Object Oriented Programming

Ms. Sharmistha Roy,

Assistant Professor,

School of Computer Engineering,

KIIT University

Evolution Of Programming Paradigm

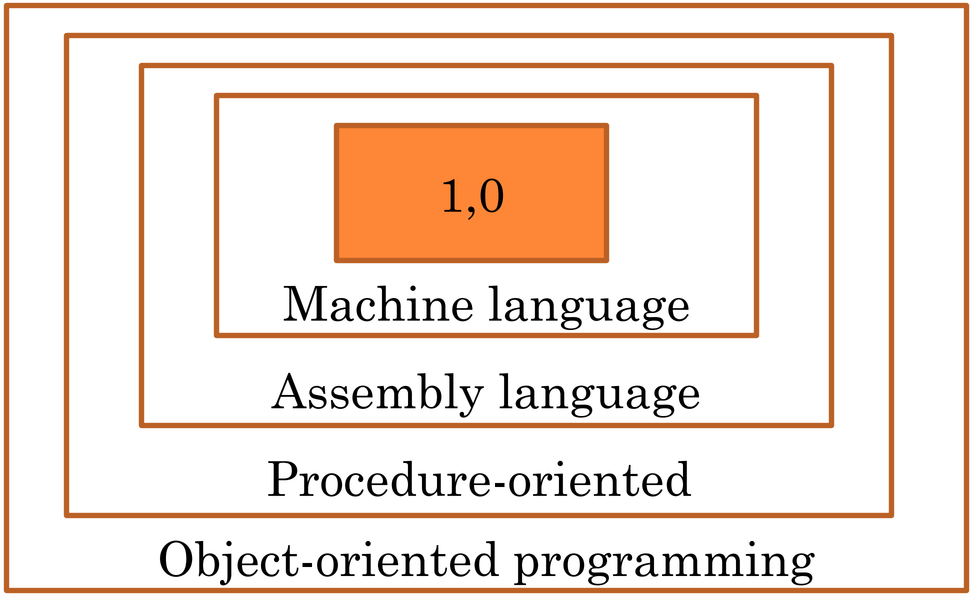
Software Crisis: development in software technology continue to be dynamic.

Issues need to be addressed to face this crisis:

- ✓ How to represent real-life entities of problems in system design?
- ✓ How to ensure reusability and extensibility of modules?
- ✓ How to develop modules that are tolerant to any changes in future?
- ✓ How to improve software productivity and decrease software cost?
- ✓ How to improve the quality of software?
- ✓ How to manage time schedules?

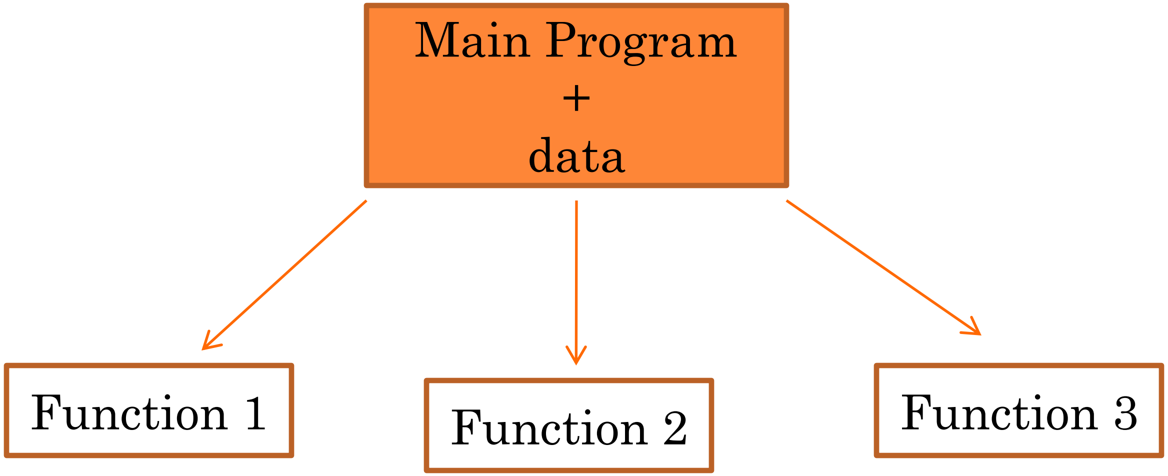
Evolution Of Programming Paradigm

- Software Evolution:
- Layers of Software evolution:



Structured procedural programming(SPP)

- Programming paradigm that to a large extent relies on the idea of dividing a program into functions and modules.
- Its basically consists of writing a list of instructions for the computer to follow, and organizing these instructions into groups known as functions.



- Examples: ‘C’, Pascal and Fortran

CHARACTERISTICS OF SPP..

