



Computer Training Institute

An ISO 9001:2008 Certified Company

C++ Programming

First published on 3 July 2012
This is the 7th Revised edition

Updated on:
03 August 2015

DISCLAIMER

- The data in the tutorials is supposed to be one for reference.
- We have made sure that maximum errors have been rectified. In spite of that, we (ECTI and the authors) take no responsibility in any errors in the data.
- The programs given in the tutorials have been prepared on, and for the IDE Microsoft Visual Studio 2013.
- To use the programs on any other IDE, some changes might be required.
- The student must take note of that.

Procedure-Oriented Programming

- The procedure oriented language deals with a sequence of processes to be done such as reading, calculating and printing.
- The **program is divided into functions** to achieve these tasks.
- In procedure oriented languages the main attention is given to how to achieve a particular task and very **less attention is given to the data** which is used by the functions.
- If the data is required to be accessed by many functions then it is declared as **global**. Global declarations are more vulnerable as data can be changed by all the functions.

Problems in the 'C' Language

- The data elements while programming in C are not considered.
- Problem of analysing the UNIX Kernel with respect to Distributed Systems.
- Lack of security for Networks.
- Data Categorisation not present.

Object Oriented Programming

- More emphasis is given on data rather than procedure.
- Programs are divided into objects.
- Classes are designed such that they characterize the objects.
- The data and the functions that can operate on the data are tied together.
- Data is hidden and cannot be accessed by external functions.
- New data and functions can be added whenever required.
- Follows bottom-up Design approach.

Introduction to C++

- C++ language was developed by Bjarne Stroustrup at AT & T Bells Laboratories in early 1980's.
- He thought of a language which can have object oriented features as well as it can retain the simplicity of C language.
- Initially the language was names as 'C with classes'. However, later in 1983 the name was changed to C++.
- C++ almost supports all the C functionalities with some new functionalities like classes, inheritance, function overloading, operator overloading. These features of C++ enable creating abstract data types, inherit properties from existing data types etc.

First C++ Program

```
#include<iostream.h>
#include<conio.h>
void main()
{
    cout << "Hello World!!!";
    getch();
}
```

Output:

Hello World!!!

- **#include<iostream.h>** instruction causes to add `iostream.h` file to program which contains the declarations of the identifier **cout**.
- **cout (<<)** is called as a Output Operator which prints string to the console.

Note: The header files should now be added by the student. Henceforth, no headers are mentioned in the sample programs.

Class

```
class class_name
```

```
{
```

```
    private:
```

```
        variables;
```

```
        functions;
```

```
    public:
```

```
        variables;
```

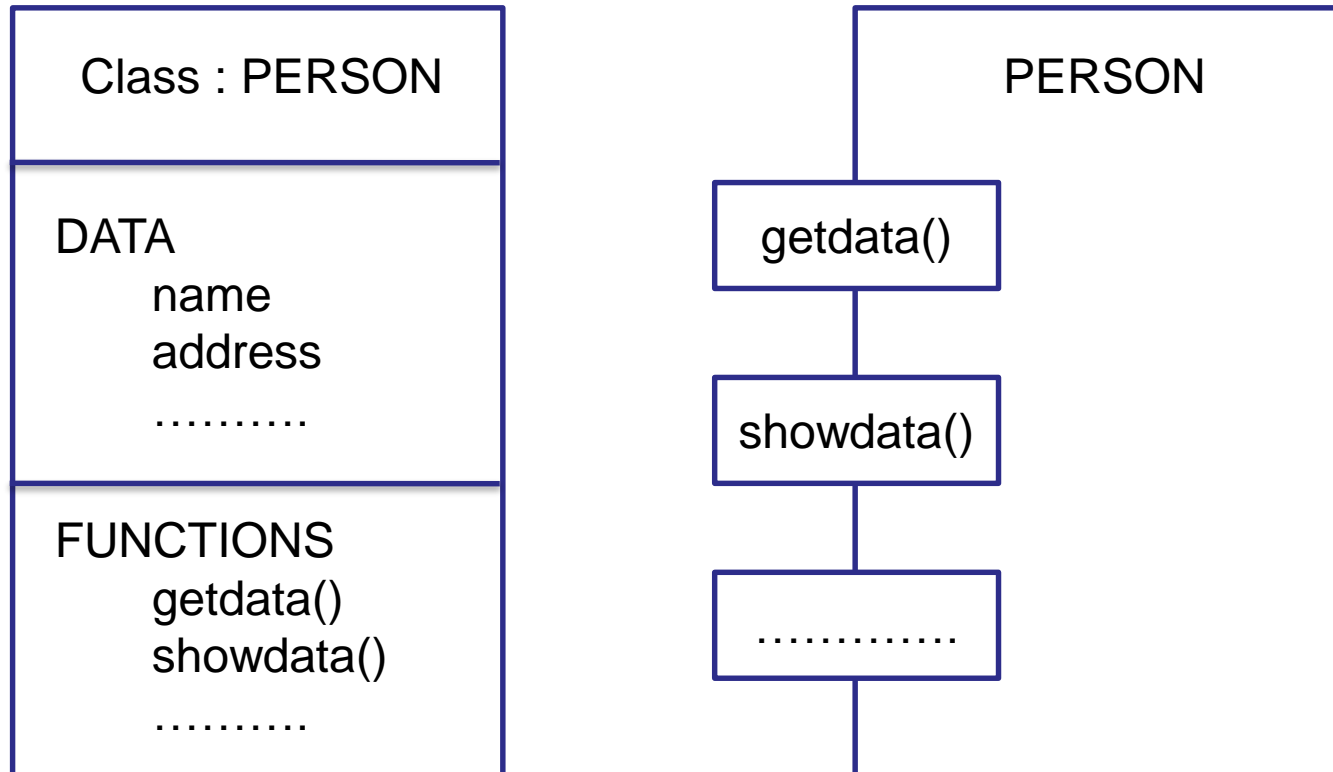
```
        functions;
```

```
};
```

- The body of a class contains **variables and functions**.

- A class defines the structure and behavior (data and code) that will be shared by a set of objects.
- The purpose of class is to encapsulate complexity, there are mechanisms for hiding the complexity of the implementation inside the class.
- The words **private** and **public** are called as **access specifiers**. By default the access specifier is private in C++.

