;
      }
};

int main()
{
   sample x;
   x.input(5,8);
   x.output();
   getch();
   return 0;
}
```

A class *sample* is created in the program with data members *a* and *b* and member functions *input()* and *output(). input()* function receives two integer parameters *a* and *b* which are of same name as data member of class *sample*. To distinguish the local variable of *input()* data member of class, this pointer is used. When *input()* is called, the data of object inside it is represented as this->a and this->b while the local variable of function is represented simply as a and b.

Output

a = 13
b = -3

C++ Program to display address of elements of an array using both array and pointers

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

int main()
{
   float arr[5];
   float *ptr;

   cout << "Displaying address using arrays: " << endl;
   for (int i = 0; i < 5; ++i)
   {
      cout << "&arr[" << i << "] = " << &arr[i] << endl;
   }

   // ptr = &arr[0]
   ptr = arr;
```

```
   cout<<"\nDisplaying address using pointers: "<< endl;
   for (int i = 0; i < 5; ++i)
   {
      cout << "ptr + " << i << " = "<< ptr + i << endl;
   }

   return 0;
}
```

```
Output
Displaying address using arrays:
&arr[0] = 0x7fff5fbff880
&arr[1] = 0x7fff5fbff884
&arr[2] = 0x7fff5fbff888
&arr[3] = 0x7fff5fbff88c
&arr[4] = 0x7fff5fbff890

Displaying address using pointers:
ptr + 0 = 0x7fff5fbff880
ptr + 1 = 0x7fff5fbff884
ptr + 2 = 0x7fff5fbff888
ptr + 3 = 0x7fff5fbff88c
ptr + 4 = 0x7fff5fbff890
```

6.9 Multiple choice Questions

1. What is the meaning of the following declaration?

int(*p[5])();

a) p is pointer to function

b) p is array of pointer to function

c) p is pointer to such function which return type is the array

d) p is pointer to array of function

2. What is size of generic pointer in C++ (in 32-bit platform) ?

a) 2

b) 4

c) 8

d) 0

3. What is the output of this program?

```
1. #include <iostream>
2. using namespace std;
3. int main()
4. {
5.     int a[2][4] = {3, 6, 9, 12, 15, 18, 21, 24};
6.     cout << *(a[1] + 2) << *(*(a + 1) + 2) << 2[1[a]];
```

```
7.      return 0;
8.  }
```

a)151821

b)212121

c)242424

d) Compile time error

4. What is the output of this program?

```
#include <iostream>
using namespace std;
int main()
{
   int i;
   const char *arr[] = {"C", "C++", "Java", "VBA"};
   const char *(*ptr)[4] = &arr;
   cout << ++(*ptr)[2];
   return 0;
}
```

a)ava

b)java

c)c++

d) compile time error

5. What is the output of this program?

```
#include <iostream>
using namespace std;
int main()
{
   int arr[] = {4, 5, 6, 7};
   int *p = (arr + 1);
   cout << *p;
   return 0;
}
```

a)4

b)5

c)6

d)7

6. What is the output of this program?

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

```
{
        int arr[] = {4, 5, 6, 7};
        int *p = (arr + 1);
cout << arr;
 return 0;
 }
```

a)4

b)5

c)address of arr

d) 7

7. What is the output of this program?

```
    #include <iostream>
using namespace std;
int main ()
    {
      int numbers[5];
      int * p;
     p = numbers;  *p = 10;
      p++;  *p = 20;
       p = &numbers[2];  *p = 30;
   p = numbers + 3;  *p = 40;
       p = numbers;  *(p + 4) = 50;
       for (int n = 0; n < 5; n++)
          cout << numbers[n] << ",";
       return 0;
     }
```

a)10,20,30,40,50,

b)10 20 30 40 50

c)compile error

d) runtime error

8. What is the output of this program?

```
 #include <iostream>
using namespace std;
  int main()
   {
        int arr[] = {4, 5, 6, 7};
        int *p = (arr + 1);
        cout << *arr + 9;
    return 0;
```