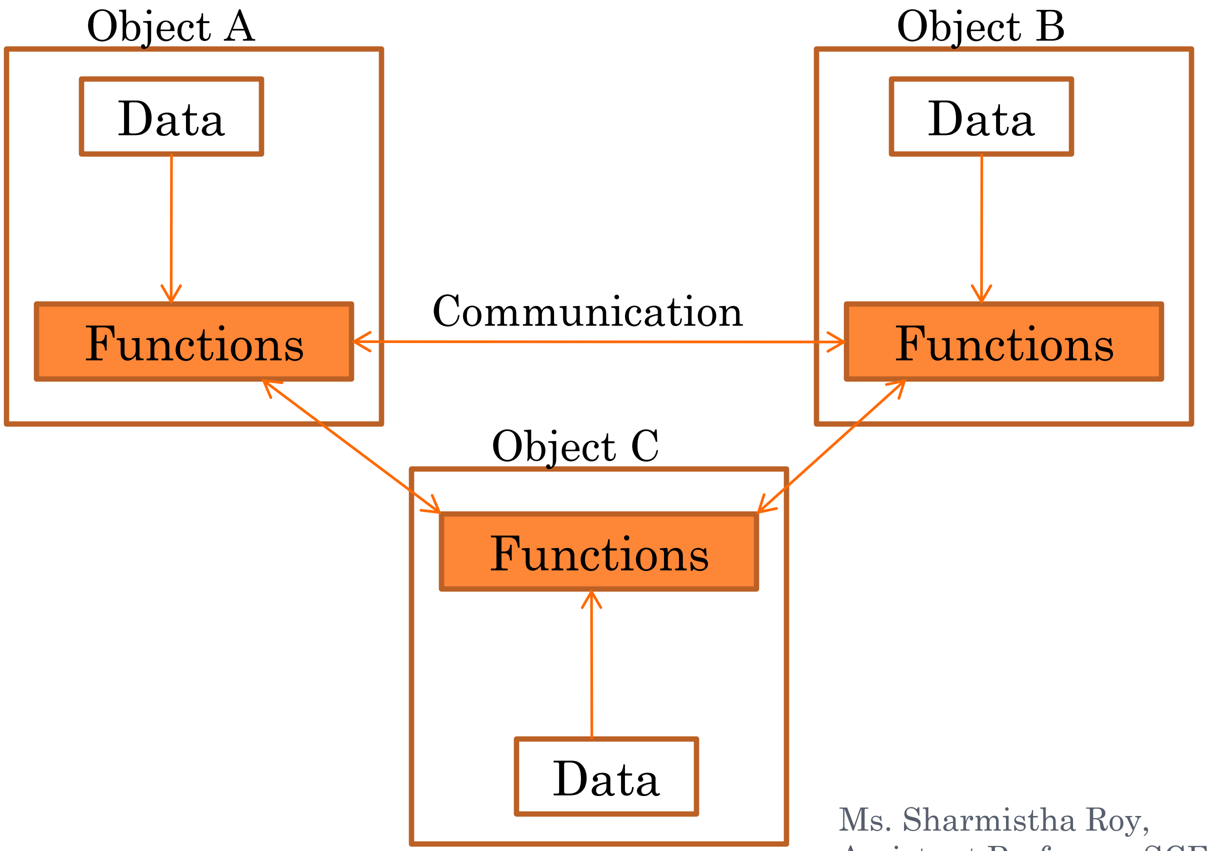
- Emphasis is on doing things i.e. algorithms.
- Large programs are divided into smaller programs known as functions.
- Most of the functions share global data.
- Data move openly around the system from function to function.
- Functions transform data from one form to another.
- Employs top down approach in program design.

DISADVANTAGES OF SPP..

- SPP lacks code reusability, extensibility and maintainability.
- Data and functions are stored separately and data is globally accessed as the systems are modularized on the basis of functions.
- Functions have unrestricted access to global data. Thus changing the global data in a module causes program side effects and that code becomes unreliable and error prone in a complex system.
- Information hiding and data encapsulation are not supported in SPP, therefore every function can access every piece of data.

Object Oriented Programming (OOP)

- It is an approach that provides a way of modularizing programs by creating partitioned memory area for both data and functions that can be used as templates for creating copies of such modules on demand.



Ms. Sharmistha Roy,
Assistant Professor, SCE , KIIT

Characteristics of OOPs..

- Emphasis is on data rather than procedure.
- Programs are divided into what are known as objects.
- Data structures are designed such that they characterize the objects.
- Functions that operate on the data of an object are tied together in the data structure.
- Data is hidden and cannot be accessed by external functions.
- Objects may communicate with each other through functions.
- New data and functions can be easily added whenever necessary.
- Follows bottom-up approach in program design.

Difference Between SPP & OPP

SPP	OOP
SPP is top-down approach	OPP is bottom-up approach
It is represented by logical entities and control flow.	It is represented with interacting objects and classes.
Here program modularisation is done on the basis of functions.	Here program modularisation is done on the basis of data structures called objects and classes
Data move openly around the system from function to function. By default, all data are public and hence provide global access.	Data is mostly hidden or permits restricted access due to public, private and protected rights. By default, all data are private and hence provide local access only.
Here decomposition of problem is functional approach.	Here decomposition of problem is object oriented approach.
It does not support code reusability.	Due to class hierarchy a part of the state and behaviour can be re-engineered to a subclass.
Functions and procedures share global data.	State and behaviour are tied together in a data structure called an object.

Elements of Object Oriented Programming

- Objects
- Classes
- Encapsulation & Data abstraction
- Inheritance
- Polymorphism
- Dynamic Binding
- Message Passing
- Information Hiding
- Overloading
- Exception Handling

OBJECTS

- Basic run-time entities in an object-oriented system i.e. fundamental building blocks for designing a software.
- It is a collection of data members and the associated member functions.
- An object represents a particular instance of a class. There can be more than one instance of an object.
- An object has three characteristics:
  - Name
  - State
  - Behaviour
- Objects take up space in the memory and have associated address.
- When a program is executed the objects interact by sending messages to one another.
- Example: Book, Bank, Customer etc.

CLASS

- Template for constructing objects.
- The entire set of data and code of an object can be made a user-defined data type with the help of a class.
- Defining a class doesn't create any objects.
- Objects are variables of the type class. Once a class has been defined we can create any number of objects belonging to that class.
- A class is thus a collection of objects of similar type.
- For e.g.: fruit mango;

  Here ‘fruit’ has been defined as a class and ‘mango’ is the object belonging to the class fruit.

Example Of Class & Object

```
class student_info
{
private:
    char name;
    int age;
public:
    void get_data();
    void display();
};

int main()
{
student_info s1;
s1.get_data();
}
```

Class name

List of attributes/ data members

Member functions

Object of the Class

Data Encapsulation

- Data encapsulation combines data and functions into a single unit called class.
- When using data encapsulation data is not accessed directly; it is only accessible through the methods present inside the class.
- Data encapsulation enables data hiding, which is an important concept possible of OPP.