Class Representation



Objects

```
class demo
{
    int no1, no2, add;
public:
    void getdata()
    {
        cout << "Enter 2 nos: " ;
        cin >> no1 >> no2;
    }
    void sum()
    {
        add = no1 + no2;
    }
    void showdata()
    {
        cout << "Sum is: " << add;
    }
};
```

```
void main()
{
    demo obj;
    obj.getdata();
    obj.sum();
    obj.showdata();
    getch();
}
```

- Each object of a given class contains the structure and behavior defined by the class.
- Thus **class is a logical representation**, where as **object has physical representation**.
- The memory is always assigned to object and not to a class.

Member Functions

Defining Function inside Class

```
class demo
{
    int no1, no2, add;
public:
    //inline function
    void getdata() //inside
    class function
    {
        cout << "Enter 2 nos: "
        ;
        cin >> no1 >> no2;
    }
    void showdata()
    {
        cout << "Sum is: " <<
        add;
    }
};
```

Defining Function outside Class

```
class demo
{
    int no1, no2, add;
public:
    void getdata(); //prototype
    declaration
    void showdata()
    {
        cout << "Sum is: " <<
        add;
    }
};
void demo :: getdata()
{
    cout << "Enter 2 nos: " ;
    cin >> no1 >> no2;
}
```

Defining Functions Outside Class

- When you are defining a function outside a class you have to incorporate **membership 'identity label'** in the header. The 'identity label' tells the compiler which class the function belongs to.

```
return_type class_name :: function_name (arguments)
{
    .....;
    .....;
}
```

- Due to **:: (scope resolution operator)** the compiler understands the function **function_name** is restricted to **class_name**.
- Due to membership label many classes can use same function name.
- Member functions can access the private data of the class.
- Member function can call another member function without using dot operator.

Inline Functions

- When we call a function it takes extra time to execute.
- To eliminate the cost of calls to small functions **inline functions** are used.
- **An inline function is a function which is expanded in line when it is invoked.**
- The compiler replaces the function call with the corresponding function code.

```
inline function_header
{
    .....;
    .....;
}
```

- The faster execution of inline function diminishes when the function grows in size.
- Inline functions should be between 1 to 2 lines.

Passing Object as a Parameter to Function

- You can pass object as a function argument.
- If you pass entire object to function then it is called as **pass-by-value** and if only you pass addresses of object then it is called as **pass-by-reference**.
- **obj1.calsum(obj2)**, in this function call obj2 is passed by value and hence the function prototype will be as follows – **void calsum (class_name o)**. This means if we change variable values of **o** they will not affect values in obj2.
- **obj1.calsum(obj2)**, in this function obj2 is passed by reference and hence the function prototype will be as follows – **void calsum (class_name &o)**. This means if we change variable values of **o** it will affect values in obj2.
- **Object to Pointers-** **obj1.calsum(obj2)** will be passed to function as follows – **void calsum(class_name *o)**. This means **o** will point to the memory of obj2.

Returning Objects

```
class maths
{
    int no1, sum;
public:
    void getdata(int);
    void showdata();
    maths calsum(maths);
};
void maths :: getdata(int x)
{
    no1 = x;
}
void maths :: showdata()
{
    cout << "The sum is:" <<
    sum;
}
```

```
maths maths :: calsum(maths o)
{
    maths temp;
    temp.sum = no1 + o.no1;
    return temp;
}
void main()
{
    maths obj, obj1, obj2;
    obj.getdata(10);
    obj1.getdata(20);
    obj2 = obj1.calsum(obj);
    obj2.showdata();
    getch();
}
```

- Here **obj1** is called as **invoking object** and the default memory will be of obj1.

Array of Objects

```
class student
{
    int roll_no;
    char name[50], address[50];
public:
    void getdata();
    void showdata();
};

void student :: getdata()
{
    cout << "Enter roll no.: ";
    cin >> roll_no;
    cout << "Enter name: ";
    gets(name);
    cout << "Enter address: ";
    gets(address);
}
```

```
void student :: showdata()
{
    cout << "Roll no.: " << roll_no;
    cout << "Name: " << name;
    cout << "Address: " << address;
}

void main()
{
    student s[3]; //array of objects
    cout << "Enter data of 3
students: ";
    for(int i=0; i<3; i++)
    {
        s[i].getdata();
    }
    cout << "The information is: ";
    for(int i=0; i<3; i++)
    {
        s[i].showdata();
    }
    getch();
}
```


Array of Objects to Functions

```

class student
{
    int roll_no;
    char name[50], address[50];
public:
    void getdata();
    void showdata(student [ ],
        int);
};

void student :: getdata()
{
    cout << "Enter roll no. : ";
    cin >> roll_no;
    cout << "Enter name: ";
    gets(name);
    cout << "Enter address: ";
    gets(address);
}

void student :: showdata(student s[ ],int n)
{
    for(int i=0;i<n;i++)
    {
        cout << "Roll no.: " << s[i].roll_no;
        cout << "Name: " << s[i].name;
        cout << "Address: " << s[i].address;
    }
}

void main()
{
    student s[3], s1; //array of objects
    cout << "Enter data of 3 students: ";
    for(int i=0; i<3; i++)
    {
        s[i].getdata();
    }
    cout << "The information is: ";
    s1.showdata(s,3);
    getch();
}

```

Array of Objects to Pointers

```

class student
{
    int roll_no;
    char name[50], address[50];
public:
    void getdata();
    void showdata(student *,
    int);
};
void student :: getdata()
{
    cout << "Enter roll no.: ";
    cin >> roll_no;
    cout << "Enter name: ";
    gets(name);
    cout << "Enter address: ";
    gets(address);
}

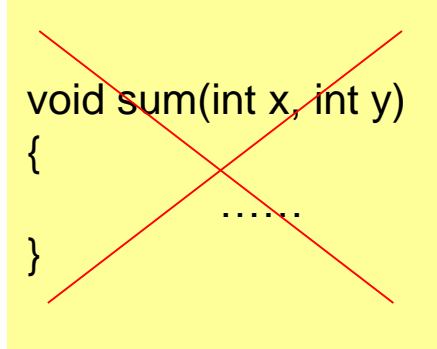
void student :: showdata(student *s,
    int n)
{
    for(int i=0;i<n;i++,s++)
    {
        cout<<"Roll no.: "<<s->roll_no;
        cout<<"Name: "<< s->name;
        cout<<"Address: "<< s->address;
    }
}
void main()
{
    student s[3], s1; //array of objects
    cout<<"Enter data of 3 students: ";
    for(int i=0; i<3; i++)
    {
        s[i].getdata();
    }
    cout << "The information is: ";
    s1.showdata(s,3);
    getch();
}

