



# Advanced C++ Programming

Trainer

MOHAMMED SHOUKAT ALI



## INHERITANCE

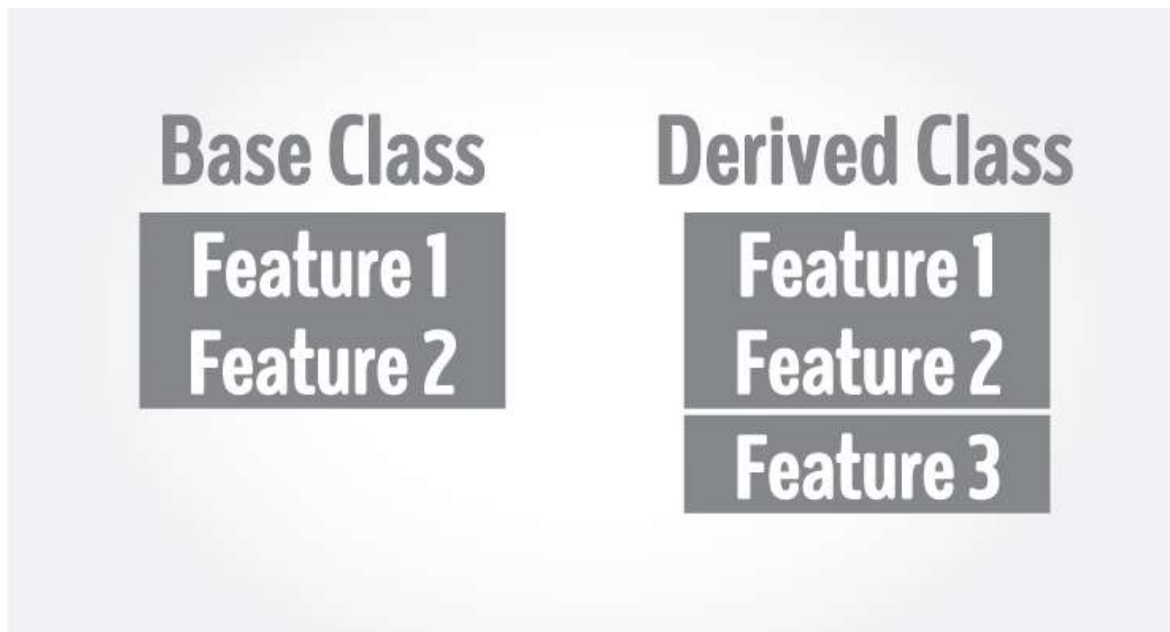
Inheritance is one of the key features of Object-oriented programming in C++. It allows user to create a new class (derived class) from an existing class (base class).

### Real life example of inheritance:

The real life example of inheritance is parents & child; all the properties of father are inherited by his son.



The derived class inherits all the features from the base class and can have additional features of its own.



## Why inheritance should be used?

Suppose, in our game, we want three characters - a **maths teacher**, a **footballer** and a **businessman**.

Since, all of the characters are persons, they can walk and talk. However, they also have some special skills. A maths teacher can **teach maths**, a footballer can **play football** and a businessman can **run a business**.

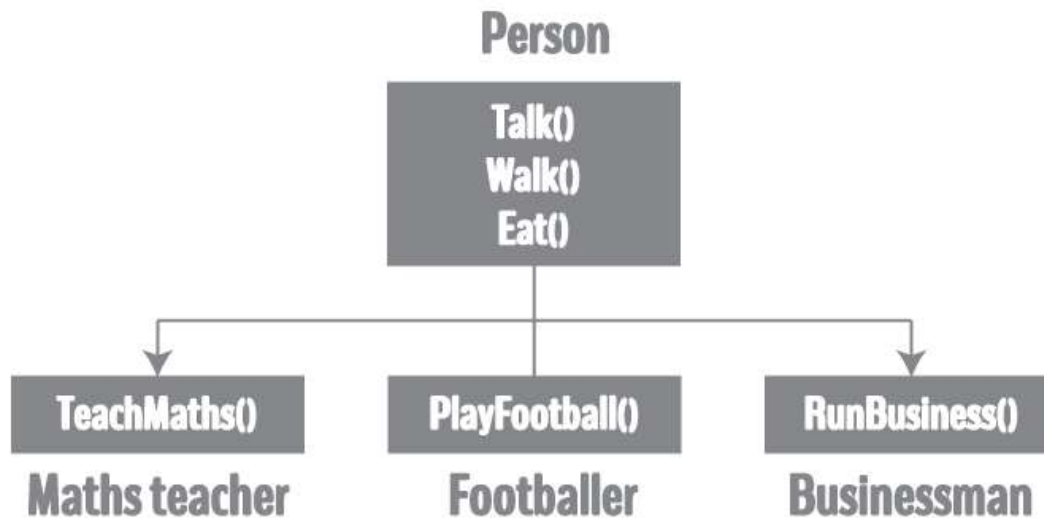
We can individually create three classes who can walk, talk and perform their special skill as shown in the figure below.



In each of the classes, we would be copying the same code for walk and talk for each character.

If we want to add a new feature - eat, we need to implement the same code for each character. This can easily become error prone (when copying) and duplicate codes.

It would be a lot easier if we had a **Person** class with basic features like talk, walk, eat, sleep, and add special skills to those features as per our characters. This is done using inheritance.



Using inheritance, now we don't implement the same code for walk and talk for each class. We just need to **inherit** them.

So, for Maths teacher (derived class), we inherit all features of a Person (base class) and add a new feature **TeachMaths**. Likewise, for a footballer, we inherit all the features of a Person and add a new feature **PlayFootball** and so on.

This makes our code cleaner, understandable and extendable.

**It is important to remember:** When working with inheritance, each derived class should satisfy the condition whether it **"is a"** base class or not. In the example above, Maths teacher **is a** Person, Footballer **is a** Person. We cannot have: Businessman **is a** Business.

### Advantages of inheritance

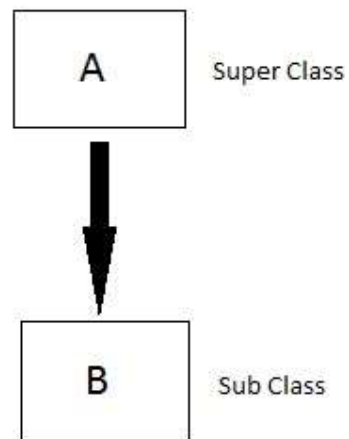
If we develop any application using concept of Inheritance than that application has following advantages,

1. Application development time is less.
2. Application takes less memory.
3. Application execution time is less.
4. Application performance is enhanced (improved).
5. Redundancy (repetition) of the code is reduced or minimized so that we get consistence results and less storage cost.

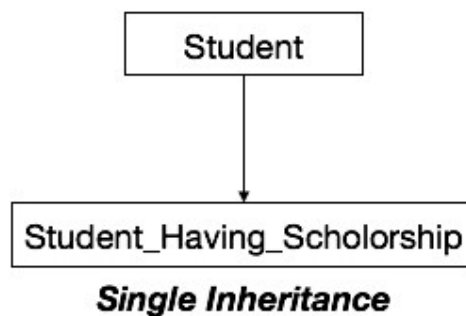
## TYPES OF INHERITANCE:

- 1) Single Inheritance
- 2) Multiple Inheritance
- 3) Multilevel Inheritance
- 4) Hierarchical Inheritance
- 5) Hybrid Inheritance

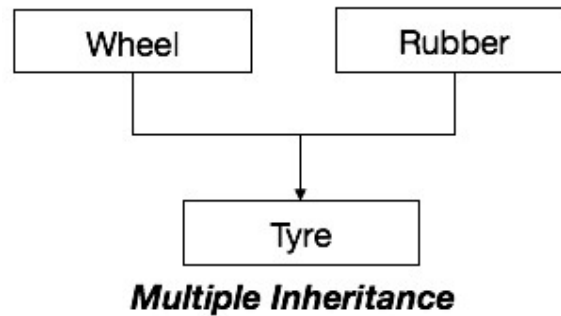
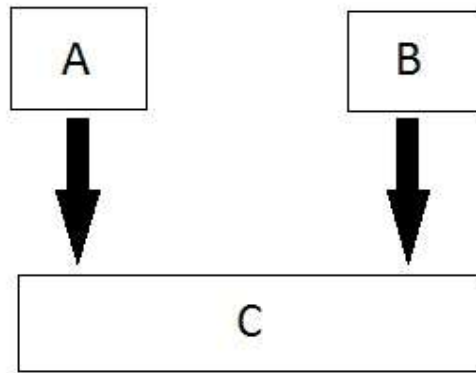
1) Single Inheritance: A class which acquires properties from a single base class is called as Single Inheritance.



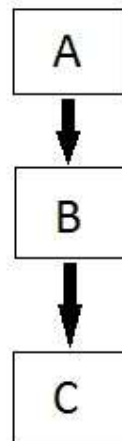
A subclass derives from a single super-class.



2) Multiple Inheritance: A class which acquires properties from more than one base class is called as multiple inheritance.

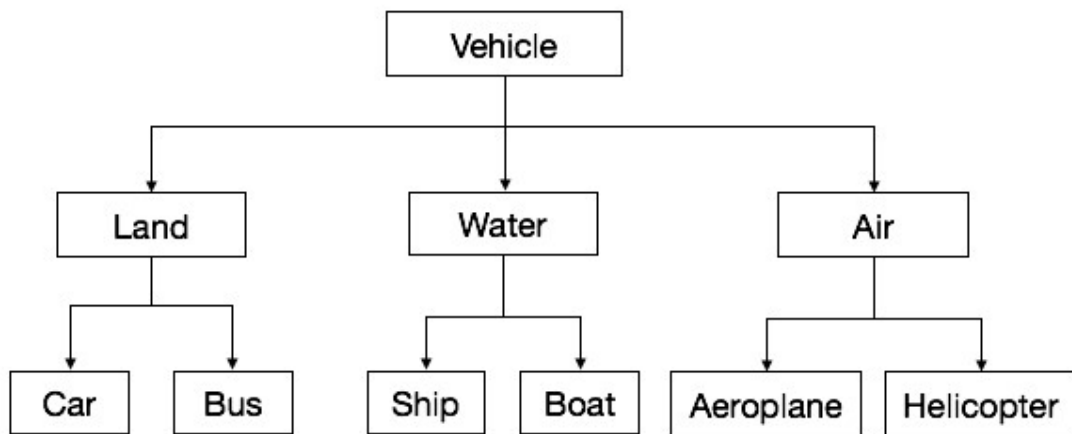
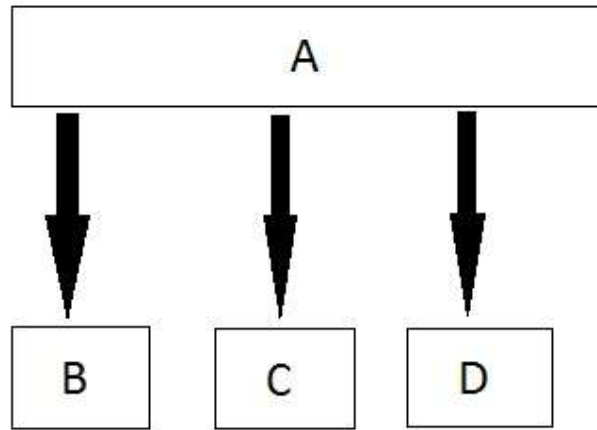


3) Multilevel Inheritance: The mechanism of deriving a class from another derived class is known as multilevel inheritance.



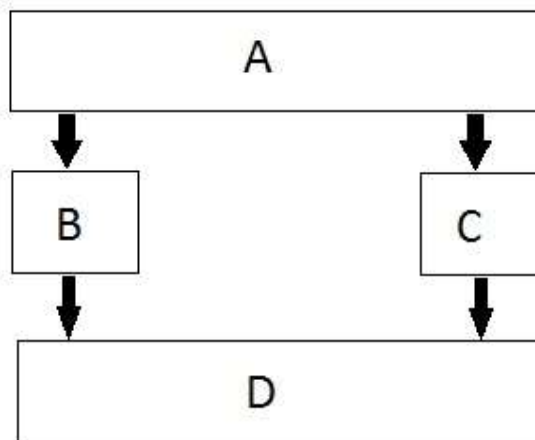
4) Hierarchical Inheritance:

A class which has a number of subclasses each of which may have subsequent subclasses, continuing for a number of levels, so as to form a tree structure.

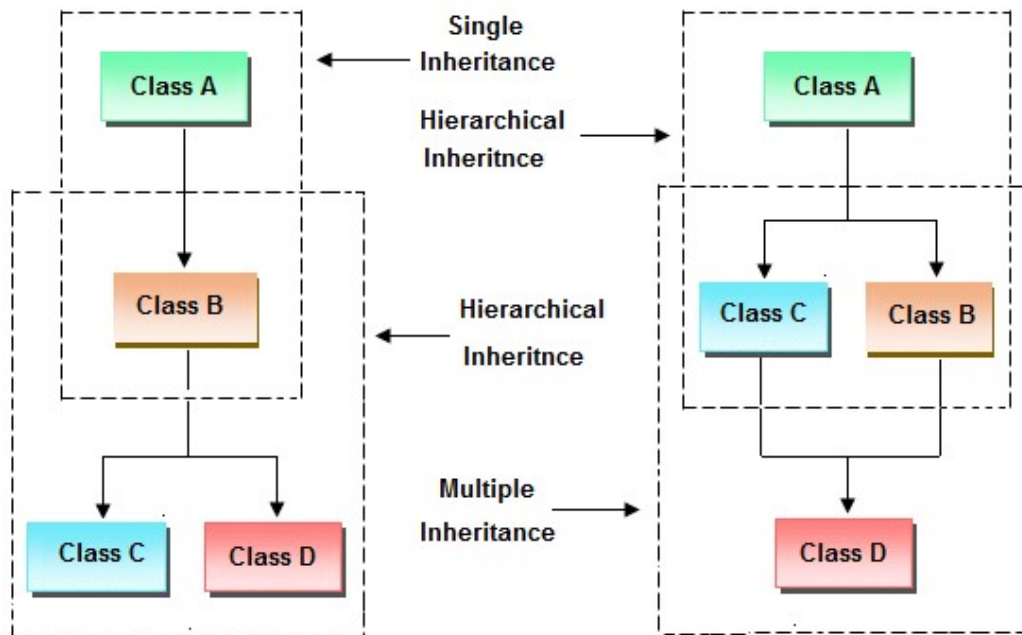


***Hierarchical Inheritance***

5) Hybrid Inheritance: Hybrid inheritance is a combination of multiple inheritance and multiple inheritance. A class is derived from two classes as in multiple inheritance. However, one of the parent classes is not a base class. It is a derived class.



Note: Combination of any inheritance type is Hybrid inheritance.



## Implementation of Inheritance in C++ Programming

```
class Person
```

```
{
```

```
... ..
```

```
};
```

```
class MathsTeacher : public Person
```

```
{
```

```
... ..
```

```
};
```

```
class Footballer : public Person
```

```
{
```

```
... ..
```

```
};
```



In the above example, class *Person* is a base class and classes *MathsTeacher* and *Footballer* are the derived from *Person*.

The derived class appears with the declaration of a class followed by a colon, the keyword *public* and the name of base class from which it is derived.

Since, *MathsTeacher* and *Footballer* are derived from *Person*, all data member and member function of *Person* can be accessible from them.

### **Example: Inheritance in C++ Programming**

Create game characters using the concept of inheritance.

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

class Person
{
    public:
        string profession;
        int age;

        Person(): profession("unemployed"), age(16) { }
        void display()
        {
            cout << "My profession is: " << profession << endl;
            cout << "My age is: " << age << endl;
            walk();
            talk();
        }
        void walk() { cout << "I can walk." << endl; }
        void talk() { cout << "I can talk." << endl; }
};
```

// MathsTeacher class is derived from base class Person.

```

class MathsTeacher : public Person
{
    public:
        void teachMaths() { cout << "I can teach Maths." << endl; }
};

// Footballer class is derived from base class Person.
class Footballer : public Person
{
    public:
        void playFootball() { cout << "I can play Football." << endl; }
};

int main()
{
    MathsTeacher teacher;
    teacher.profession = "Teacher";
    teacher.age = 23;
    teacher.display();
    teacher.teachMaths();

    Footballer footballer;
    footballer.profession = "Footballer";
    footballer.age = 19;
    footballer.display();
    footballer.playFootball();

    return 0;
}

```

## Output

My profession is: Teacher  
 My age is: 23

I can walk.  
I can talk.  
I can teach Maths.  
My profession is: Footballer  
My age is: 19  
I can walk.  
I can talk.  
I can play Football.

In this program, Person is a base class, while MathsTeacher and Footballer are derived from Person.

Person class has two data members - *profession* and *age*. It also has two member functions - *walk()* and *talk()*.

Both MathsTeacher and Footballer can access all data members and member functions of Person.

However, MathsTeacher and Footballer have their own member functions as well: *teachMaths()* and *playFootball()* respectively. These functions are only accessed by their own class.

In the *main()* function, a new MathsTeacher object *teacher* is created.

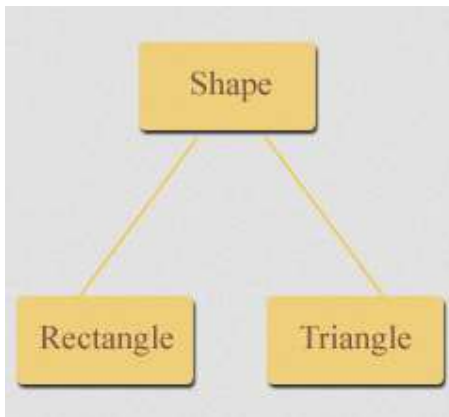
Since, it has access to Person's data members, *profession* and *age* of *teacher* is set. This data is displayed using the *display()* function defined in the Person class. Also, the *teachMaths()* function is called, defined in the MathsTeacher class.

Likewise, a new Footballer object *footballer* is also created. It has access to Person's data members as well, which is displayed by invoking the *display()* function. The *playFootball()* function only accessible by the footballer is called then after.

<https://www.programiz.com/cpp-programming/public-protected-private-inheritance>

<https://www.programiz.com/cpp-programming/function-overriding>

Following example further explains concept of inheritance:



```
class Shape
{
protected:
    float width, height;
public:
    void set_data (float a, float b)
    {
        width = a;
        height = b;
    }
};

class Rectangle: public Shape
{
public:
    float area ()
    {
        return (width * height);
    }
};
```

```

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

```

```

int main ()
{
    Rectangle rect;
    Triangle tri;
    rect.set_data (5,3);
    tri.set_data (2,5);
    cout << rect.area() << endl;
    cout << tri.area() << endl;
    return 0;
}

```

Output:

```

15
5

```

The object of the class Rectangle contains :

width, height inherited from Shape becomes the protected member of Rectangle.  
 set\_data() inherited from Shape becomes the public member of Rectangle  
 area is Rectangle's own public member

The object of the class Triangle contains:

width, height inherited from Shape becomes the protected member of Triangle.  
 set\_data() inherited from Shape becomes the public member of Triangle  
 area is Triangle's own public member

set\_data () and area() are public members of derived class and can be accessed from outside class i.e. from main()

<http://www.cppforschool.com/tutorial/overriding-inheritance.html>

## ACCESS SPECIFIERS:

C++ access specifier are:

1. private
2. public
3. protected

Access modifiers defines the scope of methods or variables. Scope means whether the methods or variable defined in class can be accessed by all class of project or the same class only or with few other class which is inherited.

C++ defines three access modifiers or access specifiers:

**1. Private:** Private member defines that the members or methods can be accessed within the same class only.

**2. Public:** Public member defines that the variable or methods can be accessed at anywhere within the project.

**3. Protected:** Protected member can be accessed to the class which is inherited by other class.

By default, all members and function of a class is private i.e if no access specifier is specified.

Syntax of declaring access modifiers in C++

```
class
{
    private:
        // private members and function
    public:
```

```

        // public members and function
protected:
        // protected members and function
};

```

### Example:

```

//program demo using access specifiers
#include<iostream>
using namespace std;
class Base
{
    private:
        int a;
    public:
        int b;
    protected:
        int c;

    public:
        Base() //constructor
        {
            a=10;
            b=20;
            c=30;
        }

        void display()
        {
            cout<<"Base class variable values"<<endl;
            //Every members can be accessed as all are in the same class
            cout<<"Value of a = "<<a<<endl;
            cout<<"Value of b = "<<b<<endl;
            cout<<"Value of c = "<<c<<endl;
        }
    }

```

```

    }
};

class Sub : public Base
{
    public:
    void display()
    {
        cout<<"\n Sub class variable values"<<endl;
        //As 'a' is private member so 'a' cannot be accessed by another class
        //cout<<"Value of a = "<<a<<endl;
        //'b' is declared as public, so it can be accessed from any class which is
        inherited
        cout<<"Value of b = "<<b<<endl;
        //'c' is declared as protected, so it can be accessed from class which is inherited
        cout<<"Value of c = "<<c<<endl;
    }
};

int main()
{
    Base objb;
    objb.display();

    Sub objj;
    objj.display();

    cout<<"\n Accessing variable of base outside base class"<<endl;
    //'a' cannot be accessed as it is private
    //cout<<"value of a = "<<objb.a<<endl;

    //'b' is public as can be accessed from any where

```



```
cout<<"value of b = "<<objb.b<<endl;

//'c' is protected and cannot be accessed here
//cout<<"value of c = "<<objb.c<<endl;
return 0;
}
```

Output:

Declare variable values

Value of a = 10

Value of b = 20

Value of c = 30

Inherit variable values

Value of b = 20

Value of c = 30

Accessing variable of declare outside declare class

value of b = 20

## Simple Program for Single Inheritance Using C++ Programming

To write a program to find out the payroll system using single inheritance.

### **ALGORITHM:**

Step 1: Start the program.

Step 2: Declare the base class emp.

Step 3: Define and declare the function getData1() to get the employee details.

Step 4: Declare the derived class salary.

Step 5: Declare and define the function getData2() to get the salary details.

Step 6: Define the function calculate() to find the net pay.

Step 7: Define the function display().

Step 8: Create the derived class object.

Step 9: Read the number of employees.

Step 10: Call the function getData1(),getData2() and calculate() to each employees.

Step 11: Call the display().

Step 12: Stop the program.

### **PROGRAM: PAYROLL SYSTEM USING SINGLE INHERITANCE**

```
#include<iostream>
```

```
using namespace std;
```

```
class emp
```

```
{
```

```
    public:
```

```
        int eno;
```

```
        char name[20], designation[20];
```

```
        void getData1()
```

```
    {
```

```

        cout<<"Enter the employee number:";
        cin>>eno;
        cout<<"Enter the employee name:";
        cin>>name;
        cout<<"Enter the designation:";
        cin>>designation;
    }
};

```

```

class salary:public emp

```

```

{
    float bp,hra,da,pf,np;
public:
    void getData2()
    {
        cout<<"Enter the basic pay:";
        cin>>bp;
        cout<<"Enter the Human Resource Allowance:";
        cin>>hra;
        cout<<"Enter the Dearness Allowance:";
        cin>>da;
        cout<<"Enter the Profitability Fund:";
        cin>>pf;
    }
    void calculate()
    {
        np = (bp + hra +da) - pf;
    }
    void display()
    {
        cout<<eno<<"\t"<<name<<"\t"<<designation<<"\t"<<bp<<"\t"<<hra<<"\t"<<da
        <<"\t"<<pf<<"\t"<<np<<"\n";
    }
}

```