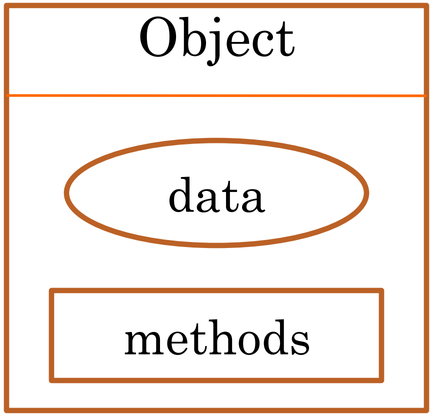


Fig: Data Encapsulation

Data Abstraction

- *Abstraction* refers to the act of representing essential features without including the background details or explanations.
- Data abstraction is an encapsulation of an object's state and behaviour.
- Data abstraction increases the power of programming languages by creating user-defined data types.
- Classes use the concept of abstraction and are defined as a list of abstract attributes and functions.
- Since classes use the concept of data abstraction, they are known as Abstract Data Types (ADT).

INHERITANCE

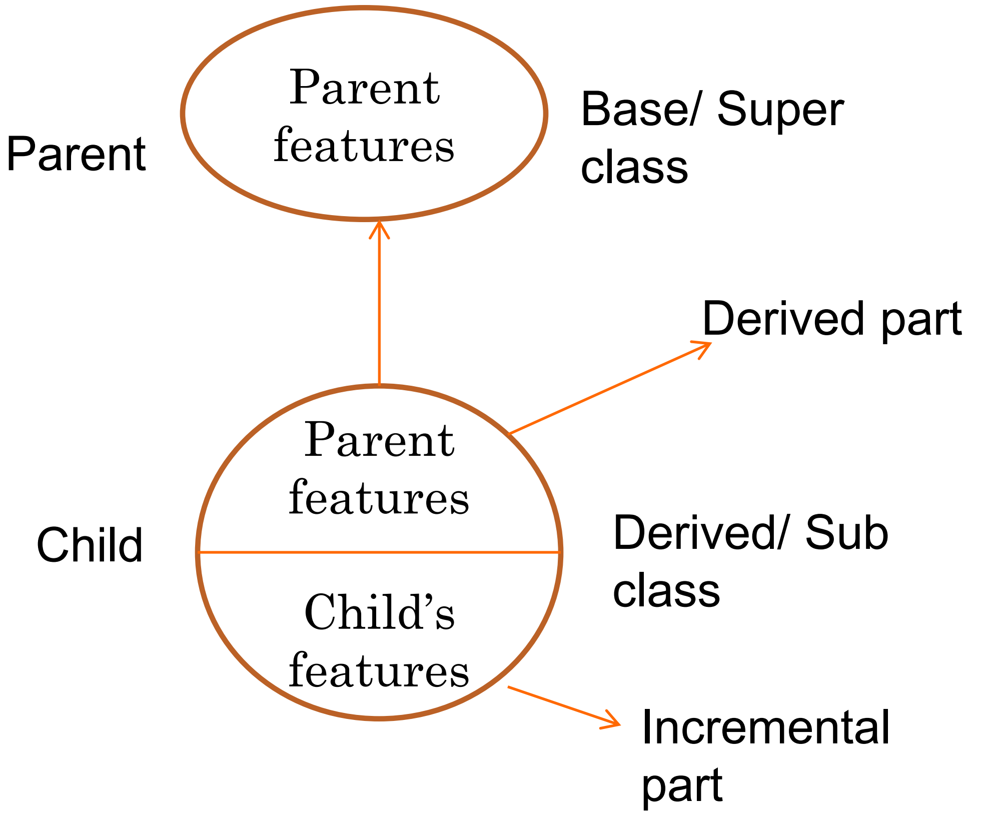
- It is the process by which one object can acquire the properties of another.
- It allows the declaration and implementation of one new class to be based on an existing class. The base class is known as parent class/super class & the derived class is known as child class/sub class.

Benefits:

- It helps to reuse an existing part rather than hand coding every time. Thus it increases reliability and decreases maintenance cost.
- It not only supports reuse but also directly facilitates extensibility within a given system.
- When software system is constructed out of reusable components, development time is less & product can be generated more quickly, easily and by rapid prototyping.

INHERITANCE..

- Example:



Parent

Parent features

Base/ Super class

Derived part

Child

Parent features

Child's features

Derived/ Sub class

Incremental part

Fig: Single Inheritance

Single Inheritance: where child inherits properties from only one base class

INHERITANCE..

- Example:

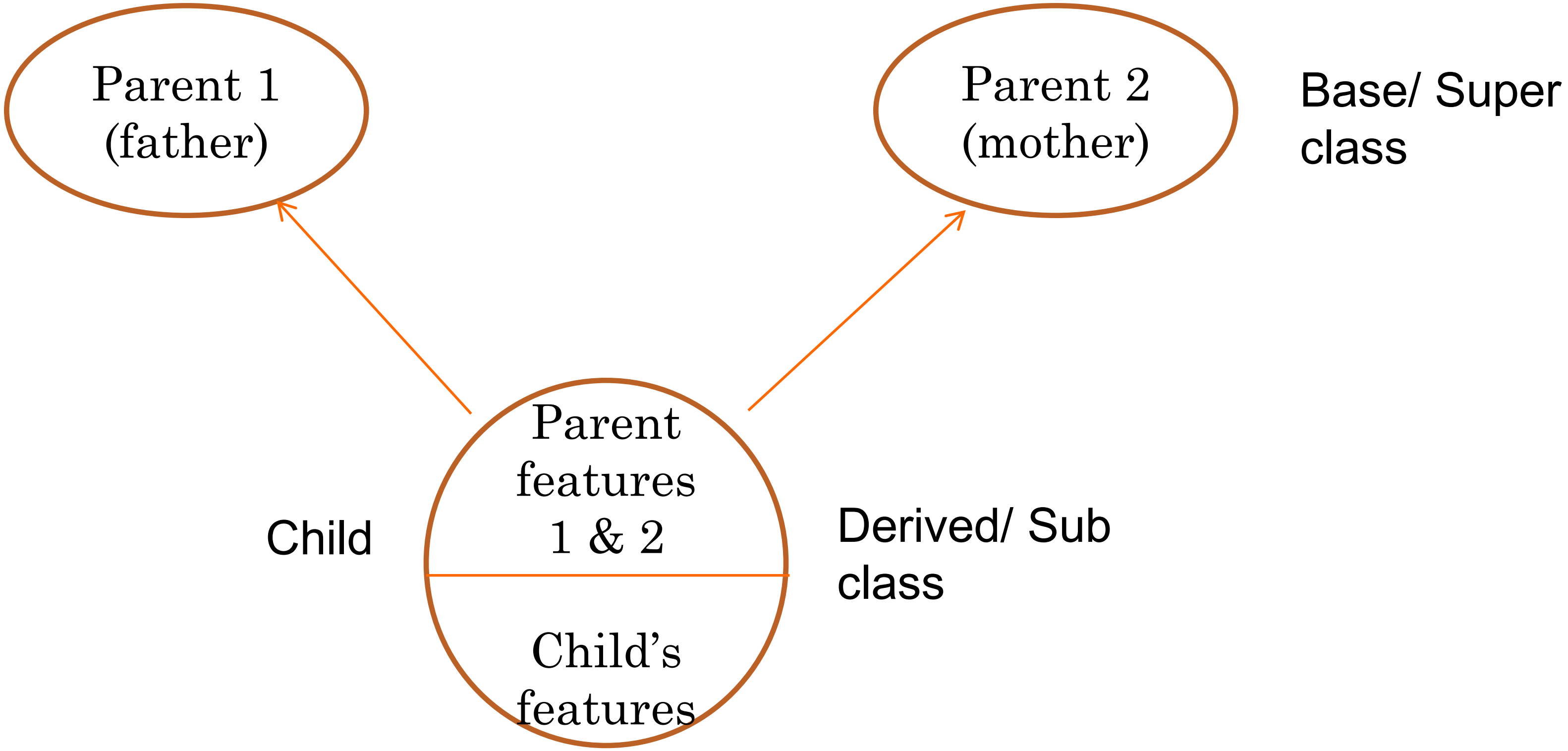


Fig: Multiple Inheritance

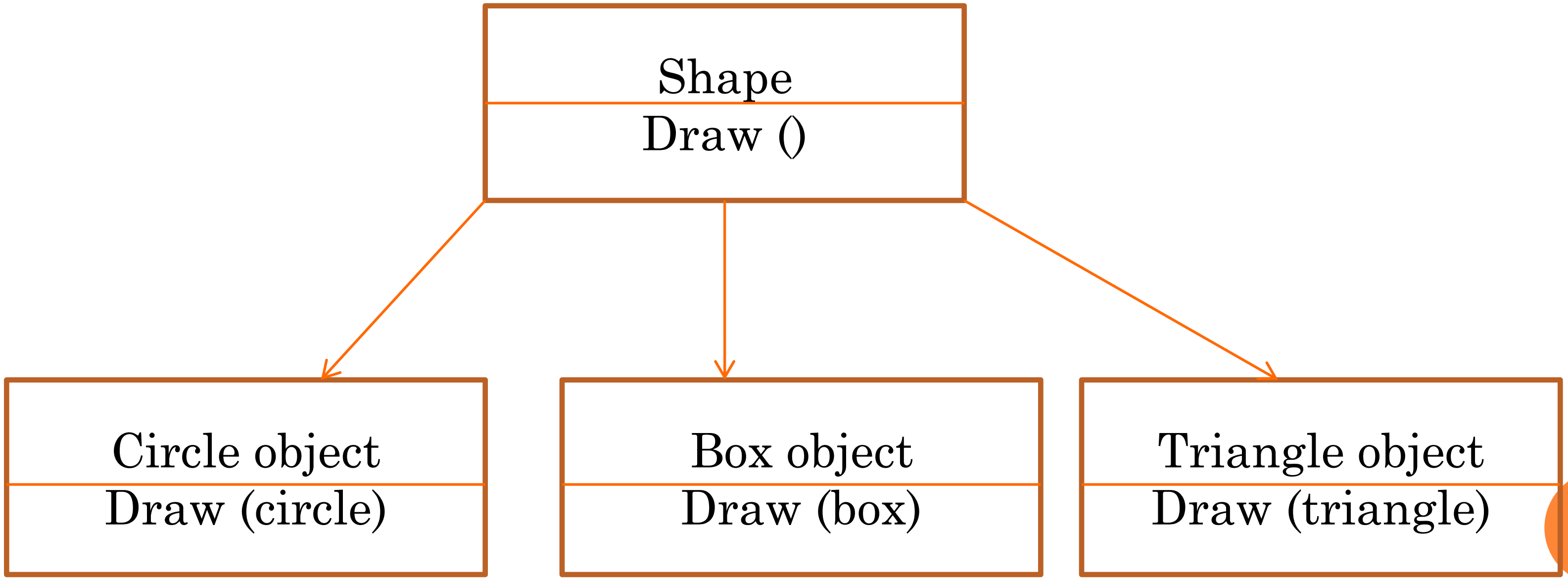
Multiple Inheritance: where derived class inherits features of more than one base class.