}

a)12

b)5

c)13

d) error

9. What does the following statement mean?

int (*fp)(char*)

a) pointer to a pointer

b) pointer to an array of chars

c) pointer to function taking a char* argument and returns an int

d) function taking a char* argument and returning a pointer to int

10. The operator used for dereferencing or indirection is ____

a) *

b) &

c) ->

d) –>>

11. Choose the right option
string* x, y;

a) x is a pointer to a string, y is a string
b) y is a pointer to a string, x is a string
c) both x and y are pointers to string types
d) none of the mentioned

12. Which one of the following is not a possible state for a pointer.

a) hold the address of the specific object
b) point one past the end of an object
c) zero
d) point to a type

13.Which of the following is illegal?

a) int *ip;
b) string s, *sp = 0;
c) int i; double* dp = &i;
d) int *pi = 0;

14. What will happen in this code?

```
    int a = 100, b = 200;
int *p = &a, *q = &b;
    p = q;
```

a) b is assigned to a

b) p now points to b

c) a is assigned to b

d) q now points to a

15. What is the output of this program?

```
#include <iostream>
using namespace std;
int main()
{
 int a = 5, b = 10, c = 15;
      int *arr[ ] = {&a, &b, &c};
     cout << arr[1];
return 0;
}
```

a) 5

b) 10

c) 15

d) it will return some random number

Answers:

1 . B 2. B 3. B 4. A 5. B 6. C 7. A 8. C 9. C 10. A 11. A 12.D 13. C 14. B 15. D

UNIT-13 STREAMS

13.1 Streams in C++

A stream is a general refered as flow of data.A stream can act like a source or destination .In C++ a stream is represented by an object of a particular class. So far we've used the cin and cout stream objects. Different streams are used to represent different kinds of data flow. For example, the ifstream class represents data flow from input disk files.

C++ Streams are Objects

The input and output streams cin and cout are actually C++ objects.

Class: A C++ construct that allows a collection of variables , constants and functions to be grouped together logically under single name.

Object: A variable of a type that is a class(also often called an instance of a class).

For example , istream is actually a type name for a class. cin is the name of a variable of type istream.

So, we would say that cin is an instance or an object of class istream.

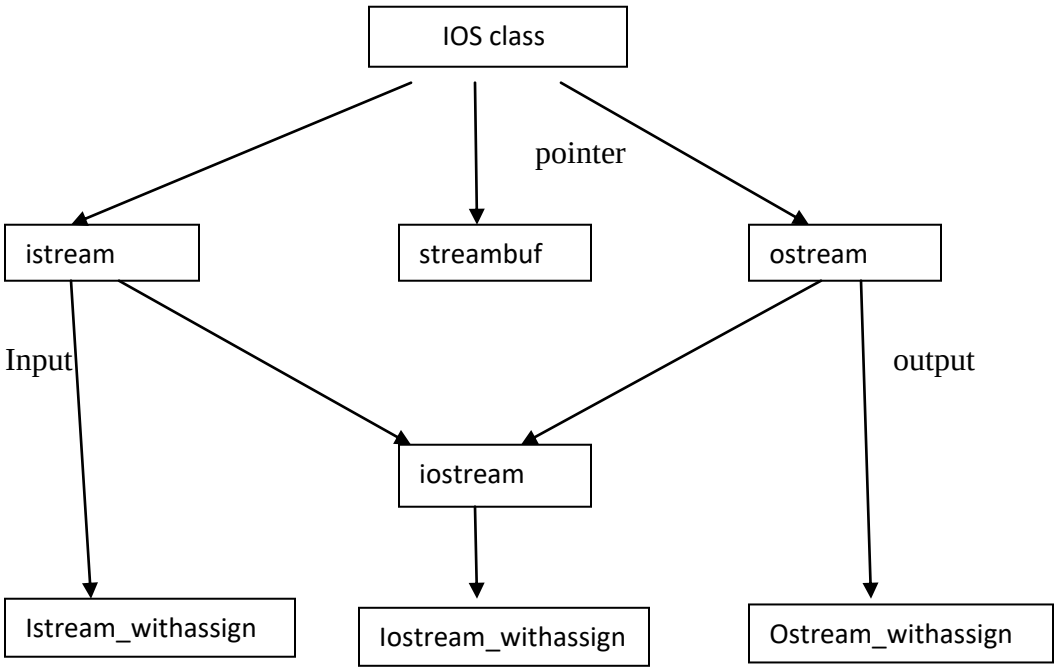
An instance of a class will usually have a number of associatated functions(called member functions) that you can use to perform operations on that object or to obtain information about it.