```

Function Overloading

- You can define **more than two functions having same name** as long as their parameters are different.
- Function overloading is one of the ways that C++ supports **polymorphism**.
- The **type and/or number of arguments** determine which overloaded version is to be called.
- C++ does not allow overloaded functions with same parameters but different return types.

```
class maths
{
    void sum()
    {
        .....
    }
    int sum(int x, int y)
    {
        .....
    }
    void sum(int x)
    {
        .....
    }
    void sum(int x, float y)
    {
        .....
    }
};
```



```
void sum(int x, int y)
{
    .....
}
```

Constructors

- In C++ you can initialize the class variables at the time of creation of objects using constructors.
- **Constructor has same name as the class** in which it resides and syntactically similar to a method.
- **Constructors do not have return type nor they are void.**
- The constructor is automatically called immediately after the object is created.

```
class demo
{
    int a, b;
public:
    demo()
    {
        a = 10; b = 20;
    }
    void showdata()
    {
        cout << "A=" << a << "B=" << b;
    }
};

void main()
{
    demo obj1;
    obj1.showdata();
    getch();
}
```

Constructor will get automatically called when the object obj1 is created

Parameterized Constructors

- In the example seen before variables of all the objects created will get initialized to 10 and 20. If we want to initialize them to different values you have to use parameterized constructors.
- The constructors with arguments is called as parameterized constructors.

```
class demo
{
    int a, b;
public:
    demo(int x, int y);
    .....;
    .....;
};

demo :: demo (int x, int y)
{
    a = x; b = y;
}
```

- You can call the parameterized constructor implicitly or explicitly
- **Implicit call**: demo obj(10, 20);
- **Explicit call**: demo obj = demo(10,20);

Constructor Overloading

```
class maths
{
    int x, y;
public:
    maths()
    { x = 0; y = 0}
    maths(int a, int b)
    { x = a; y = b;}
    maths(maths &o)
    { x = o.x, y = o.y}
};
```

- In the above class we have overloaded three constructors.
- **maths obj** declaration will invoke first constructor where x and y will be initialized to 0.
- **maths obj1(100, 200)** will invoke second constructor where x and y will be initialized to 100 & 200 respectively.

- **maths obj2(obj1)** will invoke third constructor and will copy the values of obj1 to obj2. Hence it is called as **copy constructor**.
- Some other examples of copy constructor –
- **maths obj2 = obj1** will create a new object obj2 and same time will initialize the values to that of obj1.
- **obj2 = obj1** will not invoke copy constructor but due to **= operator overloaded** it will initialize the values of obj2 to obj1 member by member.
- The parameter to a constructor can be any type except the class it belongs to.
- Hence **maths(maths o)** will not work but **maths(maths &o)** will work because we have passed the reference of it's own class.

Destructor

- Destructor is used to destroy the objects created by constructors.
- Same as constructor destructor name is also same as that of class name preceded by tilde (~) character.
- Destructor does not take any argument nor returns any value.
- Destructor is automatically invoked by the compiler on exit of a program or a block of code in which the objects are declared.
- Destructor releases the memory for further use.

e.g.

```
~student () {  
    cout<<"\nIn Destructor";  
}
```

Destructor for class student.

this Pointer

- This pointer is used to represent an object that invokes a member function.
- This pointer points to the object on which a function is called e.g. **obj.sum()** will set **this pointer to the address of object obj.**

Program with ambiguity issue

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

class maths
{
    int a, b;
    maths(int a, int b)
    {
        a=a; b=b; //both a & b
                  are locals
    }
    void display()
    {
        cout << a << ", " <<
        b;
    }
};
```

```
void main()
{
    maths obj1(10, 20);
    clrscr();
    obj1.display();
    getch();
}
```

- Here we are expecting the output would be **10, 20**. But the output would be some garbage values. Because a and b are the local variables. Hence the statement a = a and b = b assigns the values of local variables a and b to itself.

Use of this keyword

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

class maths
{
    int a, b;
    maths(int a, int b)
    {
        this->a = a;
        this->b = b;
    }
    void display()
    {
        cout << a << ", "
        << b;
    }
};
```

```
int main()
{
    maths obj1(10, 20);
    clrscr();
    obj1.display();
    getch();
    return 0;
}
```

Output :-

10, 20

new Keyword (Dynamic Memory Allocation)

- Allocating memory to the objects at the time of their construction is called as Dynamic Memory Allocation. The memory is allocated with the help of **new keyword**.
- Dynamic memory allocation enables programmer to allocate right amount of memory when the objects are not of the same size.

Use of new keyword

```
#include<iostream.h>
#include<string.h>
class String
{
    char *name;
    int length;
public:
    String()
    {
        length = 0;
        name = new char [length
+ 1];
    }
    String(char *s)
    {
        length = strlen(s);
        name = new char [length
+ 1];
        strcpy(name, s);
    }
}
```

```
void display()
{
    cout << name << "\n";
}
void join (String &, String
&);
};

void String :: join (String &a,
String &b)
{
    length = strlen(a.name) +
strlen(b.name);
delete name;
    name = new char [length + 1];
    strcpy(name, a.name);
    strcat(name, b.name);
}
```

continued on next slide...

Use of new keyword

```
void main()
{
    char *first = "Parag ";
    String name1(first),
    name2("Sarang "),
    name3("Pooja "), s1, s2;
    s1.join(name1, name2);
    s2.join(s1, name3);
    name1.display();
    name2.display();
    name3.display();
    s1.display();
    s2.display();
    getch();
}
```

The output of the program is –

Parag

Sarang

Pooja

ParagSarang

ParagSarangPooja

Static Data Members

- We can declare any data member as **Static**.
- Static data member is **initialized to zero** when the first object of the class is created. **No other initialization is permitted.**
- When we define a data member as Static **only one copy of that member is created** for all the objects.
- It is visible only within the class, but its **lifetime is for the entire program.**

```
class stat
{
    int no,
    static int scount; //Static
                        member
                        Variable

public:
    void getdata(int x)
    {
        no= x;
        scount++;
    }
    void getcount()
    {
        cout<<"Count: " <<scount;
    }
};
```

Static Data Members

```
int stat :: scount;
void main()
{
    stat a, b, c;
    a.getcount();
    b.getcount();
    c.getcount();
    a.getdata(100);
    cout << "After
initialization: " << "\n";
    a.getcount();
    cout << "After
initialization : " << "\n";
    b.getdata(200);
    b.getcount();
    c.getdata(300);
    cout << "After
initialization : " << "\n";
    c.getcount();
    getch();
}
```