```

    }
};
int main()
{
    int i,n;
    char ch;
    salary s[10];
    cout<<"Enter the number of employee:";
    cin>>n;
    for(i=0;i<n;i++)
    {
        s[i].getData1();
        s[i].getData2();
        s[i].calculate();
    }
    cout<<"\n e_no \t e_name\t designation \t bp \t hra \t da \t pf \t np \n";
    for(i=0;i<n;i++)
    {
        s[i].display();
    }
    return 0;
}

```

### **Program using Single Inheritance:**

```

#include<iostream>
using namespace std;
class person //Declare base class
{
    private:
        char name[30];
        int age;

```

```

        char address[50];
    public:
        void get_data ( );
        void put_data ( );
};

```

```

class student : private person    //Declare derived class
{
    private:
        int rollno;
        float marks;
    public:
        void get_stinfo ( );
        void put_stinfo ( );
};

```

```

void person::get_data ( )
{
    cout<<"Enter name:";
    cin>>name;
    cout<<"Enter age:";
    cin>>age;
    cout<<"Enter address:";
    cin>>address;
}

```

```

void person::put_data ( )
{

```

```

cout<<"Name: " <<name<<endl;
    cout<<"Age: " <<age<<endl;
    cout<<"Address: " <<address<<endl;
}

```

```

void student::get_stinfo ( )
{
    get_data ( );
    cout<<"Enter roll number:";
        cin>>rollno;
        cout<<"Enter marks:";
        cin>>marks;
}

```

```

void student::put_stinfo ( )
{
    put_data ( );
    cout<<"Roll Number:" <<rollno<<endl;
    cout<<"Marks:"<<marks<<endl;
}

```

```

int main ( )
{
    student st;
    st.get_stinfo ( );
    st.put_stinfo ( );
    return 0;
}

```

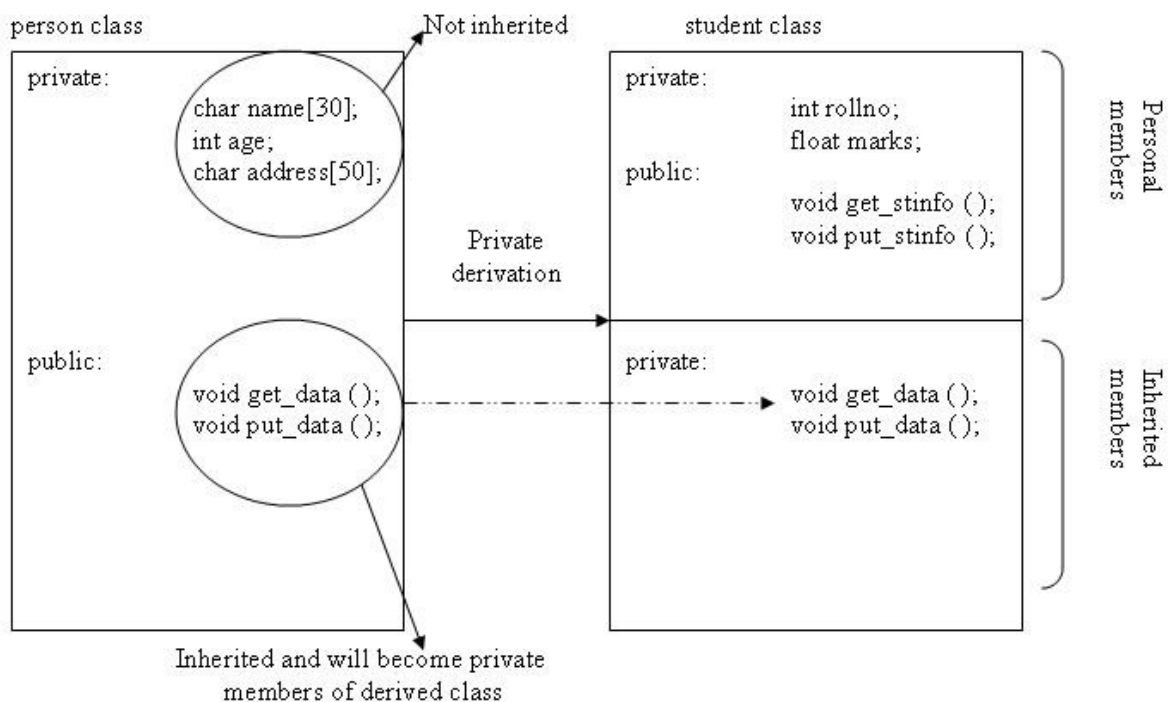
Output:

Enter name:Ricky

Enter age:25  
Enter address:KingsStreet  
Enter roll number:125  
Enter marks:80

Name: Ricky  
Age: 25  
Address: KingsStreet  
Roll Number:125  
Marks:80

Here 'person' is base class and 'student' is derived class that has been inherited from person privately. The private members of the 'person' class will not be inherited. The public members of the 'person' class will become private members of the 'student' class since the type of the derivation is private. Since the inherited members have become private, they cannot be accessed by the object of the 'student' class.



```
//program demo  
#include<iostream>
```

```

using namespace std;
class A
{
    private:
        int a;
    public:
        void display()
        {
            a=10;
            cout<<"This is from class A."<<endl;
            cout<<"The value of a = "<<a<<endl;
        }
};

class B:private A //inherit from A and everything of A becomes private in class B. If
you want to access private members of A from B, call it in public
{
    public:
        void show()
        {
            display();
            cout<<"This is from class B."<<endl;
        }
};

int main()
{
    B obj;
    obj.show();
}

```

### **MULTIPLE INHERITANCE:**

```

//program demo using multiple inheritance
#include<iostream>

```



```

using namespace std;
class A
{
    private:
        int a;
    public:
        void displayA()
        {
            a=10;
            cout<<"This is from class A."<<endl;
            cout<<"The value of a is = "<<a<<endl;
        }
};
class B
{
    private:
        int b;
    public:
        void displayB()
        {
            b=20;
            cout<<"This is from class B."<<endl;
            cout<<"The value of b is = "<<b<<endl;
        }
};
class C:public A,public B
{
    private:
        int c;
    public:
        void displayC()

```

```

        {
c=30;
cout<<"This is from class C."<<endl;
cout<<"The value of c is = "<<c<<endl;
        }
};
int main()
{
    C obj;
    obj.displayA();
    obj.displayB();
    obj.displayC();
}

```

### **Simple Program for Multiple Inheritance:**

To find out the student details using multiple inheritance.

#### **ALGORITHM:**

Step 1: Start the program.

Step 2: Declare the base class student.

Step 3: Declare and define the function get() to get the student details.

Step 4: Declare the other class sports.

Step 5: Declare and define the function getsm() to read the sports mark.

Step 6: Create the class statement derived from student and sports.

Step 7: Declare and define the function display() to find out the total and average.

Step 8: Declare the derived class object, call the functions get(), getsm() and display().

Step 9: Stop the program.

```
//program demo using Multiple inheritance
```

```
#include<iostream>
```

```
using namespace std;
```

```

class student
{
    protected:
        int rno,m1,m2;
    public:
        void get()
        {
            cout<<"Enter the Roll no :";
            cin>>rno;
            cout<<"Enter the two marks  :";
            cin>>m1>>m2;
        }
};

class sports
{
    protected:
        int sm;           // sm = Sports mark
    public:
        void getsn()
        {
            cout<<"\nEnter the sports mark :";
            cin>>sm;
        }
};

class statement: public student, public sports
{
    int tot,avg;
    public:
    void display()
    {
        tot=(m1+m2+sm);
    }
};

```

```

        avg=tot/3;
        cout<<"\n\n\tRoll No    : "<<rno<<"\n\tTotal    : "<<tot;
        cout<<"\n\tAverage    : "<<avg;
    }
};

int main()
{
    statement obj;
    obj.get();
    obj.getsm();
    obj.display();
    return 0;
}

```

### **Output:**

Declare variable values

Value of a = 10

Value of b = 20

Value of c = 30

Inherit variable values

Value of b = 20

Value of c = 30

Accessing variable of declare outside declare class

value of b = 20

### **MULTI LEVEL INHERITANCE:**

#### **Syntax of Multilevel Inheritance**

```

class base_classname
{

```

```

    properties;
    methods;
};

class intermediate_classname:visibility_mode base_classname
{
    properties;
    methods;
};

class child_classname:visibility_mode intermediate_classname
{
    properties;
    methods;
};

```

//program demo using Multi Level inheritance

```

#include<iostream>
using namespace std;
class person
{
    char name[100], gender[10];
    int age;
public:
    void getdata()
    {
        cout<<"Name: ";
        fflush(stdin);    /*clears input stream*/
        gets(name);
        cout<<"Age: ";
        cin>>age;
        cout<<"Gender: ";
        cin>>gender;
    }
    void display()
    {

```

```

        cout<<"Name: "<<name<<endl;
        cout<<"Age: "<<age<<endl;
        cout<<"Gender: "<<gender<<endl;
    }
};

class employee: public person
{
    char company[100];
    float salary;
public:
    void getdata()
    {
        person::getdata();
        cout<<"Name of Company: ";
        fflush(stdin);
        gets(company);
        cout<<"Salary: $.";
        cin>>salary;
    }
    void display()
    {
        person::display();
        cout<<"Name of Company: "<<company<<endl;
        cout<<"Salary: $."<<salary<<endl;
    }
};

class programmer: public employee
{
    int number;
public:

```

```

void getdata()
{
    employee::getdata();
    cout<<"Number of programming language known: ";
    cin>>number;
}
void display()
{
    employee::display();
    cout<<"Number of programming language known: "<<number;
}
};

int main()
{
    programmer p;
    cout<<"Enter data"<<endl;
    p.getdata();
    cout<<endl<<"Displaying data"<<endl;
    p.display();
    return 0;
}

```

OUTPUT:

Enter data

Name: Ali

Age: 30

Gender: Male

Name of Company: Sai Pali

Salary: Rs.45000

Number of programming language known: 4

Displaying data

Name: Ali

Age: 30

Gender: Male

Name of Company: Sai Pali

Salary: Rs.45000

Number of programming language known: 4

This program is an example of multiple inheritance. Here, programmer class is derived from employee which is derived from person. Each class has required attributes and methods. The public features of person is inherited by employee and the public features of employee is inherited by programmer. The method *getdata()* asks user to input data, while *display()* displays the data.

## **HIERARCHICAL INHERITANCE:**

Inheritance is the process of inheriting properties of objects of one class by objects of another class. The class which inherits the properties of another class is called Derived or Child or Sub class and the class whose properties are inherited is called Base or Parent or Super class.

When more than one classes are derived from a single base class, such inheritance is known as Hierarchical Inheritance.

### **Syntax of Hierarchical Inheritance**

```
class base_classname
{
    properties;
    methods;
};

class derived_class1:visibility_mode base_classname
{
    properties;
```



```

    methods;
};

class derived_class2:visibility_mode base_classname
{
    properties;
    methods;
};
... ..
... ..
class derived_classN:visibility_mode base_classname
{
    properties;
    methods;
};

```

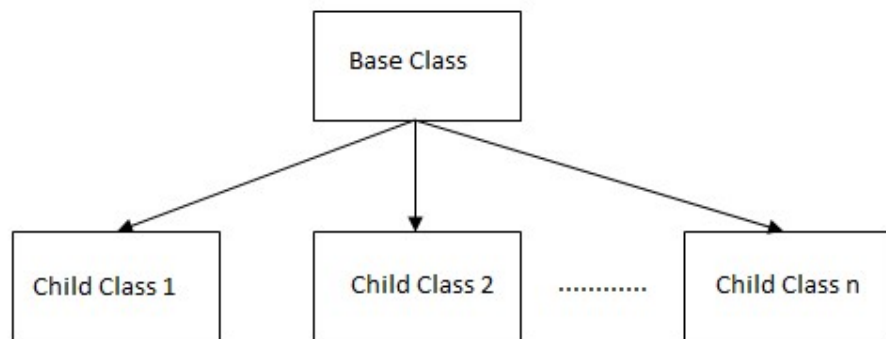


Fig: Hierarchical Inheritance

//program demo using Hierarchical inheritance

```
#include<iostream>
```

```
using namespace std;
```

```
class person
```

```
{
```

```
    char name[100], gender[10];
```

```
    int age;
```

```
    public:
```

```
        void getdata()
```

```

{
    cout<<"Name: ";
    fflush(stdin); /*clears input stream*/
    gets(name);
    cout<<"Age: ";
    cin>>age;
    cout<<"Gender: ";
    cin>>gender;
}

void display()
{
    cout<<"Name: "<<name<<endl;
    cout<<"Age: "<<age<<endl;
    cout<<"Gender: "<<gender<<endl;
}

};

```

```

class student: public person
{
    char institute[100], level[20];
    public:
        void getdata()
        {
            person::getdata();
            cout<<"Name of College/School: ";

```

```

        fflush(stdin);

        gets(institute);

        cout<<"Level: ";

        cin>>level;

    }

    void display()

    {

        person::display();

        cout<<"Name of College/School: "<<institute<<endl;

        cout<<"Level: "<<level<<endl;

    }

};

class employee: public person

{

    char company[100];

    float salary;

public:

    void getdata()

    {

        person::getdata();

        cout<<"Name of Company: ";

        fflush(stdin);

        gets(company);

        cout<<"Salary: $.";

        cin>>salary;

```

```

    }

    void display()
    {
        person::display();

        cout<<"Name of Company: "<<company<<endl;
        cout<<"Salary: $."<<salary<<endl;
    }
};

```

```

int main()
{
    student s;
    employee e;

    cout<<"Student"<<endl;
    cout<<"Enter data"<<endl;
    s.getdata();
    cout<<endl<<"Displaying data"<<endl;
    s.display();

    cout<<endl<<"Employee"<<endl;
    cout<<"Enter data"<<endl;
    e.getdata();
    cout<<endl<<"Displaying data"<<endl;
    e.display();

    return 0;
}

```

## **OUTPUT:**

Student

Enter data

Name: Shaun Marsh

Age: 30

Gender: Male

Name of College/School: Sai Pali

Level: 4

Displaying data

Name: Shaun Marsh

Age: 30

Gender: Male

Name of College/School: Sai Pali

Level: 4

Employee

Enter data

Name: Mark Taylor

Age: 24

Gender: Male

Name of Company: Microsoft Corp.

Salary: Rs.58000

Displaying data

Name: Mark Taylor

Age: 24

Gender: Male

Name of Company: Microsoft Corp.

Salary: Rs.58000

In this program, *student* and *employee* classes are derived from *person*. Person has two public methods: *getdata()* and *display()*. These methods are inherited by both *student* and *employee*. Input is given using *getdata()* method and displayed using *display()* method. This is an example of hierarchical inheritance since two classes are derived from a single class.

### **HYBRID INHERITANCE:**

Hybrid Inheritance is a method where one or more types of inheritance are combined together and used.

// program demo using Hybrid inheritance

```
#include<iostream>
```

```
using namespace std;
```

```
int a,b,c,d,e;
```

```
class A
```

```
{
```

```
protected:
```

```
public:
```

```
void getab()
```

```
{
```

```
cout<<"\n Enter a and b value:";
```

```
cin>>a>>b;
```

```
}
```

```

};

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

class C
{
    protected:
    public:
    void getd()
    {
        cout<<"\n 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:30

Enter d value:40

Addition is :100



## Virtual Functions

Virtual Function is a function in base class, which is overridden in the derived class, and which tells the compiler to perform **Late Binding** on this function.

**Virtual** Keyword is used to make a member function of the base class Virtual.

### Late Binding

In Late Binding function call is resolved at runtime. Hence, now compiler determines the type of object at runtime, and then binds the function call. Late Binding is also called Dynamic Binding or Runtime Binding.

```
#include <iostream>
using namespace std;
class A
{
    public:
    virtual void display()
    {
        cout << "Base class is invoked"<<endl;
    }
};
class B: public A
{
    public:
    void display()
    {
        cout << "Derived Class is invoked"<<endl;
    }
};
int main()
{
```

```
A* a;    //pointer of base class
B b;     //object of derived class
a = &b;
a->display(); //Late Binding occurs
}
```

Output:

Derived Class is invoked

### Example 2:

```
#include<iostream>
using namespace std;
class A
{
    public:
        void display()
        {
            cout<<"From A display()..."<<endl;
        }
        virtual void show()
        {
            cout<<"From A show()..."<<endl;
        }
};
class B:public A
{
    public:
        void display()
        {
            cout<<"From B display()..."<<endl;
        }
}
```

```

        void show()
        {
            cout<<"From B show()..."<<endl;
        }
};

int main()
{
    A *ptr;
    A obja;
    B objb;
    ptr=&obja;
    ptr->display();
    ptr->show();

    ptr=&objb;
    ptr->display();
    ptr->show();
    return 0;
}

```

### **Output:**

```

From A display()...
From A show()...
From A display()...
From B show()...

```

### **Pure Virtual Functions**

Pure virtual Functions are virtual functions with no definition. They start with virtual keyword and ends with = 0. Here is the syntax for a pure virtual function,

```
virtual void f() = 0;
```

## Pure Virtual definitions

Pure Virtual functions can be given a small definition in the Abstract class, which you want all the derived classes to have. Still, you cannot create object of Abstract class.

Also, the Pure Virtual function must be defined outside the class definition. If you will define it inside the class definition, compiler will give an error. Inline pure virtual definition is Illegal.

```
class Base          //Abstract base class
{
    public:
    virtual void show() = 0;          //Pure Virtual Function
};

void Base :: show()          //Pure Virtual definition
{
    cout << "Pure Virtual definition\n";
}

class Derived:public Base
{
    public:
    void show()
    { cout << "Implementation of Virtual Function in Derived class"; }
};

int main()
{
```

```
Base *b;  
  
Derived d;  
  
b = &d;  
  
b->show();  
  
}
```

Output :

Pure Virtual definition

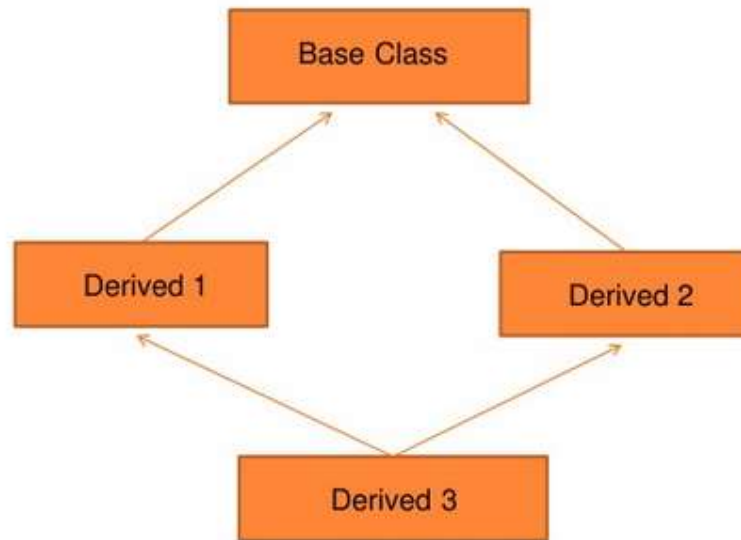
Implementation of Virtual Function in Derived class

[https://www.slideshare.net/Tech\\_MX/virtual-base-class](https://www.slideshare.net/Tech_MX/virtual-base-class)

<http://www.cpp.thiyagaraaj.com/c-programs/simple-program-for-virtual-base-class-using-c-programming>

## VIRTUAL BASE CLASS:

## DIAMOND PROBLEM



An ambiguity can arise when several paths exist to a class from the same base class. This means that a child class could have duplicate sets of members inherited from a single base class.

C++ solves this issue by introducing a virtual base class. When a class is made virtual, necessary care is taken so that the duplication is avoided regardless of the number of paths that exist to the child class.

//program demo for diamond problem

```
#include<iostream>
```

```
using namespace std;
```

```
class one
```

```
{
```

```
protected:
```

```
    int a1;
```

```
public:
```

```
    one()
```

```
    {
```

```
        cout<<"One..."<<endl;
```

```
    }
```

```
};
```

```

class two: public one
{
    protected:
        int a2;
    public:
        two()
        {
            cout<<"Two..."<<endl;
        }
};

class three: public one
{
    protected:
        int a3;
    public:
        three()
        {
            cout<<"Three..."<<endl;
        }
};

class four : public two, public three
{
    private:
        int a4;
    public:
        four()
        {
            a4=6;
            cout<<"Four..."<<endl;
        }
};

int main()
{
    four obj;

```

```

    return 0;
}

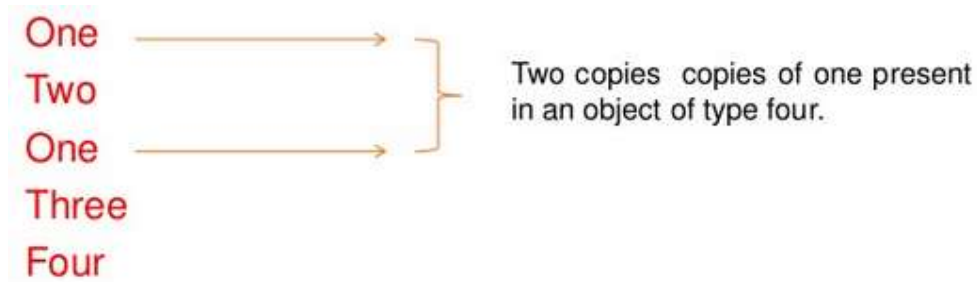
```

OUTPUT:

```

One...
Two...
One...
Three...
Four...

```



Ambiguous because variable 'a1' is present twice in the object of four, one in two class and one in three class.

Because there are two copies of 'a1' present in object 'obj', the compiler doesn't know which one is being referred.

### Manual Selection:

Manually select the required variable 'a1' by using the scope resolution operator.

```

int main()
{
    four obj;
    obj.two::a1=50;
    obj.three::a1=60;
    cout<<obj.two::a1<<endl;
    cout<<obj.three::a1<<endl;
    return 0;
}

```

obj.two::a1=50 and obj.three::a1=60



In the above statements, use of scope resolution operators resolves the problem of ambiguity. But this is not an efficient way because still two copies of 'a1' is available in object 'obj'.

How to make only one copy of 'a1' available in the object 'obj' which consumes less memory and easy to access without using the scope resolution operator.