The C++ I/O system contains a hierarchy of class that are used to define various streams to deal with both the console and disk files. These classes are called stream classes. Figure below shows the hierarchy of the stream classes used for input and out operations with the console unit. these classes are declared in the header file iostream

Advantages of Streams

1. Simplicity.
2. You can overload existing operators and functions, such as the insertion (<>) operators, to work with classes that you create.
3. The input (>>) operator and output (<<) operator are typesafe. These operators are easier to use than scanf() and printf().

The Stream Class Hierarchy



13.2 The ios Class

The **ios** class is the base class of all the stream classes, and contains the majority of the features to operate C++ streams. The three most important features are the formatting flags, the error-status flags, and the file operation mode.

The class **ios** is declared as the virtual base class so that only one copy of its members are inherited by the **iostream.**

13.3 IOS Format Flags

Format flags are a set of enum definitions in ios. They act as on/off switches that specify choices for various aspects of input and output format and operation.

Flag	Meaning
Skipws	Skip (ignore) whitespace on input.
left	Left-adjust output [12.34].
right	Right-adjust output [12.34].
internal	Use padding between sign or base indicator and number [+ 12.34].
dec	Convert to decimal,

oct Convert to octal.

hex Convert to hexadecimal.

There are several ways to set the formatting flags, and different ones can be set in different ways. Since they are members of the ios class, you must usually precede them with the name ios and the scope-resolution operator (for example, ios::skipws).

All the flags can be set using the setf() and unsetf() ios member functions.

Look at the following example:

cout.setf(ios::left); // left justify output text

cout >> “This text is left-justified”;

cout.unsetf(ios::left); // return to default (right justified)

13.4 Stream Error States

- **eofbit**
  - Set for an input stream after end-of-file encountered
  - **cin.eof()**returns **true**if end-of-file has been encountered on **cin**
- **failbit**
  - Set for a stream when a format error occurs
  - **cin.fail()** - returns **true** if a stream operation has failed
  - Normally possible to recover from these errors
- **badbit**
  - Set when an error occurs that results in data loss
  - **cin.bad()** returns **true** if stream operation failed
  - normally nonrecoverable
- **goodbit**
  - Set for a stream if neither **eofbit**, **failbit** or **badbit** are set
  - **cin.good()** returns **true** if the **bad**, **fail** and **eof** functions would all return false.
  - I/O operations should only be performed on “good” streams
- **rdstate**
  - Returns the state of the stream
  - Stream can be tested with a **switch** statement that examines all of the state bits

- Easier to use **eof**, **bad**, **fail**, and **good** to determine state

- **clear**
  - Used to restore a stream's state to "good"
  - **cin.clear()** clears **cin** and sets **goodbit** for the stream
  - **cin.clear(ios::failbit)** actuallysets the **failbit**
    - Might do this when encountering a problem with a user-defined type

- Other operators
  - **operator!**
    - Returns **true** if **badbit** or **failbit** set
  - **operator void***
    - Returns **false** if **badbit** or **failbit** set
  - Useful for file processing

13.5 The istream Class

The istream class, which is derived from ios, performs input-specific activities, or extraction. It's easy to confuse extraction and the related output activity, insertion.

istream Functions

Function	Purpose
>>	Formatted extraction for all basic (and overloaded) types.
get(ch);	Extract one character into ch.
get(str)	Extract characters into array str, until '\n'.
get(str, MAX)	Extract up to MAX characters into array.
get(str, DELIM)	Extract characters into array str until specified delimiter (typically '\n'). Leave delimiting char in stream.
get(str, MAX, DELIM)	Extract characters into array str until MAX characters or the DELIM character. Leave delimiting char in stream.
getline(str, MAX, DELIM)	Extract characters into array str, until MAX characters or the DELIM character. Extract delimiting character.
putback(ch)	Insert last character read back into input stream.
ignore(MAX, DELIM)	Extract and discard up to MAX characters until (and including) the specified delimiter (typically '\n').