POLYMORPHISM

- The ability to take more than one form is known as Polymorphism.
- An operation may exhibit different behaviors in different instances. The behavior depends upon the types of data used in the operation.
- For e.g: "<<" operator is used in C++ as insertion operator i.e. for printing output on to screen. Whereas "<<" operator is also used as bit-wise left shift operator. This process of making an operator to exhibit different behaviors in different instances is known as *operator overloading*.
- Polymorphism plays an important role in allowing objects having different internal structures to share the same external interface.
- It is extensively used in implementing inheritance.

Polymorphism..

- A single function name can be used to handle different number and different types of arguments. This is similar to a particular word having several different meanings depending on the context. Using a single function name to perform different types of tasks is known as *function overloading*.



Dynamic Binding

- Binding refers to the linking of a procedure call to the code to be executed in response to the call.
- Dynamic binding(late binding) means that the code associated with a given procedure call is not known until the time of the call at run-time.
- Consider the fig shown in polymorphism, every object(circle/box/triangle) by inheritance have the procedure “draw”. Its algorithm is unique to each object and so the draw procedure is redefined in each class that defines the object. At run-time the code matching the object under current reference will be called.

MESSAGE PASSING

- In OOPs, processing is accomplished by sending messages to objects.
- The execution and implementation of messages are defined in the class methods.
- A message passing is equivalent to a procedure call or a function call of a procedure programming.

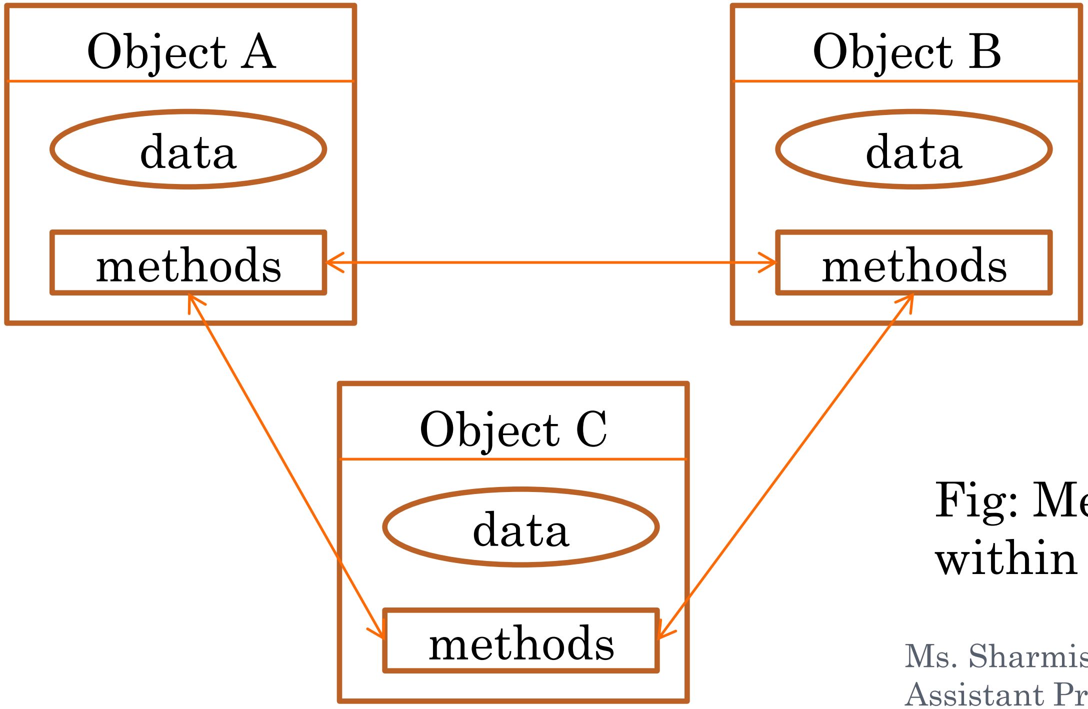
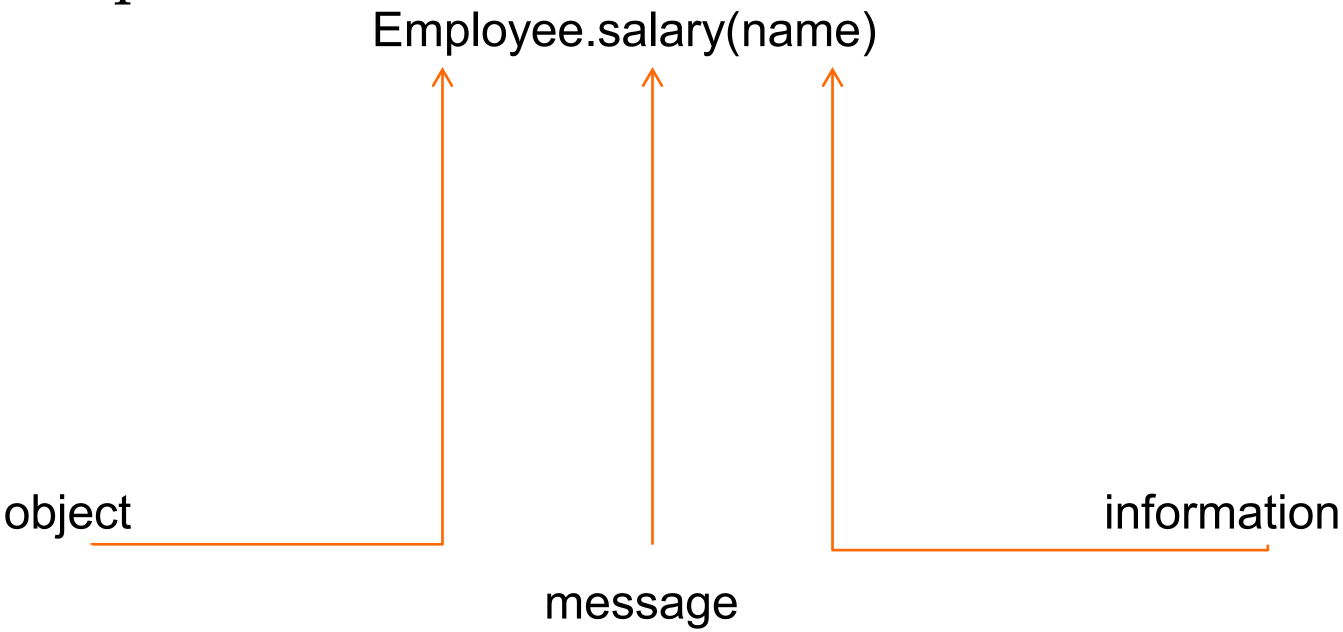