The output of the program is –

Count: 0

Count: 0

Count: 0

After initialization:

Count: 1

After initialization:

Count: 2

After initialization:

Count: 3

Static Member Functions

- We can declare static member functions as we define a data member as static.
- A static function can access other static member functions or static data members.
- Static functions are called using the class name and cannot be called using the objects of the class.

```
class stat1
{
    int no;
    static int scount;
public:
    void setno()
    {
        no = ++scount;
    }
    void showno()
    {
        cout << "\nObject
Number:" << no;
    }
}
```

continued on next slide....


```
static void showcount()
{
    cout<<"\nCount: " <<
scout;
}
}; //end of class
int stat1 :: scout;
int main()
{
    stat1 s1, s2;
    s1.setno();
    s2.setno();
    stat :: showcount();
//calling static function
    stat1 s3;
```

```
s3.setno();
stat :: showcount();
s1.showno();
s2.showno();
s3.showno();
getch();
} // end of main
```

Output:

Count: 2

Count: 3

Object Number: 1

Object Number: 2

Object Number: 3

Friend Functions

- The private members of a class cannot be accessed from outside the class.
- There might be situation in which the two classes need to share a particular function, we need to make that function as a friend function of both the class.
- So using friend function we can access private data of both the classes.

```
class ABC
{
    .....
    public:
    .....
    friend void name(ABC);
};
```

- You cannot call a friend function using an object of a class, it needs to be called as normal function.
- You need to pass object as an argument.

```
class calMath
{
    int a, b;
public:
    void setdata(int x, int y)
    {
        a = x;
        b = y;
    }
    friend float mean(calMath );
};

float mean (calMath m)
{
    float x = (m.a + m.b) / 2.0;
    return x;
}

int main()
{
    calMath mobj;
    mobj.setdata(22,19);
    cout<<"\nMean = "<<mean(mobj);
    getch();
}
```

Output:
Mean = 20.5

Friend Function to Two Classes

```

class PQR; //forward declaration
class LMN
{
    int a;
public:
    void setdata(int x)
    {
        a = x;
    }
    friend void max(LMN, PQR);
};
class PQR
{
    int b;
public:
    void setvalue(int y)
    {
        b = y;
    }
    friend void max(LMN, PQR);
};

```

```

void max(LMN l, PQR p)
{
    if(l.a > p.b)
        cout<<"Max No. = "<<l.a;
    else
        cout<<"Max No. = "<<p.b;
}
void main()
{
    LMN lmn;
    lmn.setdata(100);
    PQR pqr;
    pqr.setvalue(200);
    max(lmn, pqr);
    getch();
}

```

Output:
Max No. = 200

Operator Overloading

- In C++ you can give **special meaning to the operators**. This process is called as operator overloading.
- In C++ all the operators except the following operators can be overloaded –
 - Class member operators (., .*)
 - Scope resolution Operator (::)
 - Size operator (sizeof)
 - Conditional operators (?:)

```
return_type class_name ::  
operator (op-argslist)  
{  
    .....;  
    .....;  
}
```

- Operator functions must be either member functions (non-static) or friend functions.
- Friend function will only have one argument for unary operators and two for binary operators.
- Member function has no arguments for unary operators and only one for binary operators.

Overloading Unary Operators

```
class opunary
{
    int no;
public:
    void getdata(int );
    void showdata();
    void operator -();
};

void opunary :: getdata (int a)
{no = a; }

void opunary :: showdata()
{cout << no; }

void opunary :: operator - ()
{no = -no; }
```

```
void main()
{
    opunary obj;
    obj.getdata(10);
    cout << "Value is: ";
    obj.showdata();
    -obj;
    cout << "Value is: ";
    obj.showdata();
    getch();
}
```

Output :-

Value is: 10

Value is: -10

Overloading Binary Operators

```
class complex
{
    float real, img;
public:
    complex() { }
    complex(float r, float i)
    {
        real = r; img = i;
    }
    complex operator +(complex);
    void display();
};

complex complex :: operator
+(complex c)
{
    complex temp;
    temp.real = real + c.real;
    temp.img = img + c.img;
    return temp;
}
```

```
void complex :: display()
{
    cout<<real<<"+"<<img<<"i";
}

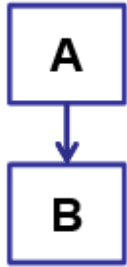
void main()
{
    complex c1, c2, c3;
    c1 = complex(3.2, 1.8);
    c2 = complex(2.3, 5.3);
    c3 = c1 + c2;
    cout << "C1 = ";
    c1.display();
    cout << "C2 = ";
    c2.display();
    cout << "C3 = ";
    c3.display(); getch();
}
```

Inheritance

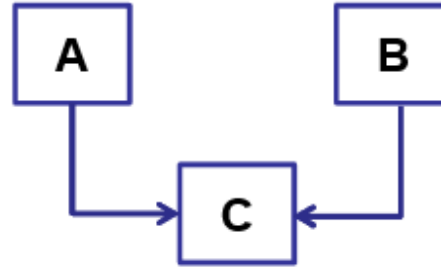
- Inheritance is a feature of OOP which enables user for reusability of the code.
- You can create new classes which can use properties of existing classes using inheritance and we can add new features in derived class.
- In inheritance the old class is called as base class and the new class is called as derived class.
- The derived class inherits some or all features from base class.
- One class can be derived from many base classes or many classes can be derived from one base class.

```
class derived : visibility_mode  
base {  
.....;  
.....;  
}
```

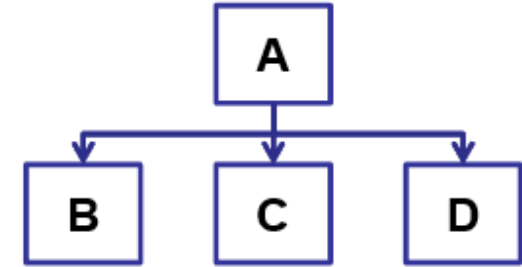

Types of Inheritance



Single Inheritance



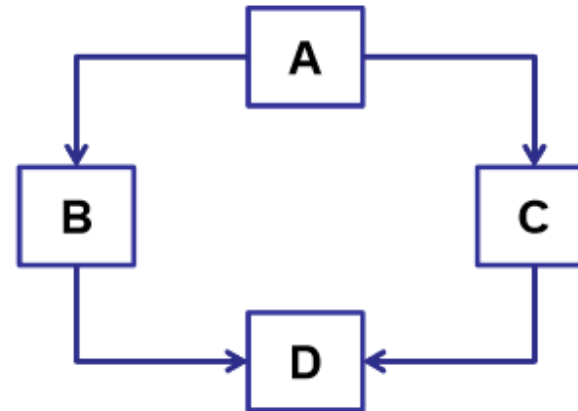
Multiple Inheritance



Hierarchical Inheritance



Multilevel Inheritance



Hybrid Inheritance

Deriving a Class from Base Class

Default Private Derivation:

```
class derived : base
{
    members of derived class;
}
```