peek(ch)	Read one character, leave it in stream.
count = gcount()	Return number of characters read by a (immediately preceding) call to get(), getline(), or read().
read(str, MAX)	For files—extract up to MAX characters into str, until EOF.
seekg()	Set distance (in bytes) of file pointer from start of file.
seekg(pos, seek_dir)	Set distance (in bytes) of file pointer from specified place in file. seek_dir can be ios::beg, ios::cur, ios::end.
pos = tellg(pos)	Return position (in bytes) of file pointer from start of file.

13.6 The ostream Class

The ostream class handles output or insertion activities.

ostream Functions

Function	Purpose
<<	Formatted insertion for all basic (and overloaded) types.
put(ch)	Insert character ch into stream.
flush()	Flush buffer contents and insert newline.
write(str, SIZE)	Insert SIZE characters from array str into file.
pos = tellp()	Return position of file pointer, in bytes.

13.7 Unformatted Input/Output Functions

put(), get() and getline()

The ostream's member function put() can be used to put out a char. put() returns the invoking ostream reference, and thus, can be cascaded. For example,

```
// ostream class
ostream & put (char c);  // put char c to ostream
// Examples
cout.put('A');
cout.put('A').put('p').put('p').put('\n');
cout.put(65);
```

```
// istream class
// Single character input
int get ();
```

```
    // Get a char and return as int. It returns EOF at end-of-file
istream & get (char & c);
    // Get a char, store in c and return the invoking istream reference

// C-string input
istream & get (char * cstr, streamsize n, char delim = '\n');
    // Get n-1 chars or until delimiter and store in C-string array cstr.
    // Append null char to terminate C-string
    // Keep the delim char in the input stream.
istream & getline (char * cstr, streamsize n, char delim = '\n');
    // Same as get(), but extract and discard delim char from the
    // input stream.

// Examples
int inChar;
while ((inChar = cin.get()) != EOF) {  // Read till End-of-file
  cout.put(inchar);
}
```

read(), write() and gcount()

```
// istream class
istream & read (char * buf, streamsize n);
    // Read n characters from istream and keep in char array buf.
    // Unlike get()/getline(), it does not append null char at the end of input.
    // It is used for binary input, instead of C-string.
streamsize gcount() const;
    // Return the number of character extracted by the last unformatted input operation
    // get(), getline(), ignore() or read().

// ostream class
ostream & write (const char * buf, streamsize n)
    // Write n character from char array.
```

Other istream functions - peek() and putback()

```
char peek ();
    //returns the next character in the input buffer without extracting it.

istream & putback (char c);
    // insert the character back to the input buffer.
```

13.8 What is a Manipulator?

Manipulators are operators used in C++ for formatting output. The data is manipulated by the programmer‘s choice of display.
There are numerous manipulators available in C++. Some of the more commonly used manipulators are provided here below:

endl Manipulator:

This manipulator has the same functionality as the ‘ \n‘ newline character.
For example:
cout <<” Exforsys” << endl;
cout << “Training”;
produces the output:
Exforsys
Training

setw Manipulator:
This manipulator sets the minimum field width on output. The syntax is:
setw(x)
Here setw causes the number or string that follows it to be printed within a field of x characters wide and x is the argument set in setw manipulator. The header file that must be included while using setw manipulator is <iomanip.h> .

```
#include <iostream.h>
#include <iomanip.h>
void main( )
{
int  x1=12345,x2= 23456, x3=7892;
cout << setw(8) << Exforsys << setw(20) << Values << endl
<< setw(8) << E1234567 << setw(20)<< x1 << endl

<< setw(8) << S1234567 << setw(20)<< x2 << endl

<< setw(8) << A1234567 << setw(20)<< x3 << endl;
}
```

The output of the above example is:
setw(8) setw(20)
Exforsys Values
E1234567 12345
S1234567 23456
A1234567 7892

setfill Manipulator:

This is used after setw manipulator. If a value does not entirely fill a field, then the cha racter specified in the setfill argument of the manipulator is used for filling the fields.

```
#include <iostream.h>
#include <iomanip.h>
void main( )
{
cout << setw(10) << setfill('$') << 50 << 33 << endl;
}
```

The output of the above program is

$$$$$$$$5033

This is because the setw sets 10 width for the field and the number 50 has only 2 positions in it. So the remaining 8 positions are filled with $ symbol which is specified in the setfill argument.

setprecision Manipulator:

The setprecision Manipulator is used with floating point numbers. It is used to set the number of digits printed to the right of the decimal point. This may be used in two forms:

fixed

scientific

These two forms are used when the keywords fixed or scientific are appropriately used before the setprecision manipulator.

The keyword fixed before the setprecision manipulator prints the floating point number in fixed notation.

The keyword scientific before the setprecision manipulator prints the floating point number in scientific notation.

```
#include <iostream.h>
#include <iomanip.h>
void main( )
{
float x = 0.1;
cout << fixed << setprecision(3) << x << endl;
cout << sceintific << x << endl;
}
```

The above gives ouput as:

0.100

1.000000e-001

The first cout statement contains fixed notation and the setprecision contains argument 3. This means that three digits after the decimal point and in fixed notation will output the first cout statement as 0.100. The second cout produces the output in scientific notation. The default value is used since no setprecision value is provided.

A1234567 7892

13.9 Some Important Questions

1. What are streams? Why they are useful?

2. What are input and output streams?

3. Briefly describe the class hierarchy provided by c++ for stream handling.

4. What is the difference between a text file and a binary file?

5. How can a file be opened for both reading and writing?

6. Describe the following manipulators setw(), setprecision(), setfill(),Setiosflags(),resetiosflags().

7. How will you create manipulators?

8. Write the syntax and use of getline () and write () functions.

9. What are the differences between manipulators and ios functions?

UNIT-2 Class and Object

INDEX

2.1 Defining Class

A *class* is a blueprint or template or set of instructions to build a specific type of object. Every object is built from a class.It allows the data and functions to be hidden ,if necessary from external use.

A class specification has two parts:

1. Class declaration
2. Class function definitions

The class declaration describes the ti1ie and scope of its members.

The general form of a class declaration is:

```
class class _name
{
private:
variable declarations;  //class members
function declarations;  //class members
public:
variable declarations;  //class members
function declaration;  //class members
}
```

They are grouped under two sections, namely, private and public to denote which are private and which are public.the keywords **private** and **public** are known as visibility labels.

2.2 Object of a Class

An object is said to be an *instance of* a class. An Object has the same relationship to a class that a variable has to a data type.

A simple Class Example

```
// classobj.cpp
#include <iostream>
using namespace std;
class classobj //define a class
{
private:
int somedata; //class data
public:
void setdata(int d) //member function to set data
{ somedata = d; }
void showdata() //member function to display data
{ cout << "Data is " << somedata << endl; }
};
```

```
int main()
{
classobj s1, s2; //define two objects of class smallobj
s1.setdata(106); //call member function to set data
s2.setdata(177);
s1.showdata(); //call member function to display data
s2.showdata();
return 0;
}
```

private and public

The body of the class contains two unfamiliar keywords: private and public. What is their purpose?.A key feature of object-oriented programming is *data hiding*. The primary mechanism for hiding data is to put it in a class and make it private. Private data or functions can only be accessed from within the class. Public data or functions, on the other hand, are accessible from outside the class.