This can be done by using **virtual base class**.

```
//program demo using virtual base class
```

```
#include<iostream>
```

```
using namespace std;
```

```
class one
```

```
{
```

```
protected:
```

```
    int a1;
```

```
public:
```

```
    one()
```

```
    {
```

```
        cout<<"One..."<<endl;
```

```
    }
```

```
};
```

```
class two: virtual public one
```

```
{
```

```
protected:
```

```
    int a2;
```

```
public:
```

```
    two()
```

```
    {
```

```
        cout<<"Two..."<<endl;
```

```
    }
```

```
};
```

```
class three: virtual public one
```

```
{
```

```
protected:
```

```
    int a3;
```

```

public:
    three()
    {
        cout<<"Three..."<<endl;
    }
};

class four: public two, public three
{
    private:
        int a4;
    public:
        four()
        {
            a4=6;
            cout<<"Four..."<<endl;
        }
};

int main()
{
    four obj;
    return 0;
}

```

OUTPUT:

One...                      Only one copy is maintained. **No ambiguity.**  
Two...  
Three...  
Four...

```

int main()
{
    four obj;
}

```

```

    obj.a1=50;    //Unambiguous since only one copy of 'a1' is present in object 'obj'
    cout<<obj.a1;
    return 0;
}

```

Now that both two and three have inherited base as virtual, any multiple inheritance involving them will cause only one copy of base to be present.  
Therefore, in four, there is only copy of base.

### **Program for Virtual Base Class Using C++ Programming**

To calculate the total mark of a student using the concept of virtual base class.

#### **ALGORITHM:**

Step 1: Start the program.

Step 2: Declare the base class student.

Step 3: Declare and define the functions getnumber() and putnumber().

Step 4: Create the derived class test virtually derived from the base class student.

Step 5: Declare and define the function getmarks() and putmarks().

Step 6: Create the derived class sports virtually derived from the base class student.

Step 7: Declare and define the function getscore() and putscore().

Step 8: Create the derived class result derived from the class test and sports.

Step 9: Declare and define the function display() to calculate the total.

Step 10: Create the derived class object obj.

Step 11: Call the function get number(),getmarks(),getscore() and display().

Step 12: Stop the program.

```

#include<iostream>
Using namespace std;
class student
{
    int rno;
    public:

```

```

void getnumber()
{
    cout<<"Enter Roll No:";
    cin>>rno;
}
void putnumber()
{
    cout<<"\n\n\tRoll No:"<<rno<<"\n";
}
};

```

```

class test:virtual public student
{

public:
    int part1,part2;
    void getmarks()
    {
        cout<<"Enter Marks\n";
        cout<<"Part1:";
        cin>>part1;
        cout<<"Part2:";
        cin>>part2;
    }
    void putmarks()
    {
        cout<<"\tMarks Obtained\n";
        cout<<"\n\tPart1:"<<part1;
        cout<<"\n\tPart2:"<<part2;
    }
};

```

```

class sports:public virtual student
{

public:
    int score;
    void getscore()
    {
        cout<<"Enter Sports Score:";
        cin>>score;
    }
    void putscore()
    {
        cout<<"\n\tSports Score is:"<<score;
    }
};

```

```

class result:public test,public sports
{
    int total;
public:
    void display()
    {
        total=part1+part2+score;
        putnumber();
        putmarks();
        putscore();
        cout<<"\n\tTotal Score:"<<total;
    }
};

```

```
void main()
{
    result obj;
    clrscr();
    obj.getnumber();
    obj.getmarks();
    obj.getscore();
    obj.display();
    getch();
}
```

**Output:**

Enter Roll No: 200

Enter Marks

Part1: 90

Part2: 80

Enter Sports Score: 80

Roll No: 200

Marks Obtained

Part1: 90

Part2: 80

Sports Score is: 80

Total Score is: 250

## C++ Abstract class and Pure Virtual Function

The goal of object-oriented programming is to divide a complex problem into small sets. This helps understand and work with problem in an efficient way.

Sometimes, it's desirable to use inheritance just for the case of better visualization of the problem.

In C++, we can create an abstract class that cannot be instantiated (we cannot create object of that class). However, we can derive a class from it and instantiate object of the derived class.

Abstract classes are the base class which cannot be instantiated.

**Note: We cannot create an object of abstract class.**

A class containing pure virtual function is known as abstract class.

### Pure Virtual Function

A virtual function whose declaration ends with `= 0` is called a pure virtual function. For example,

```
class Weapon
{
    public:
        virtual void features() = 0;
};
```

Here, the pure virtual function is

```
virtual void features() = 0
```

And, the class *Weapon* is an abstract class.

### Example: Abstract Class and Pure Virtual Function

```
//program demo using abstract class
```

```

#include<iostream>

using namespace std;

class Base    // Abstract class

{

    public:

    virtual void display()=0;

};

class Sub:public Base

{

    public:

        void display()

        {

            cout<<"This is from Sub class.."<<endl;

        }

};

int main()

{

    Sub obj;

    // Base objb; ERROR

    obj.display();

    return 0;

```



```
}
```

### **Output:**

This is from Sub class..

### **OR**

```
int main()
{
Base *ptr;    // We can make pointer of base class
Sub obj;
ptr=&obj;
ptr->display();
return 0;
}
```

### **Example: Abstract Class and Pure Virtual Function**

```
#include<iostream>
using namespace std;
// Abstract class
class Shape
{
protected:
    float l;
public:
    void getData()
    {
        cin >> l;
    }

    // virtual Function
    virtual float calculateArea() = 0;
};
```

```

class Square : public Shape
{
    public:
        float calculateArea()
        {   return l*l;  }
};

```

```

class Circle : public Shape
{
    public:
        float calculateArea()
        {   return 3.14*l*l;  }
};

```

```

int main()
{
    Square s;
    Circle c;

    cout << "Enter length to calculate the area of a square: ";
    s.getData();
    cout<<"Area of square: " << s.calculateArea();
    cout<<"\nEnter radius to calculate the area of a circle: ";
    c.getData();
    cout << "Area of circle: " << c.calculateArea();

    return 0;
}

```

OUTPUT:

```

Enter length to calculate the area of a square: 10
Area of square: 100
Enter radius to calculate the area of a circle: 4
Area of circle: 50.24

```

In this program, pure virtual function `virtual float area() = 0;` is defined inside the *Shape* class.

One important thing to note is that, you should override the pure virtual function of the base class in the derived class. If you fail to override it, the derived class will become an abstract class as well.

### **NESTING OF CLASSES:**

//program demo using nesting of classes

```
#include<iostream>
using namespace std;
class alpha
{
    public:
        alpha()
        {
            cout<<"From alpha..."<<endl;
        }
};
class beta
{
    public:
        beta()
        {
            cout<<"From beta..."<<endl;
        }
};
class gamma
{
    public:
        alpha obja;
        beta objb;
        gamma()
        {
```

```
        cout<<"From gamma..."<<endl;
    }
};

int main()
{
    gamma obj;
    return 0;
}
```

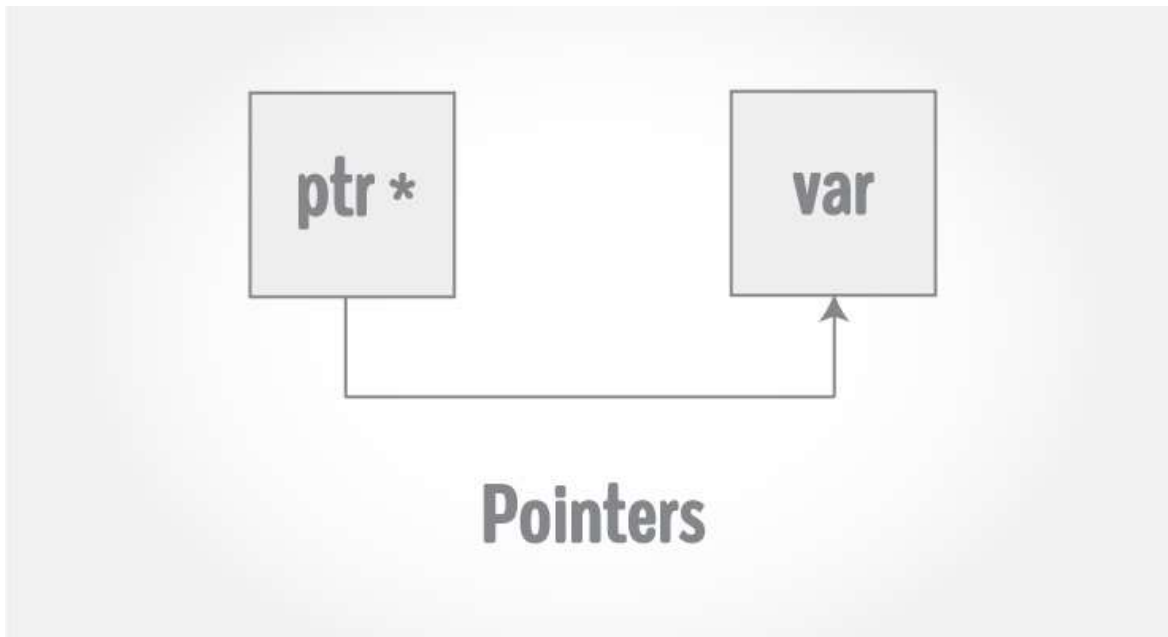
**OUTPUT:**

From alpha...

From beta...

From gamma...

## POINTERS:



Some C++ tasks are performed more easily with pointers, and other C++ tasks, such as dynamic memory allocation, cannot be performed without them.

Pointers are used in C++ program to access the memory and manipulate the address.

C++ gives us the power to manipulate the data in the computer's memory directly. We can assign and de-assign any space in the memory as we wish. This is done using Pointer variables.

Pointers variables are variables that points to a specific address in the memory pointed by another variable.

A **pointer** is a variable whose value is the address of another variable. Like any variable or constant, we must declare a pointer before we can work with it. The general form of a pointer **variable declaration is:**

```
type *var-name;
```

Here, type is the pointer's base type; it must be a valid C++ type and var-name is the name of the pointer variable. The asterisk we used to declare a pointer is the same asterisk that we use for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Following are the valid pointer declaration:

```
int *p;  
    OR  
int* p;
```

The statement above defines a pointer variable  $p$ . It holds the memory address

The asterisk is a dereference operator which means **pointer to**.

Here, pointer  $p$  is a **pointer to** int, i.e., it is pointing to an integer value in the memory address.

```
int   *ip;  // pointer to an integer  
double *dp; // pointer to a double  
float *fp;  // pointer to a float  
char  *ch   // pointer to a character
```

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

### Address in C++

To understand pointers, we should first know how data is stored on the computer.

Each variable you create in your program is assigned a location in the computer's memory. The value the variable stores is actually stored in the location assigned.

To know where the data is stored, C++ has an & operator. The **&** (reference) operator gives you the address occupied by a variable.

If var is a variable then, &var gives the address of that variable.

### Example 1: Program to read the address of the variable in C++

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

```

int main()
{
    int var1 = 3;
    int var2 = 24;
    int var3 = 17;
    cout << &var1 << endl;
    cout << &var2 << endl;
    cout << &var3 << endl;
    return 0;
}

```

## Output

```

0x7fff5fbff8ac
0x7fff5fbff8a8
0x7fff5fbff8a4

```

**Note:** You may not get the same result on your system.

The **0x** in the beginning represents the address is in hexadecimal form.

Notice that first address differs from second by 4-bytes and second address differs from third by 4-bytes.

This is because the size of integer (variable of type int) is 4 bytes in 64-bit system.

As you know every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator which denotes an address in memory. Consider the following which will print the address of the variables defined:

```

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

```

```

int var1;
char var2[10];

cout << "Address of var1 variable: ";
cout << &var1 << endl;

cout << "Address of var2 variable: ";
cout << &var2 << endl;

return 0;
}

```

When the above code is compiled and executed, it produces result something as follows:

```

Address of var1 variable: 0xbfefd5c0
Address of var2 variable: 0xbfefd5b6

```

### Using Pointers in C++:

There are few important operations, which we will do with the pointers very frequently. **(a)** we define a pointer variables **(b)** assign the address of a variable to a pointer and **(c)** finally access the value at the address available in the pointer variable. This is done by using unary operator **\*** that returns the value of the variable located at the address specified by its operand. Following example makes use of these operations:

```

//program demo using pointers

#include <iostream>
using namespace std;
int main () {
    int var = 20; // actual variable declaration.
    int *ip;      // pointer variable

    ip = &var;    // store address of var in pointer variable
}

```



```

cout << "Value of var variable: " << var << endl;

// print the address stored in ip pointer variable
cout << "Address stored in ip variable: " << ip << endl;

// access the value at the address available in pointer
cout << "Value of *ip variable: " << *ip << endl;
return 0;
}

```

When the above code is compiled and executed, it produces result something as follows:

```

Value of var variable: 20
Address stored in ip variable: 0xbfc601ac
Value of *ip variable: 20

```

## **C++ References vs Pointers**

References are often confused with pointers but three major differences between references and pointers are:

- You cannot have NULL references. You must always be able to assume that a reference is connected to a legitimate piece of storage.
- Once a reference is initialized to an object, it cannot be changed to refer to another object. Pointers can be pointed to another object at any time.
- A reference must be initialized when it is created. Pointers can be initialized at any time.

## **Creating References in C++**

Think of a variable name as a label attached to the variable's location in memory. You can then think of a reference as a second label attached to that memory location. Therefore, you can access the contents of the variable through either the original variable name or the reference. For example, suppose we have the following example:

```
int i = 17;
```

We can declare reference variables for i as follows.

```
int& r = i;
```

Read the & in these declarations as **reference**. Thus, read the first declaration as "r is an integer reference initialized to i" and read the second declaration as "s is a double reference initialized to d.". Following example makes use of references on int and double:

```
#include <iostream>

using namespace std;

int main () {
    // declare simple variables
    int i;
    double d;

    // declare reference variables
    int& r = i;
    double& s = d;

    i = 5;
    cout << "Value of i : " << i << endl;
    cout << "Value of i reference : " << r << endl;

    d = 11.7;
    cout << "Value of d : " << d << endl;
    cout << "Value of d reference : " << s << endl;

    return 0;
}
```

When the above code is compiled together and executed, it produces the following result:

Value of i : 5

Value of i reference : 5

Value of d : 11.7

Value of d reference : 11.7

## Reference operator (&) and Deference operator (\*)

Reference operator (&) gives the address of a variable.

To get the value stored in the memory address, we use the dereference operator (\*).

**For example:** If a *number* variable is stored in the memory address **0x123**, and it contains a value **5**.

The **reference (&)** operator gives the value **0x123**, while the **dereference (\*)** operator gives the value **5**.

**Note:** The (\*) sign used in the declaration of C++ pointer is not the dereference pointer. It is just a similar notation that creates a pointer.

### **C++ Program to demonstrate the working of pointer.**

```
#include <iostream>
using namespace std;
int main() {
    int *pc, c;
    c = 5;
    cout << "Address of c (&c): " << &c << endl;
    cout << "Value of c (c): " << c << endl << endl;

    pc = &c; // Pointer pc holds the memory address of variable c
    cout << "Address that pointer pc holds (pc): " << pc << endl;
    cout << "Content of the address pointer pc holds (*pc): " << *pc << endl <<
endl;

    c = 11; // The content inside memory address &c is changed from 5 to 11.
    cout << "Address pointer pc holds (pc): " << pc << endl;
    cout << "Content of the address pointer pc holds (*pc): " << *pc << endl <<
endl;

    *pc = 2;
    cout << "Address of c (&c): " << &c << endl;
    cout << "Value of c (c): " << c << endl << endl;
```

```

    return 0;
}

```

## Output

Address of c (&c): 0x7fff5fbff80c

Value of c (c): 5

Address that pointer pc holds (pc): 0x7fff5fbff80c

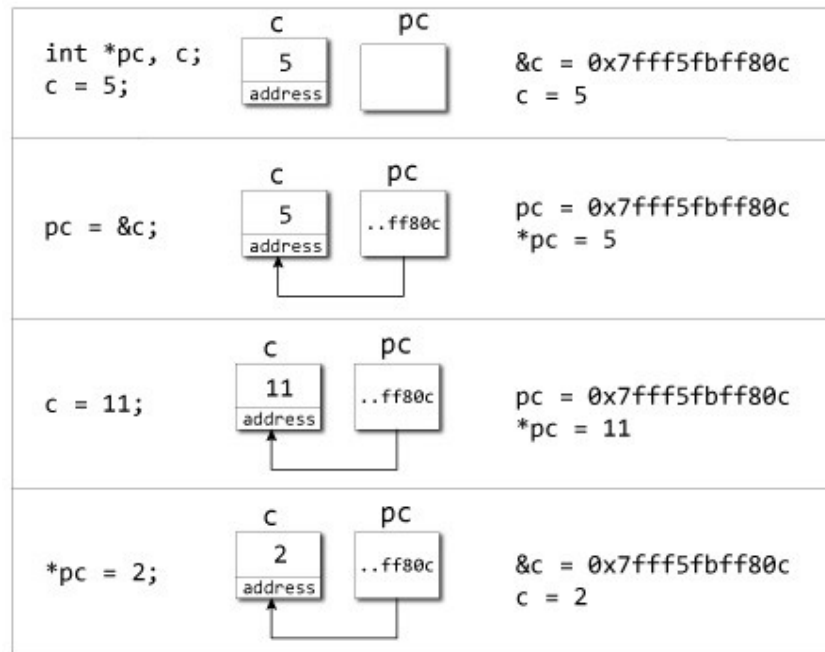
Content of the address pointer pc holds (\*pc): 5

Address pointer pc holds (pc): 0x7fff5fbff80c

Content of the address pointer pc holds (\*pc): 11

Address of c (&c): 0x7fff5fbff80c

Value of c (c): 2



## Explanation of program

- When c = 5; the value 5 is stored in the address of variable c - 0x7fff5fbff80c.

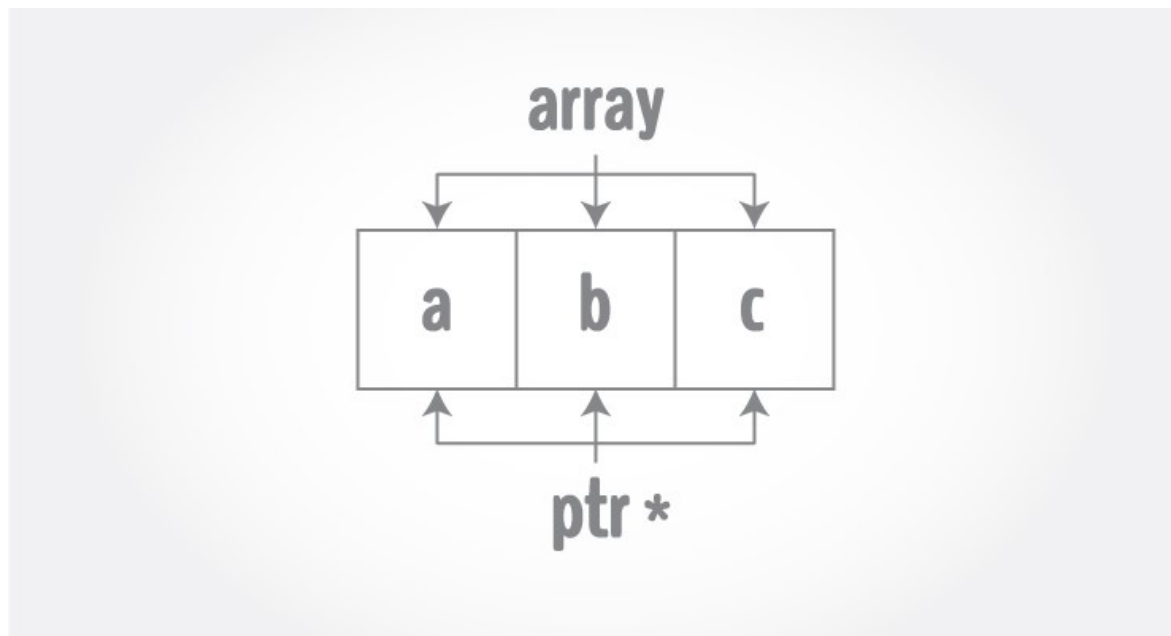
- When `pc = &c`; the pointer `pc` holds the address of `c` - `0x7fff5fbff8c`, and the expression (dereference operator) `*pc` outputs the value stored in that address, 5.
- When `c = 11`; since the address pointer `pc` holds is the same as `c` - `0x7fff5fbff8c`, change in the value of `c` is also reflected when the expression `*pc` is executed, which now outputs 11.
- When `*pc = 2`; it changes the content of the address stored by `pc` - `0x7fff5fbff8c`. This is changed from 11 to 2. So, when we print the value of `c`, the value is 2 as well.

### Common mistakes when working with pointers

Suppose, you want pointer `pc` to point to the address of `c`. Then,

```
int c, *pc;
pc=c; /* Wrong! pc is address whereas, c is not an address. */
*pc=&c; /* Wrong! *pc is the value pointed by address whereas, &c is an address. */
pc=&c; /* Correct */
*pc=c; /* Correct! *pc is the value pointed by address and, c is also a value. */
```

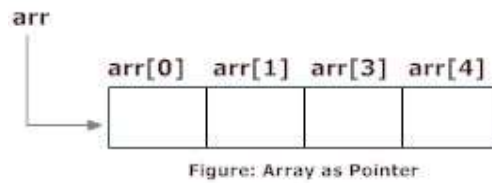
### C++ Pointers and Arrays



Pointers are the variables that hold address. Not only can pointers store address of a single variable, it can also store address of cells of an array.

Consider this example:

```
int* ptr;
int a[5];
ptr = &a[2]; // &a[2] is the address of third element of a[5].
```



Suppose, pointer needs to point to the fourth element of an array, that is, hold address of fourth array element in above case.

Since *ptr* points to the third element in the above example,  $ptr + 1$  will point to the fourth element.