Private Derivation:

```
class derived : private base
{
    members of derived class;
}
```

Public Derivation:

```
class derived : public base
{
    members of derived class;
}
```

- The : indicates derivation of derived from base class. Private or Public indicate the visibility of the features of base class into derived class.
- If **privately derived**, the public members of base class become private members of derived class and if **publically derived** then public members of base class become public members of derived class. **In private derivation we cannot access public member functions of base class from derived class object** but **in public derivation we can access public member functions of base class from derived class object.**

Protected Access Specifier

- If we require the private data to be visible in the derived class then C++ provides the protected visibility modifier.
- Protected member inherited in public mode becomes protected in derived class, hence it is accessible to the member function of derived class and ready for further inheritance.
- Protected member inherited in private mode becomes private in derived class, hence it is accessible to the member functions of derived class but not available for further inheritance.

```
class ABC
{
    private: //optional
        .....; //visible to member
        .....; //functions within the class
    protected:
        .....; //visible to member
functions
        .....; //of its own and derived
class
    public:
        .....; //visible all the functions
        .....; //in the program
};
```

Function Overriding

- When we define a function with same name and arguments in both base class and derived class, then the function is called as overridden function and this process is called as function overriding.
- Function overriding enables programmer to change the behavior of any base class functionality in the derived class.
- When you call the overridden function by child object the child version is called and when you call the overridden using base object the base version is called.

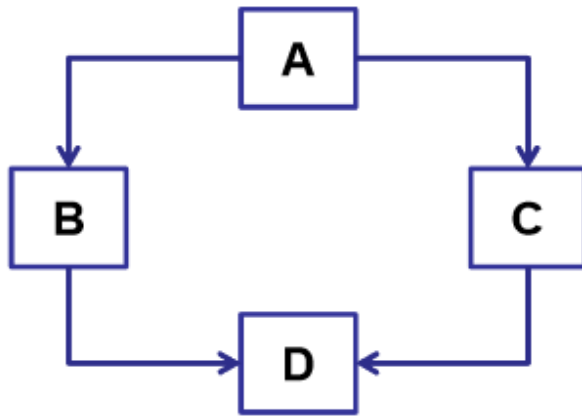
```
class A
{
    .....;
    public:
        void getdata(int);
};
Class B : public A
{
    .....;
    public:
// overridden function
        void getdata(int);
        void showdata();
};
```

Ambiguity Resolution in Inheritance

```
class ABC
{
public:
    void display(void)
    {
        cout<<"I am in Class
ABC";
    }
};
class XYZ
{
public:
    void display(void)
    {
        cout<<"I am in Class
XYZ";
    }
};
```

```
class LMN : public ABC, public XYZ
{
public:
    void show(void)
    {
        cout<<"I am in class LMN";
    }
};
int main()
{
    LMN o;
    o.display(); //will give ambiguity error
    o.ABC::display(); //display of ABC
called
    o.XYZ::display(); //display of XYZ
called
    p.show();
    getch();
}
```

Virtual Base Class



- In above inheritance, B and C classes inherit features of A and then as D is derived from B and C, there is an **ambiguity error** because the **compiler does not understand how to pass features of Class A to Class D either using B class or C Class.**

- So to remove the ambiguity you need to make the class B and class C as virtual base classes.

Class B : public **virtual** A

Class C : **virtual** public A

- When the class is made as virtual base class, the compiler takes **necessary care to pass only one copy of the members in inherited class** regardless of many inheritance paths exist between the virtual base class and derived class.

Virtual Base Class - Example

```
class student
{
protected:
    int roll_number;
public:
    void get_number(int a)
    {
        roll_number = a;
    }
    void put_number(void)
    {
        cout <<"Roll Number:
"<<roll_number << "\n";
    }
};
```

```
class test : public virtual student
{
protected:
    float part1, part2;
public:
    void get_marks(float x, float y)
    {
        part1 = x;
        part2 = y;
    }
    void put_marks(void)
    {
        cout<<"Markts Obtained:
"<<"\n"<<"Part1 = "<<part1<<
"\n"<<"Part2 = "<<part2<< "\n";
    }
};
```

continued on next slide....

```
class sports : virtual public
student
{
protected:
    float score;
public:
    void get_score(float s)
    { score = s; }
    void put_score(void)
    {
        cout << "Sports Marks = "
<< score << "\n";
    }
};
class result : public test,
public sports
{
    float total;
public:
    void display(void);
};
```

```
void result :: display(void)
{
    total = part1+part2+score;
    put_number();
    put_marks();
    put_score();
    cout<<"Total Score: "<<total
<<"\n";
}
void main()
{
    result student1;
    student1.get_number(10);
    student1.get_marks(75.25,82.90);
    student1.get_score(92.80);
    student1.display();
    getch();
}
```


Behaviour of Constructors in Derived Class

- When we create the object of derived class the default constructor of base class gets called and then default constructor of derived class gets called.
- When you create object of derived class with parameters, still default constructor of base class gets called and parameterized constructor of child class gets called.
- If base class contains constructor more than one argument, then it is compulsory for the derived class to have a constructor and pass the arguments to base class constructor.