Write A C++ Program To Calculate Simple Interest. Hide The Data Elements Of The Class Using Private Keyword.

```
# include <iostream.h>
# include <conio.h>
class bank
{
 private:
 float p;
 float r;
 float t;
 float si;
 float amount;
 public:
          void read ( )
       {
            cout <<" Principle Amount : ";
            cin>>p ;
            cout<<" Rate of Interest : ";
            cin>>r;
            cout <<" Number of years : ";
            cin>>t;
            si= (p *r*t) /100;
            amount = si + p;
       }
          void show( )
      {
           cout<<"\n Principle Amount: "<<p;
           cout <<"\n Rate of Interest: "<<r;
```

```
            cout <<"\n Number of years: "<<t;
            cout <<"\n Interest : "<<si;
            cout <<"\n Total Amount : "<<amount;
        }
};
        void main ( )
        {
            clrscr ( );
            bank b ;
            b.read ( );
            b.show ( );
            getch();
        }
```

2.3 Creating Objects

Once a class has been declared, we can create variables of that type by using the class name (like any other built-in type variable).

Example classobj s1,s2 ; // memory for s1 ,s2 is created

Accessing Class Members

The private data of a class can be accessed only through the member functions of that class. The main() cannot contain statements that access **somedata** directly. The following is the format for calling a member function:

```
Object-name.function-name(actual-arguments);
```

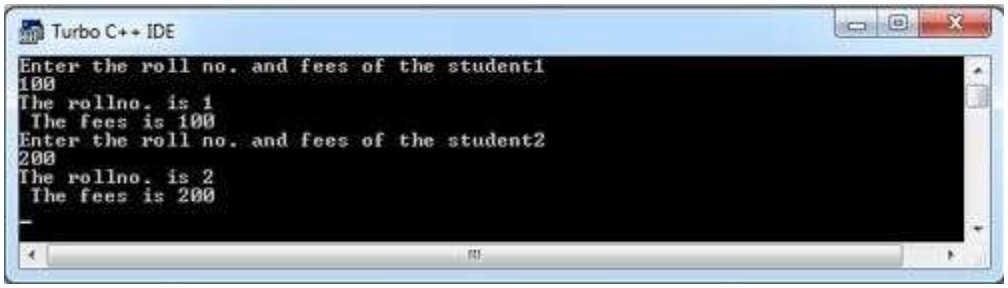
For example, the function call statement

```
s1.setdata(106);
s2.setdata(177);
```

Write A C++ Program To Illustrate The Concept Of Class Having Two Objects

```
#include<iostream.h>
#include<conio.h>
class student
{
   private:
```

```
        int rn;
        float fees;
    public:
        void readdata()
       {
          cout<<"Enter the roll no. and fees of the student";
          cin>>rn>>fees;
        }
        void writedata()
        {
        cout<<"The rollno. is "<<rn<<endl;
        cout<<" The fees is "<<fees<<endl;
        }
};
      void main()
    {
        clrscr();
        student st, st2;
        st.readdata();
        st.writedata();
        st2.readdata();
        st2.writedata();
        getch();
     }
```



2.4 **Constructors**

A constructor is a Special member function that is executed automatically whenever an object is created. The task of a constructor is to initialize the objects of its class. Its name is same as the class name. The constructor is invoked whenever an object of its associated class is created. It is called constructor because it constructs the values of data members of the class.

A constructor is dec1ared and defined as follows:

```
// class with a constructor
class construct
{
int m, n;
public:
construct (voi d) ; // constructor declaration
```

```
……..
……
};
 construct:: construct (void) // constructor definition
{ m = 0;
 n= 0;
 }
```

The constructor functions have some special characteristics. These are :

- They should be declared in the public section.
- They arc invoked automatically when the objects are created.
- They do not have return types, not even void and therefore, they cannot return values.
- They cannot be inherited, though a derived class can call the base class constructor.
- Like other C++ functions, they can have default arguments.
- Constructors cannot be virtual.
- We cannot refer to their addresses.

There are three types of constructors based on parameter list.

1. Default Constructor.
2. Parameterized Constructor.
3. Copy Constructor.

Default Constructor : A constructor that accepts no parameters is called the default constructor. The default constructor for **class C is C::C().** lf no such constructor is defined, then the compiler supplies a default constructor.
Therefore a statement such as
C c; invokes the default constructor of the compiler to create the object c.

Parameterized Constructor: when the objects are created, the constructor that can take arguments are called parameterized constructor.

```
class construct
{
 int m, n;
public:
 construct (int  x,int  y) ; // constructor declaration
……..
……
};
 construct:: construct (int  x,int  y ) // constructor definition
{ m = x;
 n= y;
 }
```

when a constructor has been parameterized, the object declaration statement such as
construct c1; //this may not work.