Fig: Message Passing within Objects

Message Passing..

- Message passing involves specifying the name of the object, the name of the function(message) and the information to be sent.
- Example:



Employee.salary(name)

object

message

information

Information Hiding

- Information hiding means that the implementation details of an object's state and behavior are hidden from users and other objects to protect the state and behavior from unauthorized access.

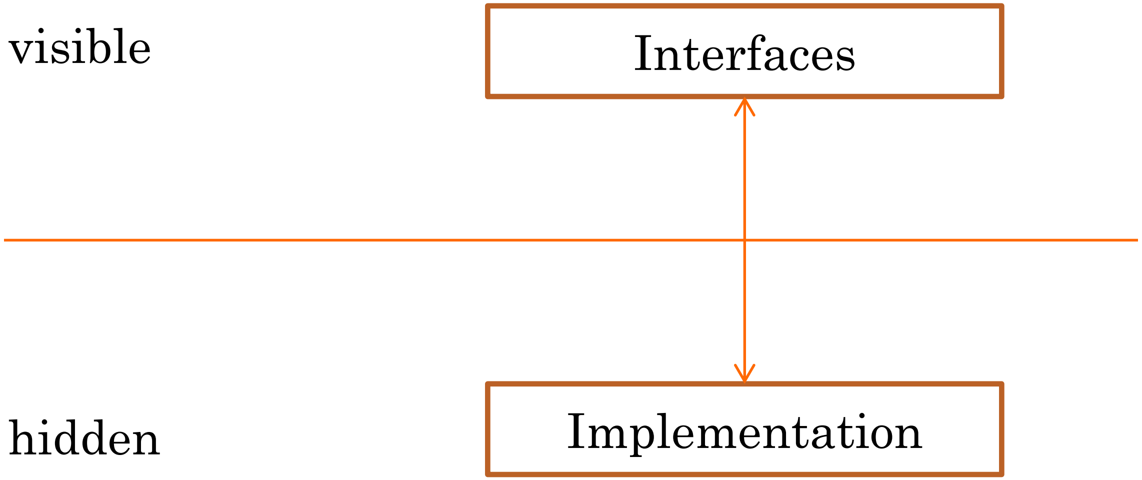


Fig: Information Hiding

ADVANTAGES OF OOPS

- Since it provides a better syntax structure, modelling real world problem is easy and flexible.
- Complex software systems can be modularised on the basis of class and objects.
- Data encapsulation and information hiding increases software reliability and modifiability. It do not allows unauthorized users to access the data.
- Polymorphism and dynamic binding increases flexibility of code by allowing the creation of generic software components.
- Inheritance allows software code to be extensible and reusable.

Applications Of OOPs

- Study yourself

INPUT & OUTPUT C++

Ms. Sharmistha Roy,
Assistant Professor,
School of Computer Engineering,
KIIT University

Main Function

C++ defines main() function as follows:

```
int main()
{
 // main program code
  return 0;
}
```

Restrictions of using main () function in C++

- ✓ Cannot be overloaded.
- ✓ Cannot be declared as inline.
- ✓ Cannot be declared as static.
- ✓ Cannot be called.

Simple C++ program

```
#include<iostream>
using namespace std;
int main()
{
cout<<“C++ is better than C\n”;
return 0;
}
```

- ✓ #include is a preprocessor directive which add the contents of the ‘iostream’ file to the program.
- ✓ iostream is the standard header file input and output stream which contains a set of small and specific general purpose functions for handling input and output data.
- ✓ Namespace defines a scope for the identifiers that are used in a program. For using the identifiers defined in the namespace scope we must include the ‘using’ directive.
- ✓ std is the namespace where ANSI C++ standard class libraries are defined.
- ✓ The ‘using’ namespace statement specifies that the members defined in ‘std’ namespace will be used frequently throughout the program. This will bring all the identifiers defined in ‘std’ to current global scope.
- ✓ cout is the identifier. It represents standard output stream.
- ✓ << operator is called the insertion or put to operator.

Stream based I/O

Streams refer to data flow. It is classified into

- ✓ Output Stream
- ✓ Input Stream

Output Stream: which allows to perform write operations on output devices. “cout” is an object of output stream.

e.g. of output stream operations:

```
✓    cout<<“KIIT University”;
     int age;
✓    cout<< age;
✓    cout<< “Age=“ <<age;
```

cascaded output operations: more than one item can be displayed using a single ‘cout’.

Input Stream: which allows to perform read operation with input devices. “cin” is an object of input stream.

e.g. of input stream operations:

```
     int age;
✓    cin>> age;
     char name[20];
✓    cin>> name;
✓    cin >> name >> age;
```

cascaded input operations: input of more than one item can be performed using a single ‘cin’.