Constructors in Single Inheritance (Example)

```
class base
{
    int a;
public:
    base()
    {
        cout << "I am into base's
default constructor";
    }
    base(int x)
    {
        cout << "I am into base's
parameterized constructor";
        a = x;
    }
    void show_a()
    {
        cout << "\nThe value of a
is: " << a;
    }
};
```

```
class derived : public base
{
    int b;
public:
    derived()
    {
        cout << "\nI am in child's
default constructor";
    }
    derived(int y, int z):base(y)
    {
        cout << "\nI am in child's
parameterized constructor";
        b = z;
    }
    void show_b()
    {
        cout << "\nThe value of b
is: " << b;
    }
};
```

continued on next slide....

```
void main()
{
    derived d;
    derived d1(10,20);
    d.show_b();
    d.show_a();
    getch();
}
```

Output:

I am into base's default constructor
I am into child's default constructor
I am into base's parameterized
constructor
I am into child's parameterized
constructor
The value of b is: 20
The value of a is: 10

Constructors in Multilevel Inheritance (Example)

```
class A
{
    int a;
public:
    A()
    {
        cout << "Into Class A's
default constructor";
    }
    A(int x)
    {
        a = x;
        cout << "\nInto Class A's
parameterized constructor";
        cout << "\nA = " << a;
    }
};
```

```
class B:public A
{
    int b;
public:
    B()
    {
        cout << "\nInto Class B's
default constructor";
    }
    B(int l, int m):A(l)
    {
        b = m;
        cout << "\nIn Class B's
parameterized constructor";
        cout << "\nB = " << b;
    }
};
```

continued on next slide....

```
class C:public B
{
    int c;
public:
    C()
    {
        cout << "\nInto Class C's
default constructor";
    }
    C(int p,int q, int r):B(p,q)
    {
        c = r;
        cout << "\nInto Class C's
parameterized constructor";
        cout << "\nC = " << c;
    }
};
```

```
void main()
{
    C c;
    C c1(10,20,30);
    getch();
}
```

Output:

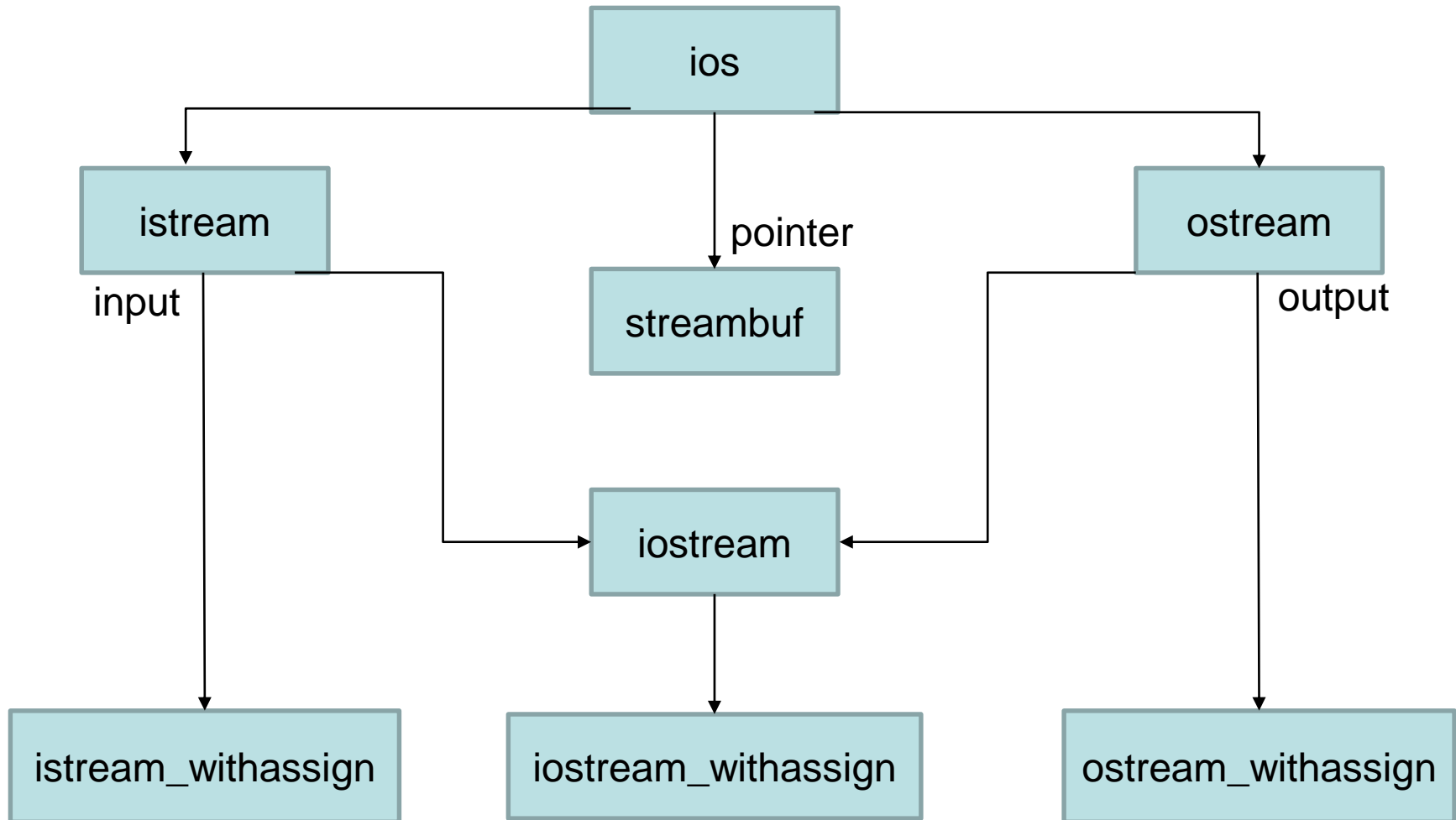
```
Into Class A's default constructor
Into Class B's default constructor
Into Class C's default constructor
Into Class A's parameterized constructor
A = 10
Into Class B's parameterized constructor
B = 20
Into Class C's parameterized constructor
C = 30
```

Abstract Class

- Abstract class is designed to act as base class for other classes.
 - It is a design concept in a program development and provides a base upon which other classes may be built.
 - You cannot create object of an Abstract class.
 - Abstract class is used only to derive other child classes.
 - A class with a Pure Virtual Function is called as an Abstract Class
-

```
Class A
{
    int a,b;
public:
    //pure virtual func
    void add()=0;
    void show();
    void get();
}
```

Console I/O Operations



put() and get() Functions

- **get()** function is defined in the `istream` class in order to handle single character inputs.
- It can be implemented as:
 - `cin.get(ch);`
- **put()** function is defined in the class `ostream` in order to handle single character outputs.
- It can be implemented as:
 - `cout.put('x');`

The above method calls the `get()` function with `ch` as a parameter.

- `ch=cin.get();`

The above method calls the `get()` function without a parameter returning the value to the variable `ch`.

The above method calls the `put` function with a character value `'x'` as a parameter.