We must pass the initial values as arguments to the constructor function when an object is declared. This can be done in two ways:

• By calling the constructor explicitly.

• By calling the constructor implicitly.

The following declaration illustrate the first method:

construct c1= construct(0,10); // explicit call

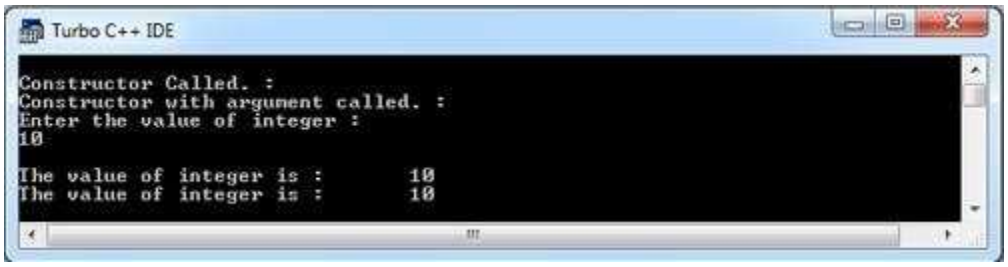
This statement creates an integer object c1 and passes the values 0 and 10 to it.

The second is implemented as follows:

construct c1(0,10); //implicit call

```
// Program to define in different way about the overload constructors
#include<iostream.h>
#include<conio.h>
class integer
{
    private:
    int a;
    public:
            integer()
        {
             a=0;
            cout<<endl<<"Constructor Called. : ";
        }
            integer(int i)
        {
             a=i;
            cout<<endl<<"Constructor with argument called. : ";
        }
           ~integer()
        {
           cout<<endl<<"Destructor Called.";
        }
           void get()
        {
            cout<<endl<<"Enter the value of integer :" <<endl;
            cin>>a;
        }
            void show()
        {
            cout<<endl<<"The value of integer is :"<<"\t"<<a;
        }
};
        void main()
   {
        clrscr();
        integer x, y(10);
        x.get();
```

```
        x.show();
        y.show();
        getch();
    }
```



copy constructor

An object can be initialized with another object of same type. This is same as copying the contents of a class to another class.

In the above program, if you want to initialise an object *A3* so that it contains same values as *A2*, this can be performed as:

```
……
int main()
{
  Area A1, A2(2, 1);
  // Copies the content of A2 to A3
   Area A3(A2);
     OR,
   Area A3 = A2;
}
```

2.5 Destructor

When an object no longer needed it can be destroyed. A class can have special member function called the destructor, which is invoked when an object is destroyed.
This function complements the operation performed by any of the constructor.

C++ destructors are used to de-allocate the memory that has been allocated for the object by the constructor.
Destructor in c++ programming

Its syntax is same as constructor except the fact that it is preceded by the tilde sign.

```
~class_name() { };   //syntax of destructor
/*...syntax of destructor....*/
 class class_name
```

```
 {
     public:
       class_name();    //constructor.
      ~class_name();    //destructor.
}

/*.....A program to highlight the concept of destructor..... */
#include <iostream>
using namespace std;
class ABC
{
   public:
      ABC () //constructor defined
      {
            cout << "Hey look I am in constructor" << endl;
      }
      ~ABC() //destructor defined

      {
          cout << "Hey look I am in destructor" << endl;
      }
};

int main()
{
    ABC cc1; //constructor is called
    cout << "function main is terminating...." << endl;
    /*....object cc1 goes out of scope ,now destructor is being called...*/
    return 0;
}  //end of program
```

Output

```
Hey look I am in constructor
function main is terminating....
Hey look I am in destructor
```

2.6 Constant Object

C++ allows to define constant objects of user-defined classes similar to constants of standard datatypes. The syntax for defining a constant object.

```
Classname const  Objectname (parameter list);
```

const Data Members

The data members of a constant object can be initialized only by a constructor, as part of object creation procedure. Once a constant object is created , no member function of its class can

modify its data members. Such data members are termed as read-only data members and object is termed as constant, or read-only object.