### Example 1: C++ Pointers and Arrays

#### 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
```

## Example 2: Pointer and Arrays

**C++ Program to display address of array elements using pointer notation.**

```
#include <iostream>
```

```
using namespace std;
```

```
int main() {
```

```
    float arr[5];
```

```
    cout<<"Displaying address using pointers notation: "<< endl;
```

```
    for (int i = 0; i < 5; ++i) {
```

```
        cout << arr + i <<endl;
```

```
    }
```

```
    return 0;
```

```
}
```

## Output

Displaying address using pointers notation:

0x7fff5fbff8a0

0x7fff5fbff8a4

0x7fff5fbff8a8

0x7fff5fbff8ac

0x7fff5fbff8b0

## Example 3: C++ Pointer and Array

**C++ Program to insert and display data entered by using pointer notation.**

```
#include <iostream>
```

```
using namespace std;
```

```
int main() {
```

```
    float arr[5];
```

```
    // Inserting data using pointer notation
```

```
    cout << "Enter 5 numbers: ";
```

```
    for (int i = 0; i < 5; ++i) {
```

```
        cin >> *(arr + i) ;
```

```
    }
```

```
    // Displaying data using pointer notation
```

```
    cout << "Displaying data: " << endl;
```

```
    for (int i = 0; i < 5; ++i) {
```

```
        cout << *(arr + i) << endl ;
```

```
    }
```

```
    return 0;
```

```
}
```

## Output



Enter 5 numbers: 2.5

3.5

4.5

5

2

Displaying data:

2.5

3.5

4.5

5

2

## **C++ Program to Access Elements of an Array Using Pointer**

### **Example: Access Array Elements Using Pointer**

```
#include <iostream>
using namespace std;
int main()
{
    int data[5];
    cout << "Enter elements: ";

    for(int i = 0; i < 5; ++i)
        cin >> data[i];

    cout << "You entered: ";
    for(int i = 0; i < 5; ++i)
        cout << endl << *(data + i);

    return 0;
}
```

Output:

Enter elements: 1

2

3

5

4

You entered: 1

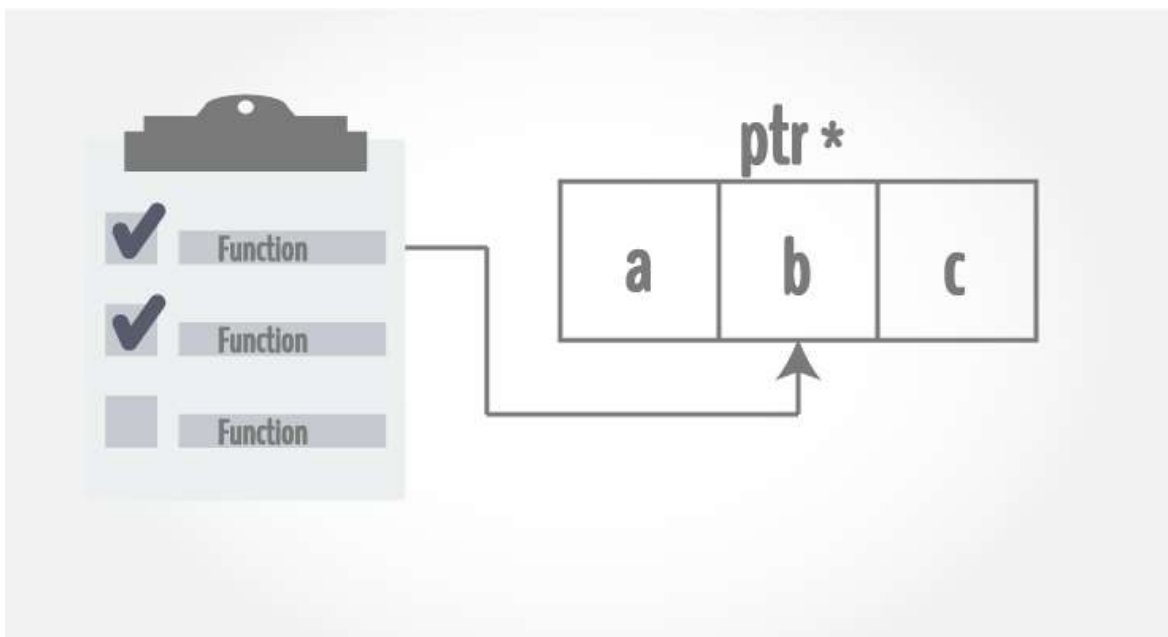
2  
3  
5  
4

In this program, the five elements are entered by the user and stored in the integer array *data*.

Then, the *data* array is accessed using a for loop and each element in the array is printed onto the screen.

## C++ Call by Reference: Using pointers

Here we learn how to pass pointers as an argument to the function, and use it efficiently in our program.



In C++ Functions we learned about passing arguments to a function. This method used is called passing by value because the actual value is passed.

However, there is another way of passing an argument to a function where the actual value of the argument is not passed. Instead, only the reference to that value is passed.

### Example 1: Passing by reference without pointers

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

```
// Function prototype
void swap(int&, int&);

int main()
{
    int a = 1, b = 2;
    cout << "Before swapping" << endl;
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    swap(a, b);
    cout << "\nAfter swapping" << endl;
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    return 0;
}

void swap(int& n1, int& n2) {
    int temp;
    temp = n1;
    n1 = n2;
    n2 = temp;
}
```

## Output

Before swapping

a = 1

b = 2

After swapping

a = 2

b = 1

In `main()`, two integer variables *a* and *b* are defined. And those integers are passed to a function `swap()` by reference.

Compiler can identify this is pass by reference because function definition is `void swap(int& n1, int& n2)` (notice the **&** sign after data type).

Only the reference (address) of the variables *a* and *b* are received in the `swap()` function and swapping takes place in the original address of the variables.

In the `swap()` function, *n1* and *n2* are formal arguments which are actually same as variables *a* and *b* respectively.

### **Example 2: Passing by reference using pointers**

```
//program to swap numbers using call by reference with pointers
```

```
#include <iostream>
```

```
using namespace std;
```

```
// Function prototype
```

```
void swap(int*, int*);
```

```
int main()
```

```
{
```

```
    int a = 1, b = 2;
```

```
    cout << "Before swaping" << endl;
```

```
    cout << "a = " << a << endl;
```

```
    cout << "b = " << b << endl;
```

```
    swap(&a, &b);
```

```
    cout << "\nAfter swaping" << endl;
```

```
    cout << "a = " << a << endl;
```

```
    cout << "b = " << b << endl;
```

```
    return 0;
```

```
}
```

```
void swap(int* n1, int* n2) {
    int temp;
    temp = *n1;
    *n1 = *n2;
    *n2 = temp;
}
```

The output of this example is same as before.

In this case, the address of variable is passed during function call rather than the variable itself.

```
swap(&a, &b); // &a is address of a and &b is address of b
```

Since the address is passed instead of value, dereference operator must be used to access the value stored in that address.

```
void swap(int* n1, int* n2) {
    ... ..
}
```

The *\*n1* and *\*n2* gives the value stored at address *n1* and *n2* respectively.

Since *n1* contains the address of *a*, anything done to *\*n1* changes the value of *a* in *main()* function as well. Similarly, *b* will have same value as *\*n2*.

### **Assignment:**

Write a program to add two integer numbers using call by reference with pointers.

```
#include<iostream>

using namespace std;

int add(int*,int*);

int main()

{
```

```

    int a,b;

    cout<<"Enter any two integer numbers: ";

    cin>>a>>b;

    cout<<"The addition is : "<<add(&a,&b)<<endl;

    return 0;

}

int add(int* n1,int* n2)

{

    return(*n1 + *n2);

}

```

### **Pointer to objects:**

A pointer to a C++ class is done exactly the same way as a pointer to a structure and to access members of a pointer to a class we use the member access operator -> operator, just as we do with pointers to structures. Also as with all pointers, we must initialize the pointer before using it.

Let us try the following example to understand the concept of pointer to a class:

```

#include <iostream>
using namespace std;
class Box
{
private:
    double length, breadth, height;
public:
    // Constructor definition
    Box(double l = 2.0, double b = 2.0, double h = 2.0)

```

```

    {
        cout <<"Constructor called." << endl;
        length = l;
        breadth = b;
        height = h;
    }
    double Volume() {
        return length * breadth * height;
    }
};

int main(void) {
    Box Box1(3.3, 1.2, 1.5);    // Declare box1
    Box Box2(8.5, 6.0, 2.0);    // Declare box2
    Box *ptrBox;                // Declare pointer to a class.

    // Save the address of first object
    ptrBox = &Box1;

    // Now try to access a member using member access operator
    cout << "Volume of Box1: " << ptrBox->Volume() << endl;

    // Save the address of second object
    ptrBox = &Box2;

    // Now try to access a member using member access operator
    cout << "Volume of Box2: " << ptrBox->Volume() << endl;

    return 0;
}

```

When the above code is compiled and executed, it produces the following result:

```

Constructor called.
Constructor called.

```

Volume of Box1: 5.94

Volume of Box2: 102

### **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 the object for which this function was called. For example, the function call A.max() will set the pointer **this** to the address of the object A. The starting address is the 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** acts as an implicit argument to all the member functions. Consider the following simple example:

```
class ABC
{
    int a;
    .....
};
```

The private variable 'a' can be used directly inside a member function, like a=125;

We can also use the following statement to do the same job:

```
    this->a = 125;
```

### **Example using this pointer:**

#### **//program demo using this pointer**

```
#include<iostream>

using namespace std;

class A
{
```



```

private:
    int a;
public:
    A() //constructor
    {
        this->a=25;
        cout<<"From A : a ="<<a<<endl;
    }
};

int main()
{
    A obj;
    return 0;
}

```

## Pointers to Derived Classes:

We can use pointers not only to the base objects but also to the objects of derived classes. Pointers to objects of a base class are type-compatible with pointers to objects of a derived class. Therefore, a single pointer variable can be made to point to objects belonging to different classes. For example, if B is a base class and D is a derived class from B, then a pointer declared as a pointer to B can also be pointer to D. Consider the following declarations:

```

B *ptr; //pointer to class B type variable

B objb; // base object

D objd; //derived object

```

```
ptr = &objb; //ptr points to the object objb
```

We can make ptr to point to the object d as follows:

```
ptr = &objd; //ptr points to object d
```

This is perfectly valid with C++ because objd is an object derived from the class B.

However, there is a problem in using ptr to access the public members of the derived class D. Using ptr, we can access only those members which are inherited from B and not the members that originally belong to D.

//program demo using Pointers to Derived Classes

```
#include<iostream>
```

```
using namespace std;
```

```
class B
```

```
{
```

```
    public:
```

```
        int a;
```

```
void show()
```

```
{
```

```
cout<<"a = "<<a<<endl;
```

```
}
```

```
};
```

```
class D: public B
```

```
{
```

```
    public:
```

```

int b;

void show()
{
    cout<<"a = "<<a<<endl;
    cout<<"b = "<<b<<endl;
}

};

int main()
{
    B *ptr;    // base class pointer
    B objb;

    ptr=&objb; //base address

    ptr->a = 20;

    cout<<"ptr points to base object"<<endl;

    ptr->show();


//derived class
    D objd;
    ptr = &objd;
    ptr->a= 100;
    // ptr->b= 120; Won't work
    cout<<"ptr points to derived object"<<endl;
    ptr->show();


//Accessing b using a pointer of derived class D
    D *ptrd;
    ptrd=&objd;
    ptrd->b = 120;

```

```
    cout<<"\n\nptrd is derived type pointer"<<endl;
    ptrd->show();
return 0;
}
```

### **Output:**

ptr points to base object

a = 20

ptr points to derived object

a = 100

ptrd is derived type pointer

a = 100

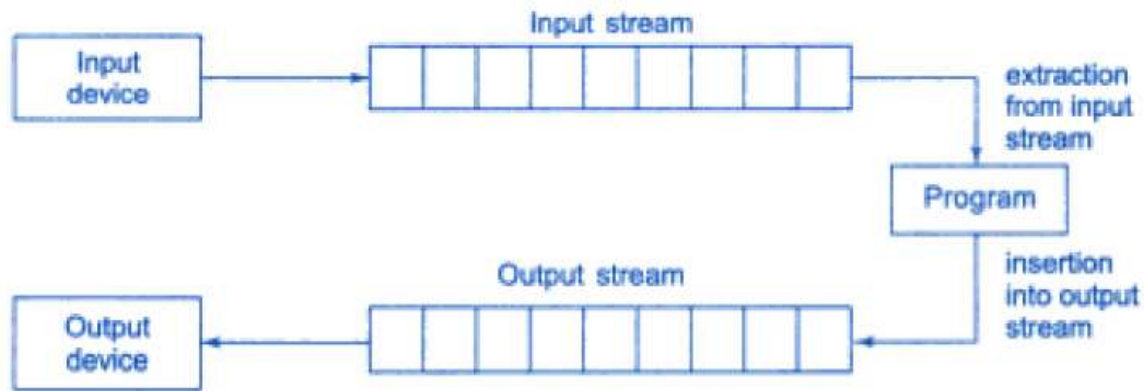
b = 120

## **MANAGING CONSOLE, I/O OPERATIONS**

### **C++ Streams:**

The I/O system in C++ is designed to work with a wide variety of devices including terminals, disks, and tape drives. Although each device is very different, the I/O system supplies an interface to the programmer that is independent of the actual device being accessed. This interface is known as *stream*.

**A stream is a sequence of bytes.** It acts either as a *source* from which the input data can be obtained or as a *destination* to which the input data can be sent. The source stream that provides data to the program is called the **input stream** and the destination stream that receives output from the program is called the **output stream**. In other words, a program extracts the bytes from an input stream and inserts bytes into an output stream as shown in figure below:



### Data Streams

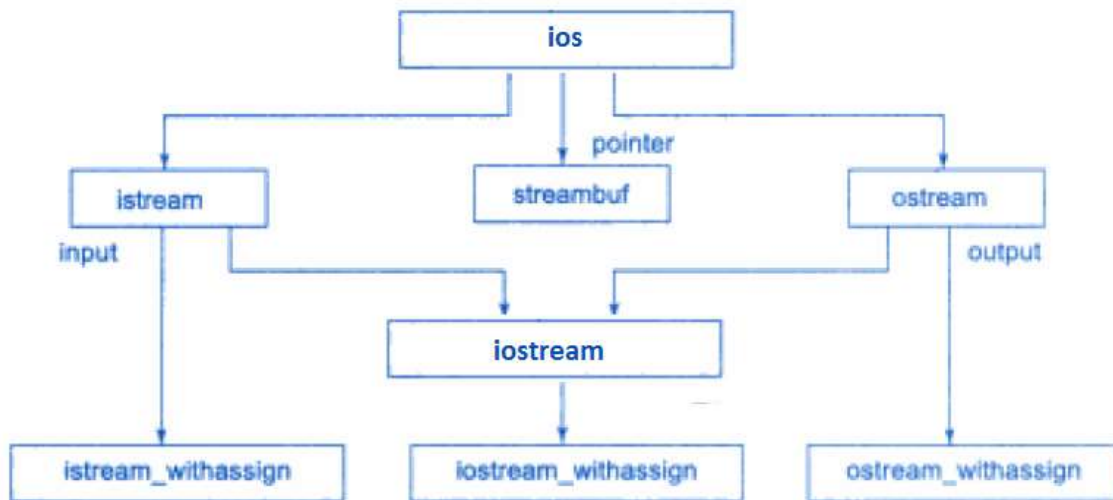
The data in the input stream can come from the keyboard or any other storage device. Similarly, the data in the output stream can go to the screen or any other storage device. **A stream acts as an interface between the program and the input/output device.** Therefore, a C++ program handles data independent of the devices used.

C++ contains several pre-defined streams that are automatically opened when a program begins its execution. These include `cin` and `cout`. We know that `cin` represents the input stream connected to the standard input device usually keyboard and `cout` represents the output stream connected to the standard output device usually the screen. We can redirect streams to other devices or files, if necessary.

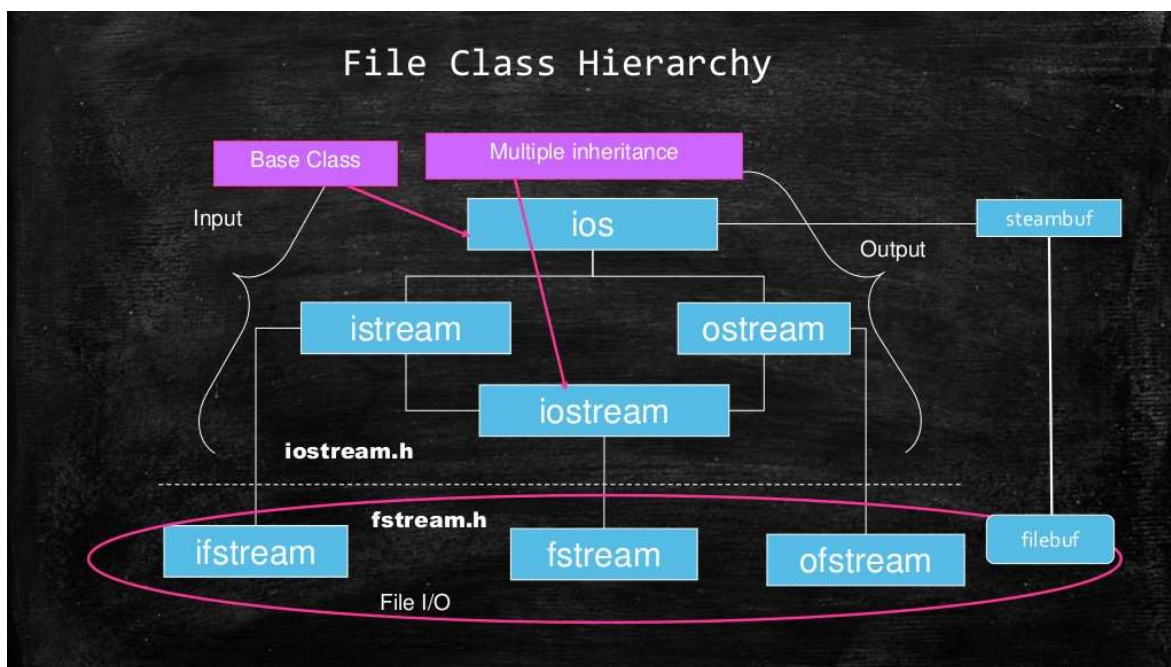
### C++ Stream Classes:

Explain the various file stream classes?

The C++ I/O system contains a hierarchy of classes that are used to define various streams to deal with both the console and disk files. These classes are called stream classes. The figure below shows the hierarchy of stream classes used for input and output operations with the console unit. These classes are declared in the header file **`iostream`**. This file should be included in all the programs that communicate with the console unit.



Stream Classes for Console I/O operations



As seen in the above figure, ios is the base class for istream(input stream) and ostream(output stream) which are in turn, base classes for iostream(input/output stream). The class ios is declared as the virtual base class so that only one copy of its members are inherited by the iostream.

The class ios provides the basic support for formatted and unformatted I/O operations. The class istream provides the facilities for formatted and unformatted input, while the class ostream provides the facilities for formatted output. The class iostream provides the facilities for handling both input and output streams. Three classes, namely:

1. `istream_withassign`
2. `ostream_withassign`
3. `iostream_withassign`

add assignment operators to these classes.

#### *Stream classes for console operations*

<i>Class name</i>	<i>Contents</i>
<b>ios</b> (General input/output stream class)	<ul style="list-style-type: none"> <li>Contains basic facilities that are used by all other input and output classes</li> <li>Also contains a pointer to a buffer object (<b>streambuf</b> object)</li> <li>Declares constants and functions that are necessary for handling formatted input and output operations</li> </ul>
<b>istream</b> (input stream)	<ul style="list-style-type: none"> <li>Inherits the properties of <b>ios</b></li> <li>Declares input functions such as <b>get()</b>, <b>getline()</b> and <b>read()</b></li> <li>Contains overloaded extraction operator <b>&gt;&gt;</b></li> </ul>
<b>ostream</b> (output stream)	<ul style="list-style-type: none"> <li>Inherits the properties of <b>ios</b></li> <li>Declares output functions <b>put()</b> and <b>write()</b></li> <li>Contains overloaded insertion operator <b>&lt;&lt;</b></li> </ul>
<b>iostream</b> (input/output stream)	<ul style="list-style-type: none"> <li>Inherits the properties of <b>ios</b>, <b>istream</b> and <b>ostream</b> through multiple inheritance and thus contains all the input and output functions</li> </ul>
<b>streambuf</b>	<ul style="list-style-type: none"> <li>Provides an interface to physical devices through buffers</li> <li>Acts as a base for <b>filebuf</b> class used ios files</li> </ul>

## Unformatted I/O operations:

### Overloaded Operators **>>** and **<<**

We have used the objects `cin` and `cout` (predefined in the `iostream` file) for the input and output of data of various types. This has been made possible by overloading the operators **>>** and **<<** to recognize all the basic C++ types. The **>>** operators are overloaded in the `istream` class and **<<** is overloaded in the `ostream` class. The following is the general format for reading data from the keyboard:

```
cin >> variable1 >> variable2 >> ..... >> variableN
```

`variable1`, `variable2`, ..... are valid C++ variable names.

The operators >> reads the data character by character and assigns it to the indicated location. The reading for a variable will be terminated at the encounter of a while space or a character that does not match the destination type. For example, consider the following code:

```
int code;  
  
cin>>code;
```

The general form for displaying data on the screen is:

```
cout<<item1<<item2<<.....<<itemN;
```

The items item1,item2 through itemN may be variables or constants of any basic type.

### **get() and put() functions:**

The classes istream and ostream define two member functions get() and put() respectively to handle the single character input/output operations.

Example:

```
//program demo using get()  
  
#include<iostream>  
  
using namespace std;  
  
int main()  
{  
  
char c;  
  
cin.get(c);  
  
while(c!='\n')  
{  
  
    cout<<c;  
  
    cin.get(c);
```



```
}  
  
return 0;  
  
}
```

Output:

Sai Pali Institute of Technology & Management

Sai Pali Institute of Technology & Management

The function put(), a member of ostream class, can be used to output a line of text, character by character. For example,

```
cout.put('x');
```

displays the character x and

```
cout.put(ch);
```

displays the value of variable ch.

```
#include<iostream>
```

```
using namespace std;
```

```
int main()
```

```
{
```

```
char c;
```

```
c = cin.get();    //read a character
```

```
while(c!='\n')
```

```
{
```

```
    cout.put(c);  //displays the character on the screen
```

```
    c = cin.get();
```

```
}  
  
return 0;  
  
}
```

Example 2:

```
#include<iostream>  
  
using namespace std;  
  
int main()  
{  
    int count=0;  
  
    char c;  
  
    cout<<"Input any text..."<<endl;  
  
    cin.get(c);  
  
    while(c!='\n')  
    {  
        cout.put(c);  
  
        count++;  
  
        cin.get(c);  
    }  
  
    cout<<"\nNumber of characters = "<<count<<endl;  
  
    return 0;  
  
}
```

Output:

Input any text...

Sai Pali Institute of Technology & Management

Number of characters = 45

When we type a line of input, the text is sent to the program as soon as we press the RETURN key. The program then reads one character at a time using the statement `cin.get(c);` and displays it using the statement `cout.put(c);` . The process is terminated when the newline character is encountered.

### **getline() and write() functions:**

We can read and display a line of text more efficiently using the line-oriented input/output functions `getline()` and `write()`. The `getline()` function reads a whole line of text that ends with a newline character. This function can be invoked by using the object `cin` as follows:

```
cin.getline(line,size);
```

Example:

```
char name[20];
```

```
cin.getline(name,20);
```

Assume that we have given the following input through the keyboard:

Bjarne Strastrup <press RETURN>

This input will be read correctly and assigned to the character array `name`. Let us suppose the input is as follows:

Object Oriented Programming <press RETURN>

In this case, the input will be terminated after reading the following 19 characters:

Object Oriented Pro

Remember, the two blank spaces contained in the string are also taken into account.