- `cout.put(ch);`

The above method calls the `put()` function with a variable `ch`.

get() and put() function

```
void main()
{
    char ch;
    cout<<"Enter character: ";
    cin.get(ch);
    cout<<"You entered: "<<ch;
    getch();
}
```

Output:

Enter character: h

You entered: h

```
void main()
{
    char ch;
    cout<<"Enter character: ";
    cin.get(ch);
    cout<<"You entered: ";
    cout.put(ch);
    getch();
}
```

Output:

Enter character: h

You entered: h

getline() and write() Functions

- getline() is a function defined to handle strings in the istream class.
- It takes two parameters:
 - A string variable that takes the string
 - Number of characters to be accepted as a string (line).
- write() is a similar function used to write strings onto files. It is defined in the ostream class.
- It takes two parameters:
 - A string that is to be displayed on the console
 - Number of characters to be written.

getline() and write() Functions

```
void main()
{
    char ch[20];
    cout<<"Enter a string: ";
    cin.getline(ch,19);
    cout<<"You entered: "<<<ch;
    getch();
}
```

Output:

Enter a string: Envision

You entered: Envision

```
void main()
{
    char ch[20];
    cout<<"Enter a string: ";
    cin.getline(ch,19);
    cout<<"You entered: ";
    cout.write(ch,19);
    getch();
}
```

Output:

Enter a string: Envision

You entered: Envision

Formatted Console I/O Operators

- Unformatted console operators are used in order to manipulate or interpret the input and output.
- On the other hand, formatted operators are used in order to give formatting to the output of your program.
- These operators are used just to give a formatting to the input / output. These cannot be used as tools to I/O.

ios Format Functions

| Function | Description |
|--------------------------|--|
| <code>width()</code> | To specify the required field size for displaying an output value |
| <code>precision()</code> | To specify the number of digits to be displayed after the decimal point of a float value |
| <code>fill()</code> | To specify a character that is used to fill the unused portion of a field |
| <code>setf()</code> | To specify format flags that can control the form of output display |
| <code>unsetf()</code> | To clear the flags specified |

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 defined within the class ostream, we call it as:

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

- e.g.:

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



```
cout.width(5);
cout << 543;
cout.width(5);
cout << 12;
```



Setting precision: precision()

- By default, the floating point values are printed with six digits after the decimal. However, we can specify the number of digits to be displayed after the decimal through `precision()`;

- e.g.:

```
cout.precision(3);  
cout << 3.14159;  
cout << 2.50062;
```

- Output:

3.142

2.5 (no trailing zeros)

Filling & Padding: fill()

- We have been printing values using larger spaces than required. These are generally blank spaces. With the help of fill(), we can fill the unused positions with the desired character.

- e.g.:

```
cout.fill('*');  
cout.width(5);  
cout << 23;
```

- Output:

| | | | | |
|---|---|---|---|---|
| * | * | * | 2 | 3 |
|---|---|---|---|---|

Formatting flags, bit-fields & setf()

- When we use `width()`, `precision()` and `fill()` functions, the data is put by default in right justification.
- We can manage this using this formatting.
- The `setf()` is a member function of the `ios` class and can set various flags in order to give proper formatting.

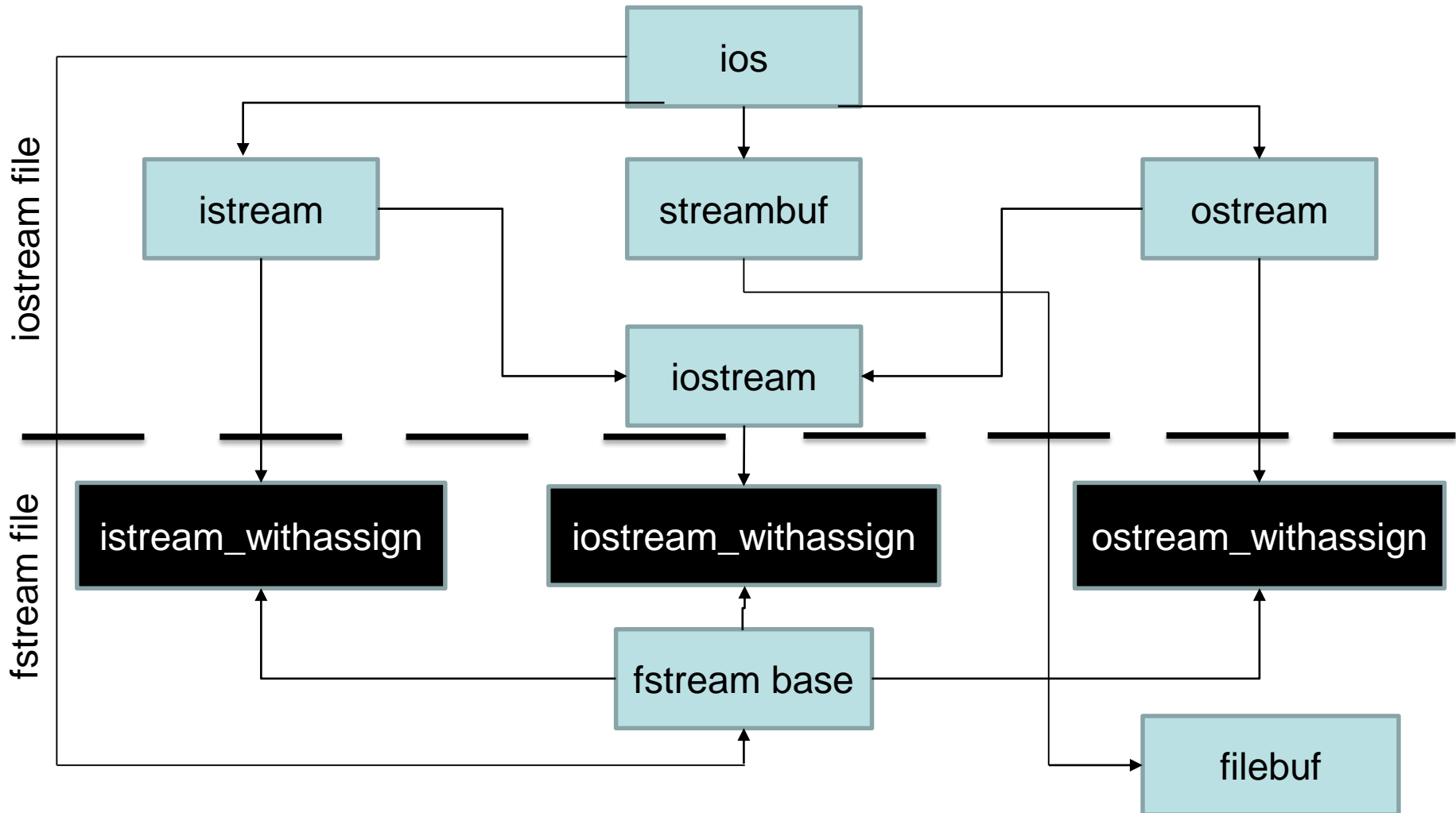
```
cout.setf(arg1, arg2);
```

- Here, `arg1` is one of the formatting flags defined in the class `ios`. It specifies the format action required for the output.
- Also present is, `arg2`. This specifies which group does the formatting flag belong to. It is called as the bit-field