Data members that are both static and const have their own rules for initialization.

const Objects

An object of a class may be declared to be const, just like any other C++ variable. For example:

```
const Date birthday(7, 3, 1969);
```

declares a const object named birthday of the Date class.

The const property of an object goes into effect after the constructor finishes executing and ends before the class's destructor executes. So the constructor and destructor can modify the object, but other methods of the class can't.

Only const methods can be called for a const object.

const Methods

You can declare a method of a class to be const. This must be done both in the method's prototype and in its definition by coding the keyword const after the method's parameter list. For example:

```
int get_month() const;          // prototype in Date class definition

int Date::get_month() const     // method definition
  {
  return month;
  }
```

2.7 Static Data Member:

A data member in a class can be declared as static. A static data member has certain special characteristics. These are:

Only one copy of that member is created for the entire class and is shared by all the objects of that class, so it is also called class data member. It must be declared as private data member.

- It can be accessed not only by the object name but also with the class name.
- It is initialized to zero when the first object of its class is created.

Below given program illustrates the use of static data member:

```
class rectangle
 {
 private:
 int length;
int width;
int area;
 static int count;
```

```
 public:
void input()
 {
 cin>>length;
 cin>>width;
 }
 void output()
{
count++;
area=length*width;
cout<< area of rectangle << count << “is = “<<area;
}
};
 void main()
 {
 rectangle r1, r2, r3;
r1.input();
 r1.output();
r2.input();
 r2.output();
r3.input();
 r3.output();
 }
```

2.8 Static Member Function:

A member function in a class can be declared as static. A static member function has certain special characteristics. These are:

- A static function can have access to only other static members.
- A static member function can be accessed not only by the object name but also with the class name.

Below given program illustrates the use of static data member:

```
class rectangle
 {
private:
 int length;
 int width;
 int area;
 static int count;
public:
void input()
 {
cin>>length; cin>>width;
}
void static fcount()
 {
```

```
count++;
 cout>>count;
}
void output()
 {
 count++;
area=length*width;
 cout<< area of rectangle << fcount() << "is = "<<area;
}
};
void main()
{
rectangle r1, r2, r3;
r1.input();
r1.output();
r2.input();
r2.output();
 r3.input();
r3.output();
}
```

2.9 Short Questions with Answers

1.What is a class?

A. A class is simply a representation of a type of object. It is the blueprint/ plan/ template that describes the details of an object.

2. What is an object?

A. An object is an instance of a class. It has its own state, behavior, and identity.

3.Define a constructor?

A. A constructor is a method used to initialize the state of an object, and it gets invoked at the time of object creation. Rules for constructor are:

- Constructor Name should be same as class name.
- A constructor must have no return type.

4. Define Destructor?

A. A destructor is a method which is automatically called when the object is made of scope or destroyed. Destructor name is also same as class name but with the tilde symbol before the name.

5.What is an abstract class?

A. An abstract class is a class which cannot be instantiated. Creation of an object is not possible with an abstract class, but it can be inherited. An abstract class can contain only Abstract method. Java allows only abstract method in abstract class while for other languages allow non-abstract method as well.

6.What are access modifiers?

A. Access modifiers determine the scope of the method or variables that can be accessed from other various objects or classes. There are 5 types of access modifiers, and they are as follows:

- Private.
- Protected.
- Public.
- Friend.
- Protected Friend.

7. What are sealed modifiers?

A. Sealed modifiers are the access modifiers where it cannot be inherited by the methods. Sealed modifiers can also be applied to properties, events, and methods. This modifier cannot be applied to static members.

8.What are the various types of constructors?

A. There are three various types of constructors, and they are as follows:

– Default Constructor – With no parameters.

– Parametric Constructor – With Parameters. Create a new instance of a class and also passing arguments simultaneously.

– Copy Constructor – Which creates a new object as a copy of an existing object.

9.Whether static method can use nonstatic members?

A. False.

10.What is the default access specifier in a class definition?

A. Private access specifier is used in a class definition.