```
//program demo using getline()

#include<iostream>

using namespace std;

int main()

{

    int size=20;

    char city[20];

    cout<<"Enter city name: ";

    cin.getline(city,size);

    cout<<"City name: "<<city<<endl;

    return 0;

}
```

Example 2:

```
//program demo using get()and put()

#include<iostream>

using namespace std;

int main()

{

    int size=20;

    char city[20];

    cout<<"Enter city name: ";

    cin.getline(city,size);

    //cout<<"City name: "<<city<<endl;

    cout.write("City Name: ",10);
```

```
cout.write(city,7);  
  
return 0;  
  
}
```

The write() function displays an entire line and has the following form:

```
cout.write(line,size);
```

Example 1:

```
//program demo using getline()  
  
#include<iostream>  
  
using namespace std;  
  
int main()  
{  
  
    char name[25];  
  
    //cout<<"Enter your name: ";  
  
    cout.write("Enter your name: ",17);  
  
    cin.getline(name,8);  
  
    //cout<<"Your Name: "<<name<<endl;  
  
    cout.write(name,8);  
  
    return 0;  
  
}
```

Example 2:

```
//program demo using write()
```

```

#include<iostream>
#include<string.h>
using namespace std;
int main()
{
char* string1="C++";
char* string2="Programming";
int m = strlen(string1);
int n = strlen(string2);
for(int i=1;i<n;i++)
{
cout.write(string2,i);
cout<<endl;
}
for(int i=n;i>0;i--)
{
    cout.write(string2,i);
    cout<<endl;
}
//concatenating strings
cout.write(string1,m).write(string2,n);
cout<<endl;
cout.write(string1,10);
return 0;
}

```

Output:

P  
Pr  
Pro  
Prog  
Progr  
Progra

Program  
Programm  
Programmi  
Programmin  
Programming  
Programmin  
Programmi  
Programm  
Program  
Progra  
Progr  
Prog  
Pro  
Pr  
P  
C++Programming  
C++ Progra

## Formatted Console I/O Operations:

C++ supports a number of features that could be used for formatting the output. These features include:

1. ios class functions and flags
2. Manipulators
3. User-defined output functions

The ios class contains a large number of member functions that would help us to format the output in a number of ways. The most important ones among them are listed in table below:

### ios format functions

Function	Task
<b>Width ()</b>	To specify the required field size for displaying an output value
<b>precision ()</b>	To specify the number of digits to be displayed after the decimal point of a float value
<b>fill()</b>	To specify a character that is used to fill the unused portion of a field
<b>setf()</b>	To specify format flags that can control the form of output display (such as left-justification and right-justification)
<b>unsetf()</b>	To clear the flags specified

### ios format functions

Manipulators are special functions that can be included in the I/O statements to alter the format parameters of a stream. The table below shows some important manipulators functions that are used frequently used. To access these manipulators, the file *omanip* should be included in the program.

### Manipulators

Manipulators	Equivalent ios function
<b>setw()</b>	<b>width()</b>
<b>setprecision()</b>	<b>precision()</b>
<b>setfill()</b>	<b>fill()</b>
<b>setiosflags()</b>	<b>setf()</b>
<b>resetiosflags()</b>	<b>unsetf()</b>

### Defining field width: width ()

We can use the **width()** function to define the width of a field necessary for the output of an item. Since, it is a member function, we have to use an object to invoke it, as shown below:

```
cout.width(w);
```

where  $w$  is the field width (number of columns). The output will be printed in a field of  $w$  characters wide at the right end of the field. The **width()** function can specify the field width for only one item (the item that follows immediately). After printing one item (as per the specifications) it will revert back to the default. For example, the statements

```
cout.width(5);
cout << 543 << 12 << "\n";
```

will produce the following output:

			5	4	3	1	2
--	--	--	---	---	---	---	---



The value 543 is printed right-justified in the first five columns. The specification width(5) does not retain the setting for printing the number 12. This can be improved as follows:

```
cout.width(5);  
cout << 543;  
cout.width(5);  
cout << 12 << "\n";
```

This produces the following output:

		5	4	3			1	2
--	--	---	---	---	--	--	---	---

Remember that the field width should be specified for each item separately. C++ never truncates the values and therefore, if the specified field width is smaller than the size of the value to be printed, C++ expands the field to fit the value. Program 10.4 demonstrates how the function **width()** works.

Example:

```
#include<iostream>  
using namespace std;  
int main()  
{  
int items[4] = {10,8,12,15};  
int cost[4] = {75,100,60,99};  
cout.width(5);  
cout<<"ITEMS";  
cout.width(8);  
cout<<"COST";  
cout.width(15);  
cout<<"TOTAL VALUE"<<endl;  
int sum=0;  
for(int i=0;i<4;i++)  
{  
cout.width(5);  
cout<<items[i];  
cout.width(8);  
cout<<cost[i];  
int value = items[i] * cost[i];  
cout.width(15);  
cout<<value<<endl;
```

```

    sum = sum + value;
}
cout<<"\nGrand Total = ";
cout.width(2);
cout<<sum<<endl;
return 0;
}

```

Output:

ITEMS	COST	TOTAL VALUE
10	75	750
8	100	800
12	60	720
15	99	1485

Grand Total = 3755

### Setting Precision: `precision()`

By default, the floating numbers are printed with six digits after the decimal point. However, we can specify the number of digits to be displayed after the decimal point while printing the floating-point numbers. This can be done by using the **`precision()`** member function as follows:

```
cout.precision(d);
```

where `d` is the number of digits to the right of the decimal point. For example, the statements

```

cout.precision(3);
cout << sqrt(2) << "\n";
cout << 3.14159 << "\n";
cout << 2.50032 << "\n";

```

will produce the following output:

```

1.141  (truncated)
3.142  (rounded to the nearest cent)
2.5    (no trailing zeros)

```

Example:

```

//program demo precision setting with precision()
#include<iostream>
#include<cmath>

```

```

using namespace std;
int main()
{
cout<<"Precision set to 3 digits"<<endl;
cout.precision(3);
cout.width(10);
cout<<"VALUE";
cout.width(15);
cout<<"SQRT OF VALUE"<<endl;
for(int n=1;n<=5;n++)
{
    cout.width(8);
    cout<<n;
    cout.width(13);
    cout<<sqrt(n)<<endl;
}
cout<<"\nPrecision set to 5 digits"<<endl;
cout.precision(5);
cout<<"sqrt(10) = "<<sqrt(10)<<endl;
return 0;
}

```

Output:

Precision set to 3 digits

VALUE	SQRT OF VALUE
1	1
2	1.41
3	1.73
4	2
5	2.24

Precision set to 5 digits

sqrt(10) = 3.1623

## Filling and Padding: fill()

We have been printing the values using much larger field widths than required by the values. The unused positions of the field are filled with white spaces, by default. However, we can use the fill() function to fill the unused positions by any desired character. It is used in the following form:

```
cout.fill (ch);
```

Where *ch* represents the character which is used for filling the unused positions. Example:

```
cout.fill('*');  
cout.width(10);  
cout << 5250 << "\n";
```

The output would be:

*	*	*	*	*	*	5	2	5	0
---	---	---	---	---	---	---	---	---	---

Financial institutions and banks use this kind of padding while printing cheques so that no one can change the amount easily. Like **precision()**, **fill()** stays in effect till we change it.

Example:

```
//program demo padding with fill()  
#include<iostream>  
#include<iomanip>  
using namespace std;  
int main()  
{  
cout.fill('<');  
cout.precision(3);  
for(int n=1;n<=6;n++)  
{  
cout.width(5);  
cout<<n;  
cout.width(10);  
cout<<1.0/float(n)<<endl;  
if(n==3)  
cout.fill('>');  
}  
cout<<"\nPadding Changed"<<endl;  
cout.fill('#'); //fill() reset  
cout.width(15);  
cout<<"12.345678"<<endl;  
return 0;  
}
```

Output:

```
<<<<1<<<<<<<<1
```

```
<<<<2<<<<<<0.5
<<<<3<<<<<0.333
>>>>4>>>>>0.25
>>>>5>>>>>>0.2
>>>>6>>>>>>0.167
```

Padding Changed  
#####12.345678

Pg No 322

<http://www.sitesbay.com/cpp/cpp-file-handling>

<https://courses.cs.vt.edu/cs1044/Notes/C04.IO.pdf>

PgNo: 17

## Managing Output with Manipulators:

The header file `iomanip` provides a set of functions called manipulators which can be used to manipulate the output formats. They provide the same features as that of the `ios` member functions and flags. For example, two or more manipulators can be used as a chain in one statement as shown below:

```
cout<<manip1<<manip2<<manip3<<item;
```

```
cout<<manip1<<item1<<manip2<<item2;
```

This kind of concatenation is useful when we want to display several columns of output.

The most commonly used manipulators are shown in the table below:

<i>Manipulator</i>	<i>Meaning</i>	<i>Equivalent</i>
<code>setw (int w)</code>	Set the field width to w.	<code>width( )</code>
<code>setprecision(int d )</code>	Set the floating point precision to d.	<code>precision( )</code>
<code>setfill(int c)</code>	Set the fill character to c.	<code>fill( )</code>
<code>setiosflags(long f)</code>	Set the format flag f.	<code>setf( )</code>
<code>resetiosflags(long f)</code>	Clear the flag specified by f.	<code>unsetf( )</code>
<code>endl</code>	Insert new line and flush stream.	<code>"\n"</code>

Some examples of manipulators are given below:

```
cout<<setw(10)<<12345;
```

This statement prints the value 12345 right-justified in a field width of 10 characters.

The output can be made left-justified by modifying the statement as follows:

```
cout<<setw(10)<<setiosflags(ios::left)<<12345;
```

Example:

```
#include<iostream>
#include<iomanip>
#include<cmath>
using namespace std;
int main()
{
    cout<<setw(10)<<12345<<endl;
    cout<<setw(10)<<setiosflags(ios::left)<<12345;

    cout<<setw(5)<<setprecision(2)<<1.2345<<setw(10)<<setprecision(4)<<sqrt(2)<<setw(15)<<setiosflags(ios::scientific)<<sqrt(3)<<endl;

    return 0;
}
```

Example 2:

```
#include<iostream>
#include<iomanip>
#include<cmath>
using namespace std;
int main()
{
    float basic, ta,da,gs;
    basic=10000; ta=800; da=5000;
    gs=basic+ta+da;
    cout<<setw(10)<<"Basic"<<setw(10)<<basic<<endl
        <<setw(10)<<"TA"<<setw(10)<<ta<<endl
        <<setw(10)<<"DA"<<setw(10)<<da<<endl
        <<setw(10)<<"GS"<<setw(10)<<gs<<endl;
    return 0;
    return 0;
}
```

}

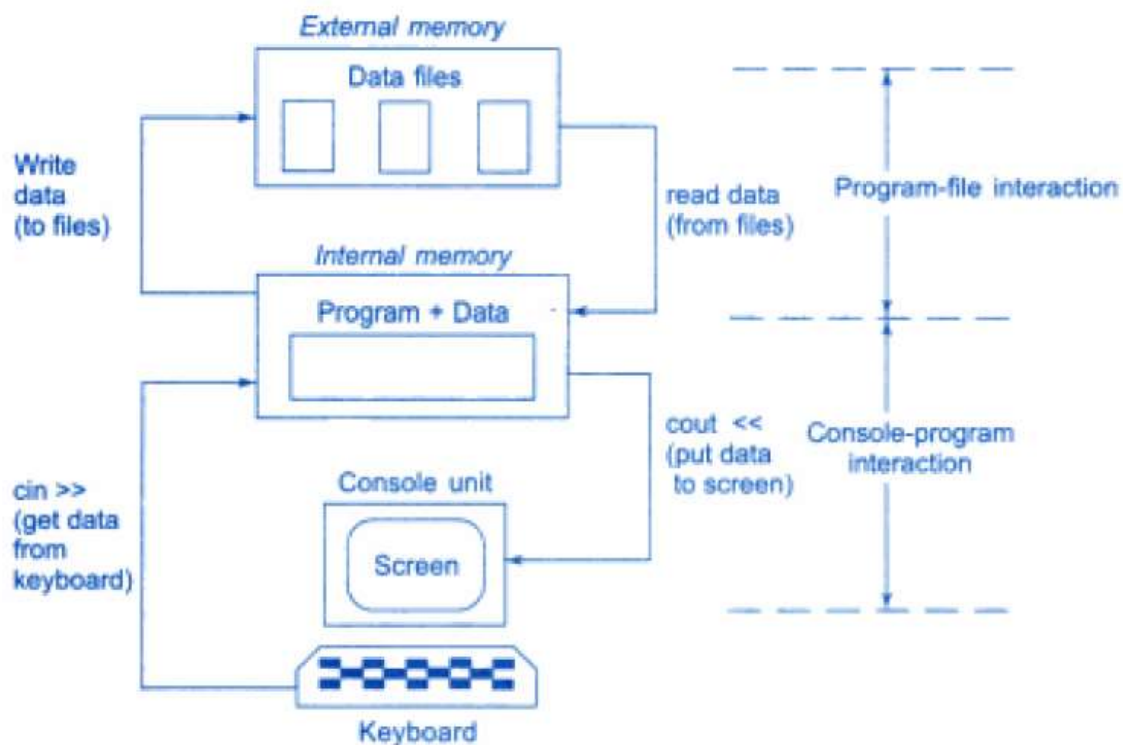
Output:

Basic	10000
TA	800
DA	5000
GS	15800

## WORKING WITH FILES:

**File Handling** concept in C++ language is used for store a data permanently in computer. Using file handling we can store our data in Secondary memory (Hard disk). A file is a collection of related data stored in a particular area on the disk. Programs can be designed to perform the read and write operations on these files. A program typically involves either or both of the following kinds of data communication:

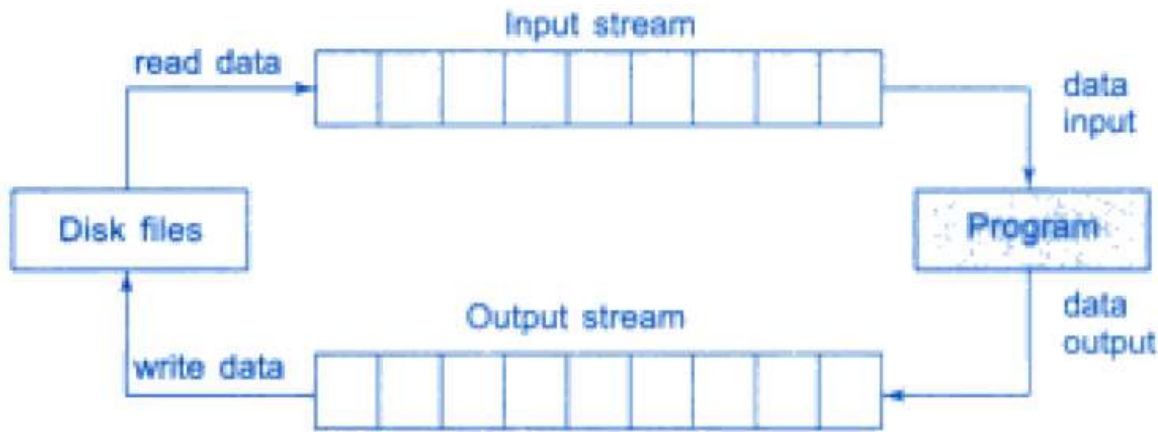
1. Data transfer between the console unit and the program.
2. Data transfer between the program and a disk file.



Console program file interaction

The I/O system of C++ handles file operations which are very much similar to the console input and output operations. It uses file streams as an interface between the programs and the files. The stream that supplies data to the program is known as input stream and the one that receives data from the program is known as output stream. In other words, the input stream extracts (or reads) data from the file and output stream inserts (or writes) data to the file.





File input and output streams

The input operation involves the creation of an input stream and linking it with the program and the input file. Similarly, the output operation involves establishing an output stream with the necessary links with the program and the output file.

### Why use File Handling

- For permanent storage.
- The transfer of input - data or output - data from one computer to another can be easily done by using files.

For read and write from a file you need another standard C++ library called **fstream**, which defines three new data types:

Datatype	Description
ofstream	This is used to create a file and write data on files
ifstream	This is used to read data from files
fstream	This is used to both read and write data from/to files

### How to achieve File Handling?

For achieving file handling in C++ we need follow following steps

1. Naming a file

2. Opening a file
3. Reading data from file
4. Writing data into file
5. Closing a file

### **Functions use in File Handling**

Function	Operation
open()	To create a file
close()	To close an existing file
get()	Read a single character from a file
put()	write a single character in file.
read()	Read data from file
write()	Write data into file.

A file stream can be defined using the classes ifstream, ofstream and fstream that are contained in the header file fstream. The class to be used depends upon the purpose that is, whether we want to read data from the file or write data to it. A file can be opened in two ways:

- 1) Using the constructor function of the class.
- 2) Using the member function open() of the class.

The first method is useful when we use only one file in the stream. The second method is used when we want to manage multiple files using one stream.

### **Opening files using constructor:**

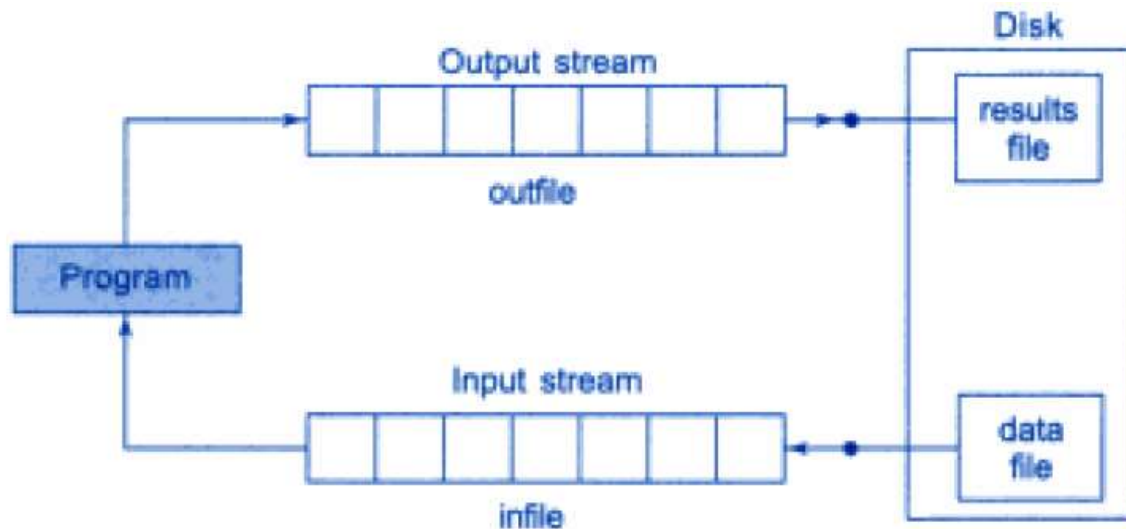
We know that a constructor is used to initialize an object while it is being created. Here, a filename is used to initialize the file stream object. This involves the following steps:

1. Create a file stream object to manage the stream using the appropriate class. That is to say, the class ofstream is used to create the output stream and the class ifstream to create the input stream.

2. Initialize the file object with the desired filename.

For example, the following statement opens a file named "results" for output:

```
ofstream outfile("results"); // output only
```



Two file streams working on separate files

Similarly, the following statement declares infile as an ifstream object and attaches it to the file data for reading(input).

```
ifstream infile("data"); // input only
```

Example:

```
//program demo for creating files with constructor function
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
char name[30];
float cost;
ofstream outf("C:\\ITEM.DAT");
cout<<"Enter item name: ";
cin>>name;
outf<<name<<endl; //write to file ITEM.DAT
cout<<"Enter item cost: ";
cin>>cost;
outf<<cost<<endl; //write to file ITEM
outf.close(); //disconnect ITEM file from outf
```

```

ifstream inf("C:\\ITEM.DAT");    //connect ITEM file to inf
inf>>name; //read name from file ITEM
inf>>cost;  //read cost from file ITEM
cout<<endl;
cout<<"Item Name: "<<name<<endl;
cout<<"Item Cost: "<<cost<<endl;
inf.close(); //Disconnect ITEM from inf

return 0;
}

```

Opening files using open():

The function open() can be used to open multiple files that use the same stream object. For example, we may want to process a set of files sequentially. In such cases, we may create a single stream object and use it to open each file in turn. This is done as follows:

```

file-stream-class stream-object;
    stream-object.open("filename");

```

Example:

```

ofstream outfile; //create stream for output
outfile.open("C:\\DATA1.DAT"); //connect stream to DATA1
.....
.....
outfile.close(); // disconnect stream from DATA1

outfile.open("C:\\DATA2.DAT"); // connect stream to DATA2
.....
.....
outfile.close(); //disconnect stream from DATA2
.....
.....

```

The above program segment opens two files in sequence for writing the data. Note that the first file is closed before opening the second one. **This is necessary because a stream can be connected to only one file at a time.**

//program demo working with multiple files i.e. creating files with open() function

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
    ofstream fout; //create output stream
    fout.open("C:\\\\COUNTRY.DAT"); //connect to file
    fout<<"United States of America"<<endl;
    fout<<"United Kingdom"<<endl;
    fout<<"South Korea"<<endl;
    fout.close(); //disconnect
    fout.open("C:\\\\CAPITAL.DAT"); //connect
    fout<<"Washington"<<endl;
    fout<<"London"<<endl;
    fout<<"Seoul"<<endl;
    fout.close(); //disconnect
```

//reading the files

```
const int N=80; //size of line
char line[N];
ifstream fin; //create input stream
fin.open("D:\\\\TESTING\\\\COUNTRY.DAT");
cout<<"Contents of country file..."<<endl;
while(fin)
{
    fin.getline(line,N); //read a line
    cout<<line<<endl; //display it
```

```

}
fin.close(); //disconnect
fin.open("C:\\CAPITAL.DAT"); //connect
cout<<"\nContents of capital file"<<endl;
while(fin)
{
    fin.getline(line,N);
    cout<<line<<endl;
}
fin.close();
return 0;
}

```

Output:

Contents of country file...

United States of America

United Kingdom

South Korea

Contents of capital file

Washington

London

Seoul

Example:

//program demo Read and Write data from/to File

```
#include<iostream>
```

```
#include<conio.h>
```

```
#include<stdio.h>
```

```

#include<stdlib.h>
#include<fstream>
using namespace std;
class student

{
    public:
    int roll;
    char name[15],f_name[20];
    void put();
    void get();
    void switch_case();
}; student s;

void student::put()
{
    fstream file;
    cout<<"Enter roll no: ";
    cin>>roll;
    cout<<"Enter name: ";
    //gets(name);
    cin>>name;
    cout<<"Enter father name: ";
    //gets(f_name);
    cin>>f_name;
    file.open("C:\\\\stu1.dat",ios::out|ios::app);
// file.seekp(0,ios::beg);
    file.write((char *)this,sizeof(student));
    file.close();
    //getch();
    s.switch_case();
}

```

```
}
```

```
void student::get()
```

```
{
```

```
    int temp;
```

```
    cout<<"Enter roll no: ";
```

```
    cin>>temp;
```

```
    fstream file;
```

```
    file.open("c:\\stu1.dat",ios::in);
```

```
    file.seekg(0,ios::beg);
```

```
    while(file.read((char *)this,sizeof(student)));
```

```
    {
```

```
        if(roll==temp)
```

```
        {
```

```
            cout<<"roll no. "<<roll<<endl;
```

```
            cout<<"stu name: "<<name<<endl;
```

```
            cout<<"father name: "<<f_name<<endl;
```

```
        }
```

```
    }
```

```
    file.close();
```

```
    //getch();
```

```
    s.switch_case();
```

```
}
```

```
void student::switch_case()
```

```
{
```

```
    int i;
```

```
    cout<<"Enter your choice (1-Read, 2-Write, 3-exit): ";
```

```
    cin>>i;
```

```
    switch(i)
```

```
    {
```



case 1:

```
s.put();  
break;
```

case 2:

```
cout<<"Enter Roll No:";  
s.get();  
break;
```

case 3:

```
exit(0);
```

default:

```
cout<<"wrong choice "  
}  
}
```

int main()

```
{  
s.switch_case();  
return 0;  
}
```

## SEQUENTIAL INPUT OUTPUT OPERATIONS

The file stream classes support a number of member functions for performing the input and output operations on files. The functions `get()` and `put()` are capable of handling a single character at a time. The function `getline()` allows us handle multiple characters at a time. Another pair of functions i.e., `read()` and `write()` are capable of reading and writing blocks of binary data.