Scope Resolution Operator ::

- It is used to access a global variable from a function in which a local variable defined with the same name as a global variable.
- Syntax: :: Global_VariableName
- e.g.

```
#include<iostream.h>
int num=20;
int main()
{
int num=10;
cout<<“Local= “<<num;
cout<<“Global= “<< ::num;
return 0;
}
```

Output:

```
Local= 10
Global= 20
```

Variable Definition at the Point of Use

- In C, local variable can only be defined at the top of a function or at the beginning of a nested block.
- Whereas in C++, local variables can be created at any position in code, even between statements. Also in some case, local variable can be defined prior to their usage.
- e.g.

```
int main()
{
for(int i=0;i<5;i++)
        cout<<“i=“<<i<<endl;
cout<<i;
return 0;
}
```

Output:

```
i=0
i=1
i=2
i=3
i=4
5
```

- Variable definition at any position in the code reduce the code readability. Therefore, local variable should be defined at the beginning or at right places.

Variable Definition at the Point of Use

```
#include<iostream.h>
    int i=10; //global variable
    int main()
    {
    cout<<i<<“\n”;           //uses global variable
    int i=20;
    {
    int i=30;
    cout<<i<<“\n”;           //uses locally defined variable within a block
    cout<<::i<<“\n”;         //uses global variable
    }
    cout<<i<<“\n”;           //uses local variable as defined near main()
    cout<<::i;               //uses global variable
    return 0;
    }
```

Output:

```
10
30
10
20
10
```

Variable Aliases- Reference Variables:

- Reference Variable is an alias of existing variable.
- In C, we have two types of variables:
  - Value variable: used to hold numeric values
  - Pointer variable: used to hold the address of some other variable
- Here in C++, Reference variable behaves similar to value variable but has an action of pointer variable. Thus reference variable enjoys the simplicity of value variable and power of the pointer variable.
- Syntax: DataType & ReferenceVariable = ValueVariable;
- Reference variables are not bound to a new memory location, but to the variables to which they are aliases.

```
int main()
    {
    int a=1, b=2;
    int &c= a;
    cout<<“a=“<<a<<“b=“<<b<<“c=“<<c<<endl;
    c=b;
    cout<<“a=“<<a<<“b=“<<b<<“c=“<<c;
    return 0; }
```

Output:

```
a=1 b=2 c=1
a=2 b=2 c=2
```

Variable Aliases- Reference Variables:

- Therefore, if int &c=a; then address of c & a is same.
- It doesnot provide the flexibility supported by the pointer variable. Unlike pointer variable, when a reference is bound to a variable, then its binding cannot be changed.
- Thus, reference variable must be initialized to some variable only at the point of its declaration. For e.g.

```
int main()
{
        int a=1, b=2 &c ; //error shows that reference variable must be initialized
        &c= a;
        return 0;
}
```

- Constants cannot be pointed to a reference variable. For e.g. int &num = 200; is invalid.

Variable Aliases- Reference Variables:

- For e.g.

```
int main()
{
        int a=1;
        int *p=&a;
        int &m=*p;
        int k=100;
        p=&k;
        return 0;
}
```

- Here m refers to a, which is pointed to by the variable p. The compiler binds the variable m to a not to the pointer p. If pointer p is bound to some other variable at run time then it does not affect the value referenced by m & a.
- Therefore here, p=&k, changes the pointer value but not the reference variable m & the variable a.

Strict Type Checking:

✓ C++ is a strongly-typed language and it uses very strict type-checking. A prototype must be known for each function which is called, and the call must match the prototype.

✓ In C++, functional prototyping is compulsory if the definition is not placed before the function call whereas, it was not compulsory in original C language.

✓ For e.g.

```
int main()
{
  int x, y;
  cout<<“Enter x & y”;
  cin>>x>>y;
  cout<<“Maximum= “<<max(x, y);    // Error: Function ‘max’ should have a prototype
  return 0;
  }
int max(int a, int b)
  {
    if(a>b)
            return a;
     else
            return b;
   }
```

Strict Type Checking:

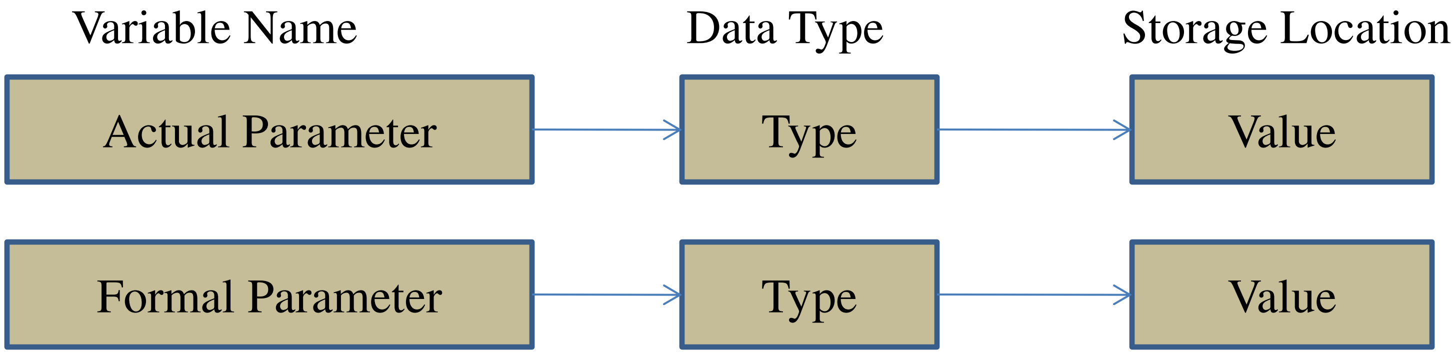
✓ Advantage of strict type-checking is that the compiler warns the user if a function is called with improper data types. It helps the user to identify errors in a function call & increases the reliability of a program.