...continued

| Format Required | Flag (arg1) | Bit-field (arg2) |
|--------------------------------------|-----------------|------------------|
| Left-justification | ios::left | ios::adjustfield |
| Right-justification | ios::right | ios::adjustfield |
| padding after sign or base indicator | ios::internal | ios::adjustfield |
| Scientific notation | ios::scientific | ios::floatfield |
| fixed point notation | ios::fixed | ios::floatfield |
| decimal base | ios::dec | ios::basefield |
| octal base | ios::oct | ios::basefield |
| hexadecimal base | ios::hex | ios::basefield |

Working on Files



Opening and Closing a File

- There are input and output stream classes for files similar to those for consoles.
- These are ifstream and ofstream respectively.
- To open a file in WRITE mode, we need to an object of the output stream.
- Thus,

```
ofstream outfile("results.txt");
```

is the desired statement.
- Similarly, to open a file in READ mode, the syntax is,

```
ifstream infile("results.txt");
```
- A file can also be opened through a function,

```
infile.open("results.txt");
```
- To close the file, we simply call the close function through the object:

```
infile.close();
```

Writing and Reading from a File

- Writing to a file is done through the overloaded operator (<<).

- e.g.:

```
ofstream outf;  
outf.open("results.txt");  
outf << "Hello World!";
```

- Similarly, reading is done through the overloaded operator (>>).

- e.g.:

```
ifstream inf;  
char name[30];  
inf.open("results.txt");  
inf >> name;
```

The getline() Function and EOF character

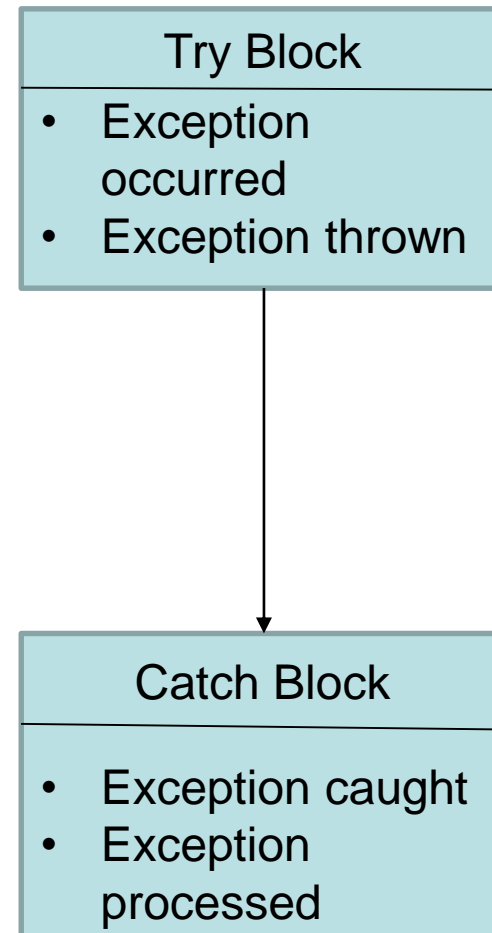
- File contents can also be read through a member function getline().
- It takes 2 parameters:
 - a string
 - number of characters
- When reading content from the file, we need to make sure that the End-of-File has not been encountered.
- To make sure we get this, a member function eof() is called to check if the end of file is encountered.
- The eof() returns a 0 if false.

```
ifstream inf;  
char name[30];  
inf.open("results.txt");  
inf.getline(name, 20);  
getch();
```

```
ifstream inf("results.txt");  
if(inf.eof()) { exit(0); }
```

Exception Handling

- The problems which are not syntax errors nor logical errors are called as Exceptions.
- When these are encountered, the program terminates unexpectedly.
- To avoid this, we use exception handling.



Exception Handling

- try block throwing an exception
- Invoking function throwing an exception
- Multiple catch statements
- catch-all
- Rethrowing an exception

Templates

- Templates is a feature in C++ which enables us to define generic classes and functions and thus provides support for generic programming.
- A template can be used to create a family of classes or functions.
- We can template for both:
 - Function Template
 - Class Template

Function Templates

```
template <class T>
void swap(T &x, T &y)
{
    T temp=x;
    x=y;
    y=temp;
}
```

- The definition above essentially declares a set of overloaded functions, one for each data type. We can invoke this swap function like an ordinary function

e.g.:

```
template <class T>
void swap(T &x, T &y)
{
    T temp=x;
    x=y;
    y=temp;
}

void func(int m, int n,
char ch, char pq)
{
    swap(m, n);
    swap(ch, pq);
}
```

Class Templates

```
template <class T>
class vector
{
    T vect;
    int size;
public:
    vector() { }
    vector(T a)
    {    vect=a;    }
};
```

- The class above is defined as a generic class and can take any data type specified.

- In order to define an object of the said class with the variable **'vect'** as an integer, we write,
`vector <int> v1;`
- Similarly, for a character, we say,
`vector <char> v2;`

END OF BASICS IN C++

- Thus, the basics of C++ programming ends here.
- We hope you are satisfied with the theory provided.
- You may get back to us for advance C++ concepts like Standard Template Libraries (STLs) and more.
- Feel free to share, distribute or use it in any form you wish to. **IT IS FOR YOU.** 😊

END OF BASICS IN C++

For advance C++ programming course or for any doubts in this tutorial, please contact us on any of the following details:

Call us on: 02065000419 / 02065111419

E-mail us at: prog@ecti.co.in

Web URL: www.ecti.co.in

For any suggestions of complaints regarding the theory or programs in the tutorial, write us at:

complaints@ecti.co.in