### **The `get()`, `getline()` and `put()` Functions**

The functions `get()` and `put()` are byte-oriented. That is, `get()` will read a byte of data and `put()` will write a byte of data. The `get()` has many forms, but the most commonly used version is shown here, along with `put()` :

```
istream & get(char & ch) ;  
ostream & put(char ch) ;
```

The `get()` function reads a single character from the associated stream and puts that value in `ch`. It returns a reference to the stream. The `put()` writes the value of `ch` to the stream and returns a reference to the stream.

The following program displays the contents of a file on the screen. It uses the `get()` function :

```
/* C++ Sequential Input/Output Operations on Files */
```

```
#include<iostream>  
#include<fstream>  
using namespace std;  
int main()  
{  
    char fname[20], ch;  
    ifstream fin;    // create an input stream  
  
    cout<<"Enter the name of the file: ";  
    cin.get(fname, 20);  
    cin.get(ch);  
  
    fin.open(fname, ios::in);    // open file  
    if(!fin)    // if fin stores zero i.e., false value  
    {  
        cout<<"Error occurred in opening the file..!!\n";  
        cout<<"Press any key to exit...\n";  
        exit(1);  
    }  
}
```

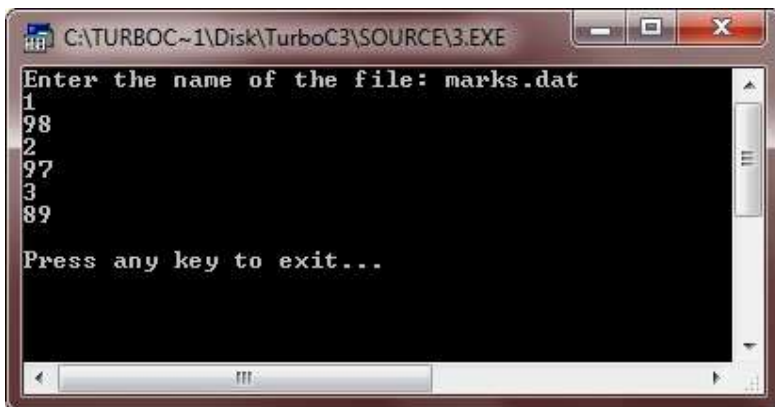
```

while(fin)          // fin will be 0 when eof is reached
{
    fin.get(ch);     // read a character
    cout<<ch;       // display the character
}

cout<<"\nPress any key to exit...\n";
fin.close();
return 0;
}

```

Here is the sample output of the above C++ program. Let's suppose that the file contain the following information:



As stated, when the end-of-file is reached, the stream associated with the file becomes zero. Therefore, when fin reaches the end of the file, it will be zero causing the while loop to stop.

The data written to a file using write() can only be read accurately using read().

The following program writes a structure to the disk and then reads it back in using write() and read() functions :

```
/* C++ Sequential Input and Output Operations with Files */
```

```
#include<iostream>
```

```

#include<fstream>

#include<string.h>

#include<stdlib.h>

using namespace std;

struct customer
{
    char name[20];

    float balance;
};

int main()
{
    customer savac;

    cout<<"Enter your name: ";

    cin.get(savac.name, 20);

    cout<<"Enter balance: ";

    cin>>savac.balance;


    ofstream fout;

    fout.open("Saving", ios::out | ios::binary);    // open output file

    if(!fout)

    {

```

```

        cout<<"File can\'t be opened...!!\n";

        cout<<"Press any key to exit...\n";

        exit(1);

    }

    fout.write((char *) & savac, sizeof(customer));    // write to file

    fout.close();        // close connection

    // read it back now

    ifstream fin;

    fin.open("Saving", ios::in | ios::binary);    // open input file

    fin.read((char *) & savac, sizeof(customer));    // read structure

    cout<<savac.name;        // display structure now

    cout<<" has the balance amount of Rs. "<<savac.balance<<"\n";

    fin.close();

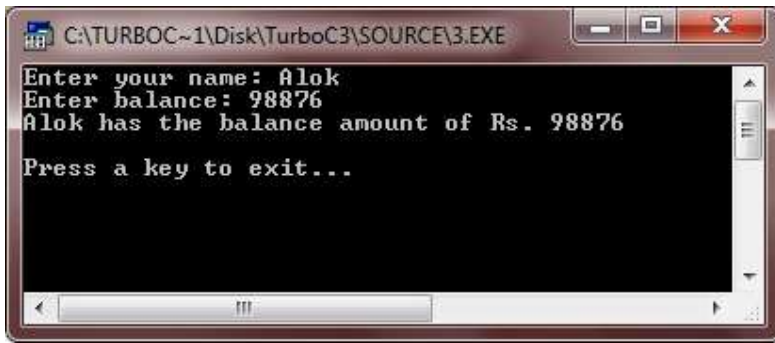
    cout<<"\nPress a key to exit...\n";

return 0;

}

```

Here is the sample run of the above C++ program:



As you can see, only a single call to `read()` and `write()` is necessary to read or write the entire structure. Each individual field need not be read or written separately.

## Reading and Writing Class Objects

The functions `read()` and `write()` can also be used for reading and writing class objects. These functions handle the entire structure of an object as a single unit, using the computer's internal representation of data. For example, the function `write()` copies a class object from memory byte by byte with no conversion. But one thing that must be remembered is that only data members are written to the disk file and not the member functions.

The length of an object is obtained by `sizeof` operator and it represents the sum total of lengths of all data members of the object. Here is an example:

```
/* C++ Sequential Input and Output Operations with Files */
```

```
#include<iostream.h>  
#include<fstream.h>  
#include<conio.h>  
#include<stdlib.h>
```

```
class student  
{  
    char name[20];  
    char grade;  
    float marks;
```

```

        public:
            void getdata(void);
            void display(void);

};

void student::getdata(void)
{
    char ch;
    cin.get(ch);
    cout<<"Enter name: ";
    cin.getline(name, 20);
    cout<<"Enter grade: ";
    cin>>grade;
    cout<<"Enter marks: ";
    cin>>marks;
    cout<<"\n";
}

void student::display(void)
{
    cout<<"Name: "<<name<<"\tGrade: "<<grade<<"\tMarks:
"<<marks<<"\n";
}

void main()
{
    clrscr();
    char fname[20];
    student cse[3];    // declare array of 3 objects
    fstream fio;       // input and output file

    cout<<"Enter file name: ";

```

```

cin.get(fname, 20);

fio.open(fname, ios::in || ios::out);
if(!fio)
{
    cout<<"Error occurred in opening the file..!!\n";
    cout<<"Press any key to exit...\n";
    getch();
    exit(1);
}

cout<<"Enter details for 3 students:\n\n";
for(int i=0; i<3; i++)
{
    cse[i].getdata();
    fio.write((char *)&cse[i], sizeof(cse[i]));
}

fio.seekg(0);    /* seekg(0) resets the file to start, to access the file
                  * from the beginning */
cout<<"The content of "<<fname<<" file are shown below:\n";

for(i=0; i<3; i++)
{
    fio.read((char *)&cse[i], sizeof(cse[i]));
    cse[i].display();
}

fio.close();

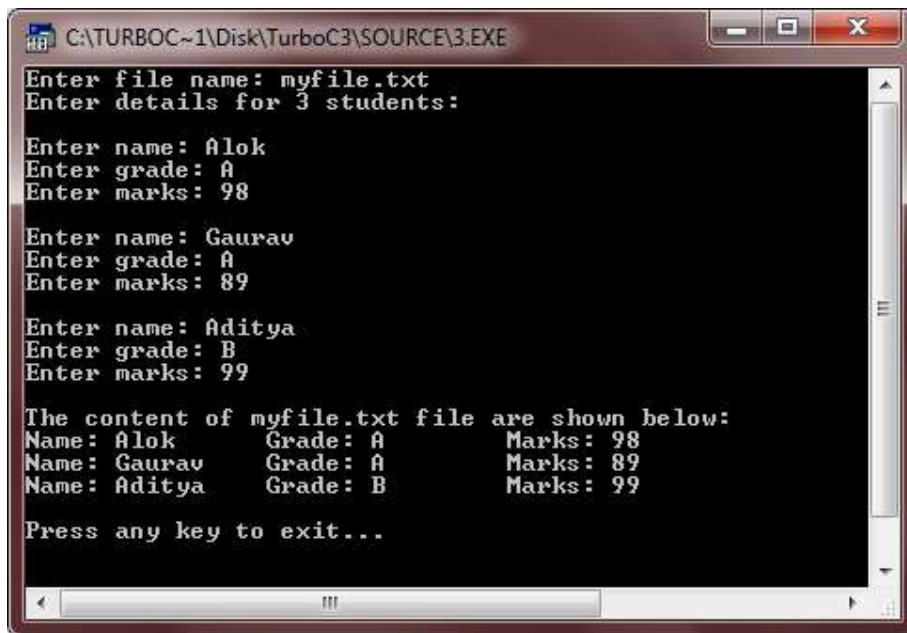
cout<<"\nPress any key to exit...\n";

getch();
}

```



Here is the sample run of the above C++ program:



```
C:\TURBOC~1\Disk\TurboC3\SOURCE\3.EXE
Enter file name: myfile.txt
Enter details for 3 students:

Enter name: Alok
Enter grade: A
Enter marks: 98

Enter name: Gaurav
Enter grade: A
Enter marks: 89

Enter name: Aditya
Enter grade: B
Enter marks: 99

The content of myfile.txt file are shown below:
Name: Alok      Grade: A      Marks: 98
Name: Gaurav    Grade: A      Marks: 89
Name: Aditya    Grade: B      Marks: 99

Press any key to exit...
```

## RANDOM ACCESS FILE: (File Pointers)

Using file streams, we can randomly access binary files. By random access, you can go to any position in the file as you wish (instead of going in a sequential order from the first character to the last)

Every file maintains two pointers called `get_pointer` (in input mode file) and `put_pointer` (in output mode file) which tells the current position in the file where reading or writing will take place. (A file pointer in this context is not like a C++ pointer but it works like a book-mark in a book.). These pointers help attain random access in file. That means moving directly to any location in the file instead of moving through it sequentially.

There may be situations where random access is the best choice. For example, if you have to modify a value in record no 21, then using random access techniques, you can place the file pointer at the beginning of record 21 and then straight-way process the record. If sequential access is used, then you'll have to unnecessarily go through first twenty records in order to reach at record 21.

### The `seekg()`, `seekp()`, `tellg()` and `tellp()` Functions

In C++, random access is achieved by manipulating `seekg()`, `seekp()`, `tellg()` and `tellp()` functions. The `seekg()` and `tellg()` functions allow you to set and examine the `get_pointer`, and the `seekp()` and `tellp()` functions perform these operations on the `put_pointer`.

The `seekg()` and `tellg()` functions are for input streams (`ifstream`) and `seekp()` and `tellp()` functions are for output streams (`ofstream`). However, if you use them with an `fstream` object then `tellg()` and `tellp()` return the same value. Also `seekg()` and `seekp()` work the same way in an `fstream` object. The most common forms of these functions are :

```
seekg()  ifstream & seekg(long);           Form 1
         ifstream & seekg(long, seek_dir); Form 2
```

```
seekp()  ofstream & seekp(long);           Form 1
         ofstream & seekp(long, seek_dir); Form 2
```

```
tellg()  long tellg()
```

tellp() long tellp()

The working of seekg() & seekp() and tellg() & tellp() is just the same except that seekg() and tellg() work for ifstream objects and seekp() and tellp() work for ofstream objects. In the above table, seek\_dir takes the definition enum seek\_dir { beg, cur, end};.

The seekg() or seekp(), when used according to Form 1, then it moves the get\_pointer or put\_pointer to an absolute position. Here is an example:

```
ifstream fin;
ofstream fout;
: // file opening routine
fin.seekg(30); // will move the get_pointer (in ifstream) to byte number 30 in
the file
fout.seekp(30); // will move the put_pointer (in ofstream) to byte
number 30 in the file
```

When seekg() or seekp() function is used according to Form 2, then it moves the get\_pointer or put\_pointer to a position relative to the current position, following the definition of seek\_dir. Since, seek\_dir is an enumeration defined in the header file iostream.h, that has the following values:

```
ios::beg, // refers to the beginning of the file
ios::cur, // refers to the current position in the file
ios::end} // refers to the end of the file
```

Here is an example.

```
fin.seekg(30, ios::beg); // go to byte no. 30 from beginning of file linked with
fin
fin.seekg(-2, ios::cur); // back up 2 bytes from the current position of get
pointer
fin.seekg(0, ios::end); // go to the end of the file
fin.seekg(-4, ios::end); // backup 4 bytes from the end of the file
```

The functions tellg() and tellp() return the position, in terms of byte number, of put\_pointer and get\_pointer respectively, in an output file and input file.

## **C++ File Pointers and Random Access Example**

Here is an example program demonstrating the concept of file pointers and random access in a C++ program:

```
/* C++ File Pointers and Random Access
 * This program demonstrates the concept
 * of file pointers and random access in
 * C++ */
```

```
#include<fstream.h>
```

```

#include<conio.h>
#include<stdlib.h>
#include<stdio.h>
#include<string.h>

class student
{
    int rollno;
    char name[20];
    char branch[3];
    float marks;
    char grade;

    public:
        void getdata()
        {
            cout<<"Rollno: ";
            cin>>rollno;
            cout<<"Name: ";
            cin>>name;
            cout<<"Branch: ";
            cin>>branch;
            cout<<"Marks: ";
            cin>>marks;

            if(marks>=75)
            {
                grade = 'A';
            }
            else if(marks>=60)
            {
                grade = 'B';
            }
            else if(marks>=50)
            {
                grade = 'C';
            }
            else if(marks>=40)
            {
                grade = 'D';
            }
            else
            {
                grade = 'F';
            }
        }

        void putdata()
        {
            cout<<"Rollno: "<<rollno<<"\tName: "<<name<<"\n";

```

```

        cout<<"Marks: "<<marks<<"\tGrade: "<<grade<<"\n";
    }

    int getrno()
    {
        return rollno;
    }

    void modify();
}stud1, stud;

void student::modify()
{
    cout<<"Rollno: "<<rollno<<"\n";
    cout<<"Name: "<<name<<"\tBranch: "<<branch<<"\tMarks:
"<<marks<<"\n";

    cout<<"Enter new details.\n";
    char nam[20]=" ", br[3]=" ";
    float mks;
    cout<<"New name:(Enter '.' to retain old one): ";
    cin>>nam;
    cout<<"New branch:(Press '.' to retain old one): ";
    cin>>br;
    cout<<"New marks:(Press -1 to retain old one): ";
    cin>>mks;

    if(strcmp(nam, ".")!=0)
    {
        strcpy(name, nam);
    }
    if(strcmp(br, ".")!=0)
    {
        strcpy(branch, br);
    }
    if(mks != -1)
    {
        marks = mks;
        if(marks>=75)
        {
            grade = 'A';
        }
        else if(marks>=60)
        {
            grade = 'B';
        }
        else if(marks>=50)
        {
            grade = 'C';
        }
    }
}

```

```

        else if(marks>=40)
        {
            grade = 'D';
        }
        else
        {
            grade = 'F';
        }
    }
}

void main()
{
    clrscr();

    fstream fio("marks.dat", ios::in | ios::out);
    char ans='y';
    while(ans=='y' || ans=='Y')
    {
        stud1.getdata();
        fio.write((char *)&stud1, sizeof(stud1));
        cout<<"Record added to the file\n";
        cout<<"\nWant to enter more ? (y/n).."
        cin>>ans;
    }

    clrscr();
    int rno;
    long pos;
    char found='f';

    cout<<"Enter rollno of student whose record is to be modified: ";
    cin>>rno;

    fio.seekg(0);
    while(!fio.eof())
    {
        pos = fio.tellg();
        fio.read((char *)&stud1, sizeof(stud1));
        if(stud1.getrno() == rno)
        {
            stud1.modify();
            fio.seekg(pos);
            fio.write((char *)&stud1, sizeof(stud1));
            found = 't';
            break;
        }
    }
    if(found=='f')
    {

```

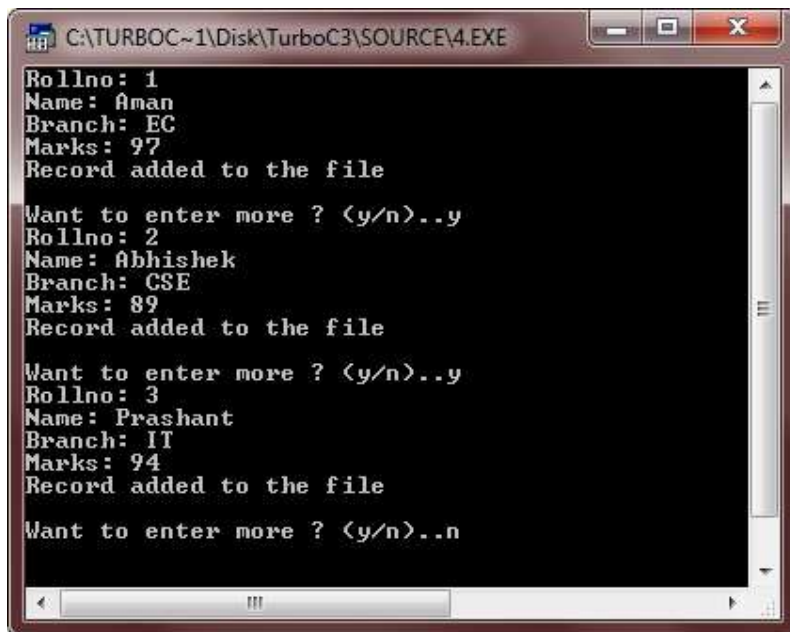
```

        cout<<"\nRecord not found in the file..!!\n";
        cout<<"Press any key to exit...\n";
        getch();
        exit(2);
    }

    fio.seekg(0);
    cout<<"Now the file contains:\n";
    while(!fio.eof())
    {
        fio.read((char *)&stud, sizeof(stud));
        stud.putdata();
    }
    fio.close();
    getch();
}

```

Here are the sample runs of the above C++ program:

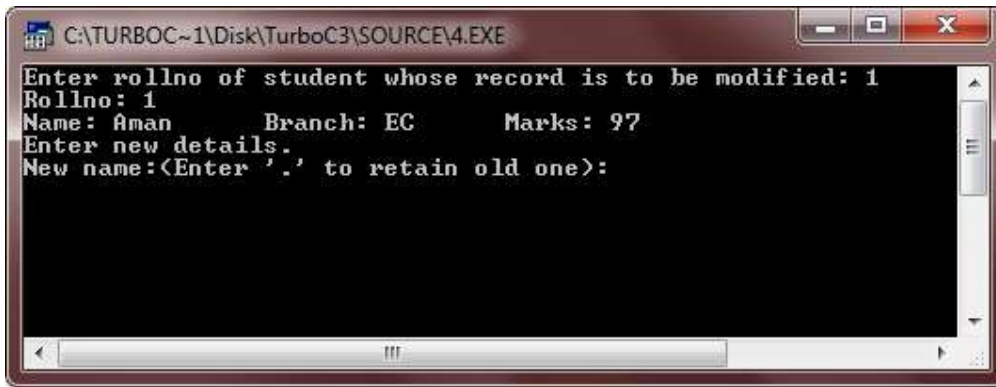


```

C:\TURBOC~1\Disk\TurboC3\SOURCE\4.EXE
Rollno: 1
Name: Aman
Branch: EC
Marks: 97
Record added to the file
Want to enter more ? (y/n)..y
Rollno: 2
Name: Abhishek
Branch: CSE
Marks: 89
Record added to the file
Want to enter more ? (y/n)..y
Rollno: 3
Name: Prashant
Branch: IT
Marks: 94
Record added to the file
Want to enter more ? (y/n)..n

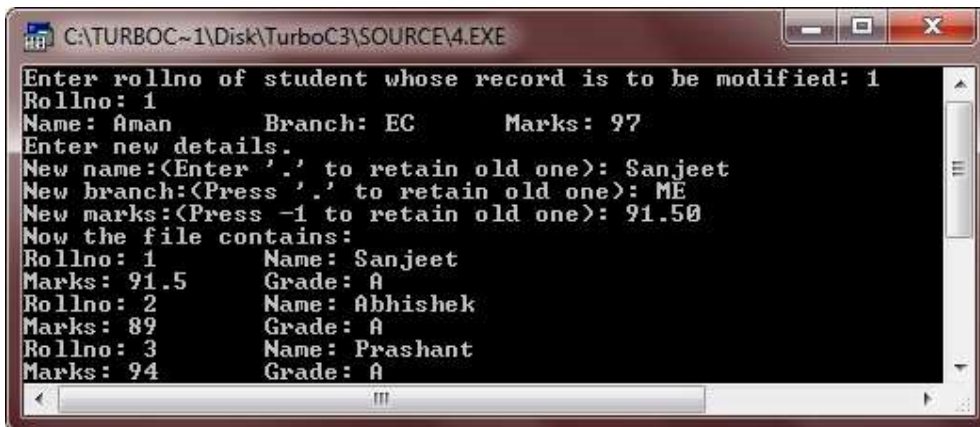
```

After entering the 3 records, just press n, then press enter. After performing this as shown above, press ENTER and here are the sample runs after pressing the ENTER button:



```
C:\TURBOC~1\Disk\TurboC3\SOURCE\4.EXE
Enter rollno of student whose record is to be modified: 1
Rollno: 1
Name: Aman Branch: EC Marks: 97
Enter new details.
New name:<Enter \'.\' to retain old one>:
```

Now enter the roll number of that student whose record is to be modified. Here we enter 1, and then enter the new information for that roll number as shown here in the above and below outputs:



```
C:\TURBOC~1\Disk\TurboC3\SOURCE\4.EXE
Enter rollno of student whose record is to be modified: 1
Rollno: 1
Name: Aman Branch: EC Marks: 97
Enter new details.
New name:<Enter \'.\' to retain old one>: Sanjeet
New branch:<Press \'.\' to retain old one>: ME
New marks:<Press -1 to retain old one>: 91.50
Now the file contains:
Rollno: 1 Name: Sanjeet
Marks: 91.5 Grade: A
Rollno: 2 Name: Abhishek
Marks: 89 Grade: A
Rollno: 3 Name: Prashant
Marks: 94 Grade: A
```



## Command Line Argument in C++

If any input value is passed through command prompt at the time of running of program is known as **command line argument**. It is a concept to passing the arguments to the main() function by using command prompt.

### When Use Command Line Argument

When you need to developing an application for DOS operating system then in that case command line arguments are used. DOS operating system is a command interface operating system so by using command we execute the program. With the help of command line arguments we can create our own commands.

In Command line arguments application main() function will takes two arguments that is;

- argc
- argv

**argc:** argc is an integer type variable and it holds total number of arguments which is passed into main function. It take Number of arguments in the command line including program name.

**argv[]:** argv[] is a char\* type variable, which holds actual arguments which is passed to main function.

### Compile and run CMD programs

Command line arguments are not compile and run like normal C++ programs, these programs are compile and run on command prompt. To Compile and Link Command Line Program we need TCC Command.

- First open command prompt
- Follow you directory where your code saved.
- For compile -> C:/TC/BIN>TCC mycmd.cpp
- For run -> C:/TC/BIN>mycmd 10 20
- **Explanation:** Here mycmd is your program file name and **TCC** is a Command. In "mycmd 10 20" statement we pass two arguments.

## Example of Command Line Argument in C++

```
#include<iostream.h>
#include<conio.h>

void main(int argc, char* argv[])
{
    int i;
    clrscr();
    cout<<"Total number of arguments: "<<argc;
    for(i=0;i< argc;i++)
    {
        cout<<endl<< i;<<"argument: "<<argv[i];
        getch();
    }
}
```

### Output

```
C:/TC/BIN>TCC mycmd.cpp
C:/TC/BIN>mycmd 10 20
Number of Arguments: 3
0 arguments c:/tc/bin/mycmd.exe
1 arguments: 10
2 arguments: 20
```

## C++ Templates:

Templates are powerful features of C++ which allows you to write generic programs. In simple terms, you can create a single function or a class to work with different data types using templates.

Templates are often used in larger codebase for the purpose of code reusability and flexibility of the programs.

The concept of templates can be used in two different ways:

- Function Templates
- Class Templates

### Function Templates

A function template works in a similar way to a normal function, with one key difference.

A single function template can work with different data types at once but, a single normal function can only work with one set of data types.

Normally, if you need to perform identical operations on two or more types of data, you use function overloading to create two functions with the required function declaration.

However, a better approach would be to use function templates because you can perform the same task writing less and maintainable code.

---

#### How to declare a function template?

A function template starts with the keyword **template** followed by template parameter/s inside **< >** which is followed by function declaration.

```
template <class T>
T someFunction(T arg)
{
    ... ..
}
```

In the above code,  $T$  is a template argument that accepts different data types (int, float) and **class** is a keyword.

You can also use keyword **typename** instead of class in the above example.

When, an argument of a data type is passed to someFunction( ), compiler generates a new version of someFunction() for the given data type.

---

### Example 1: Function Template to find the largest number

#### **Program to display largest among two numbers using function templates.**

// If two characters are passed to function template, character with larger ASCII value is displayed.

```
#include <iostream>
using namespace std;
// template function
template <class T>
T Large(T n1, T n2)
{
    return (n1 > n2) ? n1 : n2;
}
int main()
{
    int i1, i2;
    float f1, f2;
    char c1, c2;
    cout << "Enter two integers:\n";
    cin >> i1 >> i2;
    cout << Large(i1, i2) << " is larger." << endl;
    cout << "\nEnter two floating-point numbers:\n";
    cin >> f1 >> f2;
    cout << Large(f1, f2) << " is larger." << endl;
```

```
    cout << "\nEnter two characters:\n";  
    cin >> c1 >> c2;  
    cout << Large(c1, c2) << " has larger ASCII value.";  
    return 0;  
}
```

#### Output

Enter two integers:

5

10

10 is larger.

Enter two floating-point numbers:

12.4

10.2

12.4 is larger.

Enter two characters:

z

Z

z has larger ASCII value.

In the above program, a function template `Large()` is defined that accepts two arguments *n1* and *n2* of data type `T`. `T` signifies that argument can be of any data type.

`Large()` function returns the largest among the two arguments using a simple conditional operation.

Inside the `main()` function, variables of three different data types: `int`, `float` and `char` are declared. The variables are then passed to the `Large()` function template as normal functions.

During run-time, when an integer is passed to the template function, compiler knows it has to generate a `Large()` function to accept the `int` arguments and does so.

Similarly, when floating-point data and char data are passed, it knows the argument data types and generates the Large() function accordingly.

This way, using only a single function template replaced three identical normal functions and made your code maintainable.

## Example 2: Swap Data Using Function Templates

Program to swap data using function templates.

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

template <class T>
void Swap(T &n1, T &n2)
{
    T temp;
    temp = n1;
    n1 = n2;
    n2 = temp;
}

int main()
{
    int i1 = 1, i2 = 2;
    float f1 = 1.1, f2 = 2.2;
    char c1 = 'a', c2 = 'b';

    cout << "Before passing data to function template.\n";
    cout << "i1 = " << i1 << "\ni2 = " << i2;
    cout << "\nf1 = " << f1 << "\nf2 = " << f2;
    cout << "\nc1 = " << c1 << "\nc2 = " << c2;

    Swap(i1, i2);
    Swap(f1, f2);
```

```

        Swap(c1, c2);

    cout << "\n\nAfter passing data to function template.\n";
    cout << "i1 = " << i1 << "\ni2 = " << i2;
    cout << "\nf1 = " << f1 << "\nf2 = " << f2;
    cout << "\nc1 = " << c1 << "\nc2 = " << c2;

    return 0;
}

```

## Output

Before passing data to function template.

```

i1 = 1
i2 = 2
f1 = 1.1
f2 = 2.2
c1 = a
c2 = b

```

After passing data to function template.

```

i1 = 2
i2 = 1
f1 = 2.2
f2 = 1.1
c1 = b
c2 = a

```

In this program, instead of calling a function by passing a value, a call by reference is issued.

The Swap() function template takes two arguments and swaps them by reference.

## Class Templates

Like function templates, you can also create class templates for generic class operations.

Sometimes, you need a class implementation that is same for all classes, only the data types used are different.

Normally, you would need to create a different class for each data type OR create different member variables and functions within a single class.

This will unnecessarily bloat your code base and will be hard to maintain, as a change in one class/function should be performed on all classes/functions.

However, class templates make it easy to reuse the same code for all data types.

---

#### How to declare a class template?

```
template <class T>
class className
{
    ... ..
public:
    T var;
    T someOperation(T arg);
    ... ..
};
```

In the above declaration, T is the template argument which is a placeholder for the data type used.

Inside the class body, a member variable *var* and a member function *someOperation()* are both of type T.

#### How to create a class template object?

To create a class template object, you need to define the data type inside a < > when creation.

```
className<dataType> classObject;
```



For example:

```
className<int> classObject;  
className<float> classObject;  
className<string> classObject;
```

### Example 3: Simple calculator using Class template

Program to add, subtract, multiply and divide two numbers using class template

```
#include <iostream>  
using namespace std;  
  
template <class T>  
class Calculator  
{  
private:  
    T num1, num2;  
  
public:  
    Calculator(T n1, T n2)  
    {  
        num1 = n1;  
        num2 = n2;  
    }  
  
    void displayResult()  
    {  
        cout << "Numbers are: " << num1 << " and " << num2 << "." <<  
endl;  
        cout << "Addition is: " << add() << endl;  
        cout << "Subtraction is: " << subtract() << endl;  
        cout << "Product is: " << multiply() << endl;  
        cout << "Division is: " << divide() << endl;  
    }  
}
```

```

    T add() { return num1 + num2; }

    T subtract() { return num1 - num2; }

    T multiply() { return num1 * num2; }

    T divide() { return num1 / num2; }
};

int main()
{
    Calculator<int> intCalc(2, 1);
    Calculator<float> floatCalc(2.4, 1.2);

    cout << "Int results:" << endl;
    intCalc.displayResult();

    cout << endl << "Float results:" << endl;
    floatCalc.displayResult();

    return 0;
}

```

## Output

```

Int results:
Numbers are: 2 and 1.
Addition is: 3
Subtraction is: 1
Product is: 2
Division is: 2

```

```

Float results:

```

Numbers are: 2.4 and 1.2.

Addition is: 3.6

Subtraction is: 1.2

Product is: 2.88

Division is: 2

In the above program, a class template Calculator is declared.

The class contains two private members of type T: *num1* & *num2*, and a constructor to initialize the members.

It also contains public member functions to calculate the addition, subtraction, multiplication and division of the numbers which return the value of data type defined by the user. Likewise, a function `displayResult()` to display the final output to the screen.

In the `main()` function, two different Calculator objects `intCalc` and `floatCalc` are created for data types: `int` and `float` respectively. The values are initialized using the constructor.

Notice we use `<int>` and `<float>` while creating the objects. These tell the compiler the data type used for the class creation.

This creates a class definition each for `int` and `float`, which are then used accordingly.

Then, `displayResult()` of both objects is called which performs the Calculator operations and displays the output.

# EXCEPTION HANDLING

An exception is a problem that arises during the execution of a program. A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

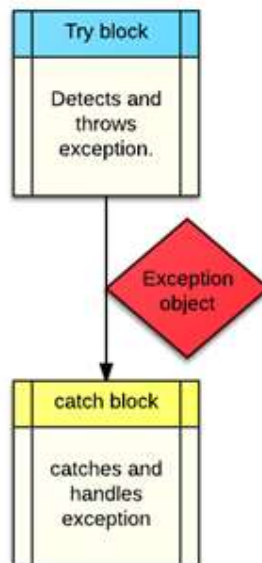
## What is an exception?

Exception is an event that happens when unexpected circumstances appear. It can be a runtime error or we can create an exceptional situation programmatically.

Exception handling consists in transferring control from the place where exception happened to the special functions (commands) called handlers.

## How to handle exceptions?

Exceptions are handled by using **try/catch** block. The code that can produce an exception is surrounded with **try** block. The handler for this exception is placed in **catch** block:



Exceptions provide a way to transfer control from one part of a program to another. C++ exception handling is built upon three keywords: **try**, **catch**, and **throw**.

- **throw:** A program throws an exception when a problem shows up. This is done using a **throw** keyword.
- **catch:** A program catches an exception with an exception handler at the place in a program where we want to handle the problem. The **catch** keyword indicates the catching of an exception.
- **try:** A **try** block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.

Assuming a block will raise an exception, a method catches an exception using a combination of the **try** and **catch** keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following:

```
try
{
    // protected code
}catch( ExceptionName e1 )
{
    // catch block
}catch( ExceptionName e2 )
{
    // catch block
}catch( ExceptionName eN )
{
    // catch block
}
```

### **Advantage**

It maintains the normal flow of the application. In such case, rest of the code is executed even after exception.

### **Example:**

```
#include <iostream>
```

```

using namespace std;

double division(int a, int b) {
    if( b == 0 ) {
        throw "Division by zero condition!";
    }
    return (a/b);
}

int main () {
    int x = 50;
    int y = 0;
    double z = 0;

    try {
        z = division(x, y);
        cout << z << endl;
    }catch (const char* msg) {
        cerr << msg << endl;
    }

    return 0;
}

```

#### EXAMPLE 2:

```

//program demo using exception handling
#include<iostream>
using namespace std;
double division(int a,int b)
{
    if( b == 0 ) {

```

```

        throw "Division by zero condition!";
    }
    return (a/b);

}

int main()
{
    int x = 50;
    int y = 0;
    double z = 0;

    try {
        z = division(x, y);
        cout << z << endl;

    }catch (const char* msg) {
        cerr << msg << endl;
    }

    int sum = x+y;
    cout<<"Sum is = "<<sum<<endl;
    return 0;
}

```

### **Define New Exceptions**

We can define our own exceptions by inheriting and overriding exception class functionality. Following is the example, which shows how we can use `std::exception` class to implement our own exception in standard way:

```

//program demo to create new exception
#include <iostream>
#include <exception>
using namespace std;
struct MyException : public exception {

```

```

const char * what () const throw () {
    return "C++ Exception";
}
};

int main() {
    try {
        throw MyException();
    } catch(MyException& e) {
        std::cout << "MyException caught" << std::endl;
        std::cout << e.what() << std::endl;
    } catch(std::exception& e) {
        //Other errors
    }
    return 0;
}

```

This would produce the following result:

MyException caught

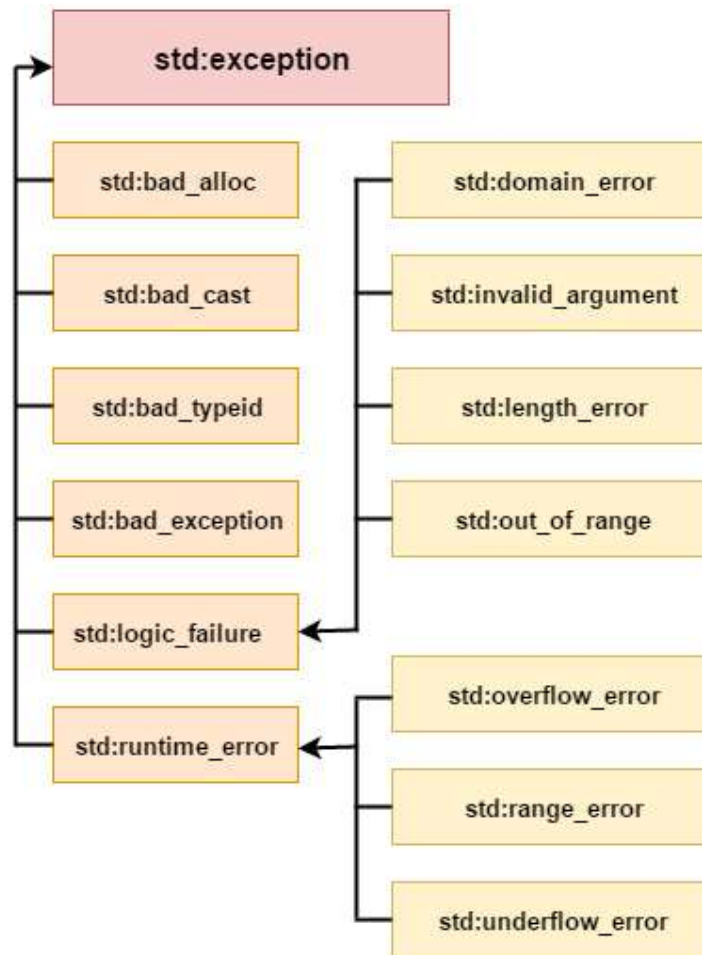
C++ Exception

Here, what () is a public method provided by exception class and it has been overridden by all the child exception classes. This returns the cause of an exception.

## C++ Exception Classes

In C++ standard exceptions are defined in <exception> class that we can use inside our programs. The arrangement of parent-child class hierarchy is shown below:





All the exception classes in C++ are derived from `std::exception` class. Let's see the list of C++ common exception classes.

Exception	Description
<code>std::exception</code>	It is an exception and parent class of all standard C++ exceptions.
<code>std::logic_failure</code>	It is an exception that can be detected by reading a code.
<code>std::runtime_error</code>	It is an exception that cannot be detected by reading a code.
<code>std::bad_exception</code>	It is used to handle the unexpected exceptions in a c++ program.

<code>std::bad_cast</code>	This exception is generally be thrown by <b>dynamic_cast</b> .
<code>std::bad_typeid</code>	This exception is generally be thrown by <b>typeid</b> .
<code>std::bad_alloc</code>	This exception is generally be thrown by <b>new</b> .

---

## C++ Exception Handling Keywords

In C++, we use 3 keywords to perform exception handling:

- try
- catch, and
- throw

### C++ example without try/catch

```
#include <iostream>

using namespace std;

float division(int x, int y) {
    return (x/y);
}

int main () {
    int i = 50;
    int j = 0;
    float k = 0;

    k = division(i, j);

    cout << k << endl;

    cout<<"Sum = "<<(i+j)<<endl;

    return 0;
}
```

Output:

Floating point exception (core dumped)

### **C++ try/catch example**

```
//program demo using try & catch

#include <iostream>

using namespace std;

float division(int x, int y) {

    if( y == 0 ) {

        throw "Attempted to divide by zero!";

    }

    return (x/y);

}

int main () {

    int i = 25;

    int j = 0;

    float k = 0;

    try {

        k = division(i, j);

        cout << k << endl;

    }catch (const char* e) {

        cerr << e << endl;

    }

    cout<<"Sum = "<<(i+j)<<endl;

    return 0;

}
```

Output:

Attempted to divide by zero!

## C++ User-Defined Exceptions

The new exception can be defined by overriding and inheriting **exception** class functionality.

### C++ user-defined exception example

Let's see the simple example of user-defined exception in which **std::exception** class is used to define the exception.

```
#include <iostream>

#include <exception>

using namespace std;

class MyException : public exception{
    public:
        const char * what() const throw()
        {
            return "Attempted to divide by zero!\n";
        }
};

int main()
{
    try
    {
        int x, y;

        cout << "Enter the two numbers : \n";

        cin >> x >> y;

        if (y == 0)
```

```

    {
        MyException z;
        throw z;
    }
else
    {
        cout << "x / y = " << x/y << endl;
    }
}
catch(exception& e)
{
    cout << e.what();
}
return 0;
}

```

Output:

Enter the two numbers :

10

2

x / y = 5

Output:

Enter the two numbers :

10

0

Attempted to divide by zero!

-->

**Note:** In above example what() is a public method provided by the exception class. It is used to return the cause of an exception.

## C++ STRINGS:



String is a collection of characters.

C++ provides following two types of string representations:

- The C-style character string.
- The string class type introduced with Standard C++.

### **C-strings**

In C programming, the collection of characters is stored in the form of arrays, this is also supported in C++ programming. Hence, it's called C-strings.

C-strings are arrays of type `char` terminated with null character, that is, `\0` (ASCII value of null character is 0).

#### [How to define a C-string?](#)

```
char str[] = "C++";
```

In the above code, `str` is a string and it holds 4 characters.

Although, "C++" has 3 character, the null character `\0` is added to the end of the string automatically.

## The C-Style Character String

The C-style character string originated within the C language and continues to be supported within C++. This string is actually a one-dimensional array of characters which is terminated by a **null** character '\0'. Thus a null-terminated string contains the characters that comprise the string followed by a **null**.

The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello."

```
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

If we follow the rule of array initialization, then we can write the above statement as follows:

```
char greeting[] = "Hello";
```

Following is the memory presentation of above defined string in C/C++:

Index	0	1	2	3	4	5
Variable	H	e	l	l	o	\0
Address	0x23451	0x23452	0x23453	0x23454	0x23455	0x23456

Actually, we do not place the null character at the end of a string constant. The C++ compiler automatically places the '\0' at the end of the string when it initializes the array. Let us try to print above-mentioned string:

```
#include <iostream>
using namespace std;
int main () {
    char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};

    cout << "Greeting message: ";
    cout << greeting << endl;

    return 0;
```

```
}
```

When the above code is compiled and executed, it produces result something as follows:

Greeting message: Hello

Example : C++ String to read a word

**C++ program to display a string entered by user.**

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

int main()
{
    char str[100];

    cout << "Enter a string: ";
    cin >> str;
    cout << "You entered: " << str << endl;

    cout << "\nEnter another string: ";
    cin >> str;
    cout << "You entered: " << str << endl;

    return 0;
}
```

## Output

Enter a string: C++  
You entered: C++

Enter another string: Programming is fun.  
You entered: Programming

Notice that, in the second example only "Programming" is displayed instead of "Programming is fun".

This is because the extraction operator >> works as scanf() in C and considers a space " " has a terminating character.



## //PROGRAM DEMO USING STRINGS

```
#include<iostream>
using namespace std;
int main()
{
    char firstname[50];
    char lastname[50];
    cout<<"Enter first name: ";
    cin.get(firstname,50);
    cin.get();
    cout<<"Enter last name: ";
    cin.get(lastname,50);
    cout<<"The given string is: "<<firstname<<" "<<lastname;
    return 0;
}
```

## C++ String to read a line of text

### C++ program to read and display an entire line entered by user.

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

int main()
{
    char str[100];
    cout << "Enter a string: ";
    cin.get(str, 100);

    cout << "You entered: " << str << endl;
    return 0;
}
```

## Output

Enter a string: Programming is fun.  
You entered: Programming is fun.

To read the text containing blank space, cin.get function can be used. This function takes two arguments.

First argument is the name of the string (address of first element of string) and second argument is the maximum size of the array.

In the above program, *str* is the name of the string and 100 is the maximum size of the array.

C++ supports a wide range of functions that manipulate null-terminated strings:

S.N.	Function & Purpose
1	<b>strcpy(s1, s2);</b> Copies string s2 into string s1.
2	<b>strcat(s1, s2);</b> Concatenates string s2 onto the end of string s1.
3	<b>strlen(s1);</b> Returns the length of string s1.
4	<b>strcmp(s1, s2);</b> Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.
5	<b>strchr(s1, ch);</b> Returns a pointer to the first occurrence of character ch in string s1.
6	<b>strstr(s1, s2);</b> Returns a pointer to the first occurrence of string s2 in string s1.

//PROGRAM DEMO USING STRINGS

```
#include<iostream>
#include<cstring>
using namespace std;
int main()
{
    char name1[100];
    char name2[100];
    cout<<"Enter your name: ";
    cin.get(name1,100);
    strcpy(name2,name1);
    cout<<"Your name is: "<<name2<<endl;
```

```

    return 0;
}
//PROGRAM DEMO USING STRINGS
#include<iostream>
#include<cstring>
using namespace std;
int main()
{
    char name1[100];
    char name2[100];
    cout<<"Enter first name: ";
    cin.get(name1,100);
    cin.get();
    cout<<"Enter last name: ";
    cin.get(name2,100);
    cout << "strcat( name1, name2): " << strcat(name1,name2) << endl;

    return 0;
}

```

```

//program demo for string concatenation
#include <iostream>
#include <cstring>
using namespace std;
int main () {
    char str1[10] = "Hello";
    char str2[10] = "World";
    char str3[10];
    int len ;

    // copy str1 into str3
    strcpy( str3, str1);
    cout << "strcpy( str3, str1) : " << str3 << endl;

    // concatenates str1 and str2
    strcat( str1, str2);
    cout << "strcat( str1, str2): " << str1 << endl;
}

```

```

// total length of str1 after concatenation
len = strlen(str1);
cout << "strlen(str1): " << len << endl;

return 0;
}

```

When the above code is compiled and executed, it produces result something as follows:

```

strcpy( str3, str1) : Hello
strcat( str1, str2): HelloWorld
strlen(str1) : 10

```

## 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 following example:

At this point, you may not understand this example because so far we have not discussed Classes and Objects. So can have a look and proceed until you have understanding on Object Oriented Concepts.

```

//program demo using string object

#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
```

```
//PROGRAM DEMO USING STRINGS
```

```
#include<iostream>
```

```
#include<cstring>
```

```
using namespace std;
```

```
int main()
```

```
{
```

```
    char password[100];
```

```
    cout<<"Enter password: ";
```

```
    cin.get(password,50);
```

```
    if(strlen(password)<10)
```

```
    {
```

```
        cout<<"Please check the password.\nPassword must be at least 10 or more characters"<<endl;
```

```
    }
```

```
    else
```

```
    {
```

```
        cout<<"Welcome to Sai Pali..."<<endl;
```

```
    }
```

```
    return 0;
```

```
}
```

```
//program demo to echo password character as '*'
```

```
#include<iostream>
```

```
#include<string>
```

```
#include<conio.h>
```

```

#include<cstring>
using namespace std;
int main(){
    string res;
    char c;
    cout<<"Enter your password:";
    do{
        c = getch();
        switch(c){
            case 0://special keys. like: arrows, f1-12 etc.
                getch();//just ignore also the next character.
                break;
            case 13://enter
                cout<<endl;
                break;
            case 8://backspace
                if(res.length()>0){
                    res.erase(res.end()-1); //remove last character from string
                    cout<<c<<' '<<c;//go back, write space over the character and back
again.
                }
                break;
            default://regular ascii
                res += c;//add to string
                cout<<'*';//print `*`
                break;
        }
    }while(c!=13);
    //print result:
    //cout<<res<<endl;

    cout<<"Your password is: "<<res<<endl;

```

```

if(res.length()<10)
{
    cout<<"Please check the password.\nPassword must be at least 10 or more
characters"<<endl;
}
else
{
    cout<<"Welcome to Sai Pali..."<<endl;
}
return 0;
}

```

## string Object

In C++, we can also create a string object for holding strings.

Unlike using char arrays, string objects has no fixed length, and can be extended as per our requirement.

### Example : C++ string using string data type

```

#include <iostream>
using namespace std;

int main()
{
    // Declaring a string object
    string str;
    cout << "Enter a string: ";
    getline(cin, str);

    cout << "You entered: " << str << endl;
    return 0;
}

```

## Output

```

Enter a string: Programming is fun.
You entered: Programming is fun.

```

In this program, a string *str* is declared. Then the string is asked from the user.

Instead of using `cin>>` or `cin.get()` function, you can get the entered line of text using `getline()`.

`getline()` function takes the input stream as the first parameter which is `cin` and `str` as the location of the line to be stored.

### Passing String to a Function

Strings are passed to a function in a similar way arrays are passed to a function.

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

void display(char s[]);

int main()
{
    char str[100];
    string str1;

    cout << "Enter a string: ";
    cin.get(str, 100);

    cout << "Enter another string: ";
    getline(cin, str1);

    display(str);
    display(str1);

    return 0;
}

void display(char s[])
{
```



```

    cout << "You entered char array: " << s <<;
}

void display(string s)
{
    cout << "You entered string: " << s << endl;
}

```

Output

```

Enter a string: Programming is fun.
Enter another string: Programming is fun.
You entered char array: Programming is fun.
You entered string: Programming is fun.

```

In the above program, two strings are asked to enter. These are stored in *str* and *str1* respectively, where *str* is a char array and *str1* is a string object.

Then, we have two functions `display()` that outputs the string onto the string.

The only difference between the two functions is the parameter. The first `display()` function takes char array as a parameter, while the second takes string as a parameter.

This process is known as function overloading.

### **Example 1: Find Frequency of Characters of a String Object**

```

#include <iostream>
using namespace std;
int main()
{
    string str = "C++ Programming is awesome";
    char checkCharacter = 'a';
    int count = 0;

    for (int i = 0; i < str.size(); i++)
    {
        if (str[i] == checkCharacter)

```

```

    {
        ++ count;
    }
}

cout << "Number of " << checkCharacter << " = " << count;

return 0;
}

```

## Output

Number of a = 2

## C++ Program to Find the Number of Vowels, Consonants, Digits and White Spaces in a String

//program demo that takes a string object from the user and calculates the number of vowels, consonants, digits and white-spaces

```

#include <iostream>
using namespace std;

int main()
{
    string line;
    int vowels, consonants, digits, spaces;

    vowels = consonants = digits = spaces = 0;

    cout << "Enter a line of string: ";
    getline(cin, line);

    for(int i = 0; i < line.length(); ++i)
    {
        if((line[i]=='a' || line[i]=='e' || line[i]=='i' ||
            line[i]=='o' || line[i]=='u' || line[i]=='A' ||
            line[i]=='E' || line[i]=='I' || line[i]=='O' ||
            line[i]=='U'))
        {
            ++vowels;
        }
        else if((line[i]>='a' && line[i]<='z') || (line[i]>='A' && line[i]<='Z'))
        {
            ++consonants;
        }
        else if(line[i]>='0' && line[i]<='9')
        {

```

```

        ++digits;
    }
    else if (line[i]==' ')
    {
        ++spaces;
    }
}

cout << "Vowels: " << vowels << endl;
cout << "Consonants: " << consonants << endl;
cout << "Digits: " << digits << endl;
cout << "White spaces: " << spaces << endl;

return 0;
}

```

## Output

Enter a line of string: I have 2 C++ programming books.  
Vowels: 8  
Consonants: 14  
Digits: 1  
White spaces: 5

## C++ Program to Copy Strings

### Example 1: Copy String Object

```

#include <iostream>
using namespace std;

int main()
{
    string s1, s2;

    cout << "Enter string s1: ";
    getline (cin, s1);

    s2 = s1;

    cout << "s1 = " << s1 << endl;
    cout << "s2 = " << s2;

    return 0;
}

```

```
Enter string s1: C++ Strings
s1 = C++ Strings
s2 = C++ Strings
```

To copy c-strings in C++, strcpy() function is used.

---

### **Example 1: Copy C-Strings**

```
#include <iostream>
#include <cstring>

using namespace std;

int main()
{
    char s1[100], s2[100];

    cout << "Enter string s1: ";
    cin.getline(s1, 100);

    strcpy(s2, s1);

    cout << "s1 = " << s1 << endl;
    cout << "s2 = " << s2;

    return 0;
}
```

### **Output**

```
Enter string s1: C-Strings
s1 = C-Strings
s2 = C-Strings
```

## [C++ Program to Concatenate Two Strings](#)

### **Example 1: Concatenate String Objects**

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

int main()
{
    string s1, s2, result;
```

```

    cout << "Enter string s1: ";
    getline (cin, s1);

    cout << "Enter string s2: ";
    getline (cin, s2);

    result = s1 + s2;

    cout << "Resultant String = "<< result;

    return 0;
}

```

//program demo for comparing and swapping strings

```

#include<iostream>
#include<string>
using namespace std;
int main()
{
    string s1("Road");
    string s2("Read");
    string s3("Red");
    cout<<"s1 = "<<s1<<endl;
    cout<<"s2 = "<<s2<<endl;
    cout<<"s3 = "<<s3<<endl;
    int x = s1.compare(s2);
    if(x==0)
        cout<<"s1 == s2"<<endl;
    else if (x>0)
        cout<<"s1 > s2"<<endl;
    else
        cout<<"s1<s2"<<endl;
    cout<<"\nBefore swap: "<<endl;
    cout<<"s1 = "<<s1<<endl;

```

```
cout<<"s2 = "<<s2<<endl;
s1.swap(s2);
cout<<"\nAfter swapping: "<<endl;
cout<<"s1 = "<<s1<<endl;
cout<<"s2 = "<<s2<<endl;
return 0;
}
```

## C++ DYNAMIC MEMORY:

A good understanding of how dynamic memory really works in C++ is essential to becoming a good C++ programmer. Memory in our C++ program is divided into two parts:

- **The stack:** All variables declared inside the function will take up memory from the stack.
- **The heap:** This is unused memory of the program and can be used to allocate the memory dynamically when program runs.

Many times, we are not aware in advance how much memory we will need to store particular information in a defined variable and the size of required memory can be determined at run time.

We can allocate memory at run time within the heap for the variable of a given type using a special operator in C++ which returns the address of the space allocated. This operator is called **new** operator.

If we are not in need of dynamically allocated memory anymore, we can use **delete** operator, which de-allocates memory previously allocated by new operator.

### The new and delete operators

There is following generic syntax to use **new** operator to allocate memory dynamically for any data-type.

```
new data-type;
```

Here, **data-type** could be any built-in data type including an array or any user defined data types include class or structure. Let us start with built-in data types. For example, we can define a pointer to type double and then request that the memory be allocated at execution time. We can do this using the **new** operator with the following statements:

```
double* pvalue = NULL; // Pointer initialized with null  
pvalue = new double; // Request memory for the variable
```

The memory may not have been allocated successfully, if the free store had been used up. So it is good practice to check if new operator is returning NULL pointer and take appropriate action as below:

```
double* pvalue = NULL;
if( !(pvalue = new double )) {

    cout << "Error: out of memory." <<endl;
    exit(1);
}
```

The **malloc()** function from C, still exists in C++, but it is recommended to avoid using malloc() function. The main advantage of new over malloc() is that new doesn't just allocate memory, it constructs objects which is prime purpose of C++.

At any point, when we feel a variable that has been dynamically allocated is not anymore required, we can free up the memory that it occupies in the free store with the delete operator as follows:

```
delete pvalue;      // Release memory pointed to by pvalue
```

Let us put above concepts and form the following example to show how new and delete work:

```
//program demo using new and delete

#include <iostream>
using namespace std;

int main () {
    double* pvalue = NULL; // Pointer initialized with null
    pvalue = new double;   // Request memory for the variable
    *pvalue = 29494.99;    // Store value at allocated address
    cout << "Value of pvalue : " << *pvalue << endl;

    delete pvalue;        // free up the memory.
```



```
    return 0;
}
```

If we compile and run above code, this would produce the following result:

Value of pvalue : 29495

## **Dynamic Memory Allocation for Arrays**

Consider you want to allocate memory for an array of characters, i.e., string of 20 characters. Using the same syntax what we have used above we can allocate memory dynamically as shown below.

```
char* pvalue = NULL; // Pointer initialized with null
pvalue = new char[20]; // Request memory for the variable
```

To remove the array that we have just created the statement would look like this:

```
delete [] pvalue; // Delete array pointed to by pvalue
```

Following is the syntax of new operator for a multi-dimensional array as follows:

```
int ROW = 2;
int COL = 3;
double **pvalue = new double* [ROW]; // Allocate memory for rows

// Now allocate memory for columns
for(int i = 0; i < COL; i++) {
    pvalue[i] = new double[COL];
}
```

The syntax to release the memory for multi-dimensional will be as follows:

```
for(int i = 0; i < ROW; i++) {
    delete[] pvalue[i];
}
delete [] pvalue;
```

**EXAMPLE:**

```
//program demo using dynamic memory allocation using arrays
#include<iostream>
using namespace std;
int main()
{
    int* pvalue = NULL; // Pointer initialized with null
    pvalue = new int[10]; // Request memory for the variable
    cout<<"Enter any 5 integer number: "<<endl;
    for(int i=0;i<5;i++)
    {
        cin>>pvalue[i];
    }
    cout<<"The array elements are: "<<endl;
    for(int i=0;i<5;i++)
    {
        cout<<pvalue[i];
    }
    delete [] pvalue;
    return 0;
}
```

**Dynamic Memory Allocation for Objects**

Objects are no different from simple data types. For example, consider the following code where we are going to use an array of objects to clarify the concept:

```
#include <iostream>
using namespace std;
class Box {
public:
    Box() {
        cout << "Constructor called!" <<endl;
    }

    ~Box() {
```

```

        cout << "Destructor called!" <<endl;
    }
};

int main( ) {
    Box* myBoxArray = new Box[4];

    delete [] myBoxArray; // Delete array

    return 0;
}

```

If you were to allocate an array of four Box objects, the Simple constructor would be called four times and similarly while deleting these objects, destructor will also be called same number of times.

If we compile and run above code, this would produce the following result:

```

Constructor called!
Constructor called!
Constructor called!
Constructor called!
Destructor called!
Destructor called!
Destructor called!
Destructor called!

```

## NAMESPACES IN C++:

Consider a situation, when we have two persons with the same name, John, in the same class. Whenever we need to differentiate them definitely we would have to use some additional information along with their name, like either the area if they live in different area or their mother or father name, etc.

Same situation can arise in our C++ applications. For example, we might be writing some code that has a function called xyz() and there is another library available which

is also having same function xyz(). Now the compiler has no way of knowing which version of xyz() function we are referring to within our code.

A **namespace** is designed to overcome this difficulty and is used as additional information to differentiate similar functions, classes, variables etc. with the same name available in different libraries. Using namespace, we can define the context in which names are defined. **In essence, a namespace defines a scope.**

### Defining a Namespace:

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

```
namespace namespace_name {  
    // code declarations  
}
```

To call the namespace-enabled version of either function or variable, prepend the namespace name as follows:

```
name::code; // code could be variable or function.
```

Let us see how namespace scope the entities including variable and functions:

```
//program demo for creating namespaces.
```

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

```
// first name space  
namespace first_space {  
    void show() {  
        cout << "Inside first_space" << endl;  
    }  
}
```

```
// second name space
```

```

namespace second_space {
    void show() {
        cout << "Inside second_space" << endl;
    }
}

```

```

int main () {

    // Calls function from first name space.
    first_space :: show();

    // Calls function from second name space.
    second_space::show();

    return 0;
}

```

If we compile and run above code, this would produce the following result:

```

Inside first_space
Inside second_space

```

Example 2:

```

//program demo to find the area and perimeter of a circle using namespace
#include<iostream>
using namespace std;
namespace first
{
    void show()
    {
        int radius;
        cout<<"Enter radius:";
        cin>>radius;
        float area = 3.142 * radius * radius;
        cout<<"The area of circle is : "<<area<<endl;
    }
}

```

```

    }
}
namespace second
{
    void show()
    {
        int radius;
        cout<<"Enter radius: ";
        cin>>radius;
        float perimeter = 2 * 3.142 * radius;
        cout<<"The perimeter is : "<<perimeter<<endl;
    }
}
int main()
{
    first::show();
    second::show();
    return 0;
}

```

## The using directive

We can also avoid prepending of namespaces with the **using namespace** directive. This directive tells the compiler that the subsequent code is making use of names in the specified namespace. The namespace is thus implied for the following code:

```

//program demo to create namespaces

#include <iostream>
using namespace std;

// first name space
namespace first_space{
    void func(){
        cout << "Inside first_space" << endl;
    }
}

```

```

    }
}

// second name space
namespace second_space{
    void func(){
        cout << "Inside second_space" << endl;
    }
}

using namespace first_space;
int main () {

    // This calls function from first name space.
    func();

    return 0;
}

```

If we compile and run above code, this would produce the following result:

Inside first\_space

Example 2:

//program demo to find the area and perimeter of a circle using namespace

```

#include<iostream>
using namespace std;
namespace first
{
    void show()
    {
        int radius;
        cout<<"Enter radius:";
        cin>>radius;
        float area = 3.142 * radius * radius;
    }
}

```

```

        cout<<"The area of circle is : "<<area<<endl;
    }
}
namespace second
{
    void show()
    {
        int radius;
        cout<<"Enter radius: ";
        cin>>radius;
        float perimeter = 2 * 3.142 * radius;
        cout<<"The perimeter is : "<<perimeter<<endl;
    }
}
int main()
{
    {
        using namespace first;
        {
            show();
        }
    }
    {
        using namespace second;
        {
            show();
        }
    }
    return 0;
}

```

Example 3:

```

#include<iostream>
using namespace std;

```



```

namespace myconstants {
    const double pi = 3.141592;
}

namespace myshapes {
    double area;
    double perimeter;

    void AreaOfCircle(double radius)
    {
        area = myconstants::pi * radius * radius;
    }

    void PerimeterOfCircle(double radius)
    {
        perimeter = 2 * myconstants::pi * radius;
    }
}

int main(int argc, char * argv[])
{
    double r;
    cout << endl << "Enter Radius:";
    cin >> r;
    myshapes::AreaOfCircle(r);
    cout << endl << "Area of the Circle is :" << myshapes::area;
    myshapes::PerimeterOfCircle(r);
    cout << endl << "Perimeter of the Circle is :" << myshapes::perimeter;
}

```

The using directive can also be used to refer to a particular item within a namespace. For example, if the only part of the std namespace that you intend to use is cout, you can refer to it as follows:

```
using std::cout;
```

Subsequent code can refer to `cout` without prepending the namespace, but other items in the **std** namespace will still need to be explicit as follows:

```
#include <iostream>
```

```
using std::cout;
```

```
int main () {
```

```
    cout << "std::endl is used with std!" << std::endl;
```

```
    return 0;
```

```
}
```

If we compile and run above code, this would produce the following result:

```
std::endl is used with std!
```

Names introduced in a **using** directive obey normal scope rules. The name is visible from the point of the **using** directive to the end of the scope in which the directive is found. Entities with the same name defined in an outer scope are hidden.

## Discontiguous Namespaces

A namespace can be defined in several parts and so a namespace is made up of the sum of its separately defined parts. The separate parts of a namespace can be spread over multiple files.

So, if one part of the namespace requires a name defined in another file, that name must still be declared. Writing a following namespace definition either defines a new namespace or adds new elements to an existing one:

```
namespace namespace_name {
```

```
    // code declarations
```

```
}
```

## Nested Namespaces

Namespaces can be nested where you can define one namespace inside another namespace as follows:

```
namespace namespace_name1 {  
    // code declarations  
    namespace namespace_name2 {  
        // code declarations  
    }  
}
```

You can access members of nested namespace by using resolution operators as follows:

```
// to access members of namespace_name2  
using namespace namespace_name1::namespace_name2;
```

```
// to access members of namespace_name1  
using namespace namespace_name1;
```

In the above statements if you are using namespace\_name1, then it will make elements of namespace\_name2 available in the scope as follows:

```
#include <iostream>  
using namespace std;  
  
// first namespace  
namespace first_space{  
    void func(){  
        cout << "Inside first_space" << endl;  
    }  
  
    // second namespace  
    namespace second_space{  
        void func(){  
            cout << "Inside second_space" << endl;  
        }  
    }  
}
```

```
    }  
  }  
}
```

```
using namespace first_space::second_space;  
int main () {  
  
    // This calls function from second name space.  
    func();  
  
    return 0;  
}
```

If we compile and run above code, this would produce the following result:

Inside second\_space

## using

The keyword using introduces a name into the current declarative region (such as a block), thus avoiding the need to qualify the name. For example:

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

```
namespace first  
{  
int x = 5;  
int y = 10;  
}
```

```
namespace second  
{  
double x = 3.1416;
```

```
double y = 2.7183;
```

```
}
```

```
int main () {
```

```
using first::x;
```

```
using second::y;
```

```
cout << x << '\n';
```

```
cout << y << '\n';
```

```
cout << first::y << '\n';
```

```
cout << second::x << '\n';
```

```
return 0;
```

```
}
```

## C++ Preprocessor:

The preprocessors are the directives, which give instruction to the compiler to preprocess the information before actual compilation starts.

All preprocessor directives begin with `#`, and only white-space characters may appear before a preprocessor directive on a line. Preprocessor directives are not C++ statements, so they do not end in a semicolon (`;`).

We already have seen a **#include** directive in all the examples. This macro is used to include a header file into the source file.

There are number of preprocessor directives supported by C++ like `#include`, `#define`, `#if`, `#else`, `#line`, etc. Let us see important directives:

### The #define Preprocessor

The `#define` preprocessor directive creates symbolic constants. The symbolic constant is called a **macro** and the general form of the directive is:

```
#define macro-name replacement-text
```

When this line appears in a file, all subsequent occurrences of macro in that file will be replaced by replacement-text before the program is compiled. For example:

```
#include <iostream>
#define PI 3.14159
using namespace std;
int main () {
    cout << "Value of PI :" << PI << endl;
    return 0;
}
```

Write a program to find the area and perimeter of a circle using preprocessor directive.

### Function-Like Macros

You can use `#define` to define a macro which will take argument as follows:

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

#define MIN(a,b) (((a)<(b)) ? a : b)

int main () {
    int i, j;
    i = 100;
    j = 30;

    cout <<"The minimum is " << MIN(i, j) << endl;

    return 0;
}
```

If we compile and run above code, this would produce the following result:

The minimum is 30

## Conditional Compilation

There are several directives, which can use to compile selectively portions of your program's source code. This process is called conditional compilation.

The conditional preprocessor construct is much like the if selection structure. Consider the following preprocessor code:

```
#ifndef NULL
#define NULL 0
#endif
```

You can compile a program for debugging purpose and can debugging turn on or off using a single macro as follows:

```
#ifdef DEBUG
    cerr <<"Variable x = " << x << endl;
#endif
```

causes the **cerr** statement to be compiled in the program if the symbolic constant DEBUG has been defined before directive `#ifdef DEBUG`. You can use `#if 0` statment to comment out a portion of the program as follows:

```
#if 0
    code prevented from compiling
#endif
```

Let us try the following example:

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

#define MIN(a,b) (((a)<(b)) ? a : b)

int main () {
    int i, j;
    i = 100;
    j = 30;

    #ifdef DEBUG
        cerr <<"Trace: Inside main function" << endl;
    #endif

    #if 0
        /* This is commented part */
        cout << MKSTR(HELLO C++) << endl;
    #endif

    cout <<"The minimum is " << MIN(i, j) << endl;
```



```

#ifdef DEBUG
    cerr << "Trace: Coming out of main function" << endl;
#endif
    return 0;
}

```

If we compile and run above code, this would produce the following result:

Trace: Inside main function

The minimum is 30

Trace: Coming out of main function

## **The # and ## Operators**

The # and ## preprocessor operators are available in C++ and ANSI/ISO C. The # operator causes a replacement-text token to be converted to a string surrounded by quotes.

Consider the following macro definition:

```

#include <iostream>
using namespace std;

#define MKSTR( x ) #x

int main () {
    cout << MKSTR(HELLO C++) << endl;

    return 0;
}

```

If we compile and run above code, this would produce the following result:

HELLO C++

Let us see how it worked. It is simple to understand that the C++ preprocessor turns the line:

```
cout << MKSTR(HELLO C++) << endl;
```

into the following line:

```
cout << "HELLO C++" << endl;
```

The ## operator is used to concatenate two tokens. Here is an example:

```
#define CONCAT( x, y ) x ## y
```

When CONCAT appears in the program, its arguments are concatenated and used to replace the macro. For example, CONCAT(HELLO, C++) is replaced by "HELLO C++" in the program as follows.

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

#define concat(a, b) a ## b

int main() {
    int xy = 100;

    cout << concat(x, y);
    return 0;
}
```

If we compile and run above code, this would produce the following result:

```
100
```

Let us see how it worked. It is simple to understand that the C++ preprocessor transforms:

```
cout << concat(x, y);
```

into the following line:

```
cout << xy;
```

## Predefined C++ Macros

C++ provides a number of predefined macros mentioned below:

Macro	Description
<code>__LINE__</code>	This contain the current line number of the program when it is being compiled.
<code>__FILE__</code>	This contain the current file name of the program when it is being compiled.
<code>__DATE__</code>	This contains a string of the form month/day/year that is the date of the translation of the source file into object code.
<code>__TIME__</code>	This contains a string of the form hour:minute:second that is the time at which the program was compiled.

Let us see an example for all the above macros:

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

int main () {
    cout << "Value of __LINE__ : " << __LINE__ << endl;
    cout << "Value of __FILE__ : " << __FILE__ << endl;
    cout << "Value of __DATE__ : " << __DATE__ << endl;
    cout << "Value of __TIME__ : " << __TIME__ << endl;

    return 0;
}
```

If we compile and run above code, this would produce the following result:

Value of \_\_LINE\_\_ : 5

Value of \_\_FILE\_\_ : main.cpp

Value of \_\_DATE\_\_ : Oct 21 2016

Value of \_\_TIME\_\_ : 01:01:48

*// function macro*

*#include <iostream>*

*using namespace std;*

*#define getmax(a,b) ((a)>(b)?(a):(b))*

*int main()*

*{*

*int x=5, y;*

*y= getmax(x,2);*

*cout << y << endl;*

*cout << getmax(7,x) << endl;*

*return 0;*

*}*