✓ For e.g.

```
    void swap(int *x, int *y)
      {
            int t;
            t= *x;
            *x=*y;
            *y=t;
      }
    main()
     {
            int a,b;
            swap(&a, &b);
            float c,d;
            swap(&c, &d); //Error because pointers are not integer data type
     }
```

Parameters Passing by Reference:

- ✓ A function in C++ can take arguments passed by value, by pointer, or by reference.
- ✓ A copy of the actual parameters in the function call is assigned to the formal parameters in the case of pass-by-value, whereas the address of the actual parameters is passed in the case of pass-by-pointer.
- ✓ In the case of pass-by-reference, an *alias of the actual parameter* is passed.



Variable Name

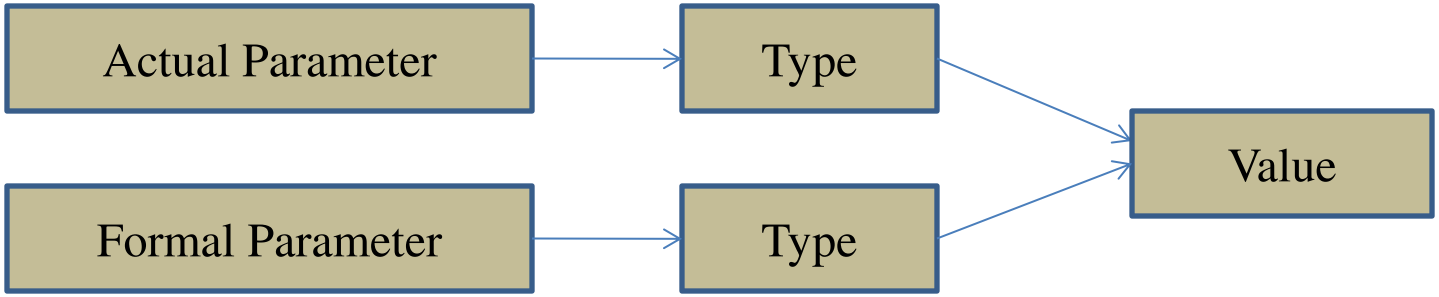
Data Type

Storage Location

Actual Parameter → Type → Value

Formal Parameter → Type → Value

Fig: Call-by-value



Actual Parameter → Type → Value

Formal Parameter → Type → Value

Fig: Call-by-pointer/ reference

Parameters Passing by Reference:

- For e.g.

Pass-by-pointers

```
void swap(int *x, int *y)
            {
            int t;
            t= *x;
            *x=*y;
            *y=t;
            }
main()
 {
   int a,b;
   swap(&a, &b);
 }
```

Pass-by-reference

```
void swap(int &x, int &y)
             {
                int t;
                t= x;
                x=y;
                y=t;
             }
main()
  {
    int a,b;
    swap(a, b);
   }
```

- Here in the 2nd example, the statement swap(a,b) in main() is translated in to swap(&a, &b) internally during complilation.
- The prototype of the function void swap(int &x,int &y) indicates that the formal parameter are of reference type and hence they must be bound to the memory location of the actual parameter. Thus any access made to reference formal parameters in swap() refers to the actual parameters.
- Thus the compiler treats them similar to pointers but doesnot allow the modification of the address stored in them.

Inline Function:

- Function execution involves the overhead of jumping to and from the calling statement.
- Trading of this overhead in execution time is considerably large whenever a function is small, hence in such cases inline function can be used.
- A function in C++ is treated as a macro if the keyword inline precedes its definition. The drawback in macro is that they are not really functions, therefore the usual error checking doesnot occur during compilation.
- Syntax is:

```
inline ReturnType FunctionName (Parameters)
          {
          //    body of a main function
          }
```

- The advantage is that the compiler replaces the function call with the corresponding function code.
- The support of inline functions allow to enjoy the flexibility and benefits of modular programming, while at the same time delivering computational speedup of macros.

Inline Function:

- ✓ The speed benefits of inline functions diminish as the functions grows in size.
- ✓ Some of the situations where inline expansion may not work are:
  - ✓ For functions returning values, if a loop, a switch, or a goto exists.
  - ✓ For functions not returning values, if a return statement exists.
  - ✓ If functions contain static variables.
  - ✓ If inline functions are recursive.

Literals- Constant Qualifiers:

- ✓ Literals are constants, to which symbolic names are associated for the purpose of readability and ease of handling standard constant values.
- ✓ C++ provides the following three ways of defining constant:
- ✓ # define preprocessor directive
- ✓ Enumerated data types
- ✓ const keyword
- ✓ In C, variables can be initialized with constant values at the point of its definition like: float PI = 3.1452;
- ✓ However, an accidental change of the value of the variable PI is not restricted by C.
- ✓ But, C++ overcomes this by supporting a new constant qualifier for defining a variable, whose value cannot be changed once it is assigned with a value at the time of variable definition.
- ✓ The qualifier used in C++ to define such variables is the ***const*** qualifier.

Literals- Constant Qualifiers:

- ✓ Syntax: *const [DataType] VariableName = ConstantValue;*
- ✓ Note that if DataType is ommitted, it is considered as int by default.
- ✓ E.g. #include <iostream>

```
const float PI = 3.1452;
int main()
{
float radius, area;
cout<<"Enter radius";
cin>> radius;
area= PI * radius * radius;
cout<<" Area of circle is "<< area;
}
```

- ✓ In the above program, the use of statement like: PI = 2.3; leads to compilation error: Cannot modify a const object.
- ✓ In C++, the const qualifier can be used to indicate the parameters that are to be treated as read-only in the function body.

Function Overloading:

- The definition of several functions with the same name, but with different actions is called function overloading.
- In C++, two or more functions can be given the same name provided each has a unique signature (in either the number or data type of their argument).
-
```
void show (int val)
{
        cout<<“Integer: “<< val <<endl;
}
void show (char *val)
{
        cout<<“String: “<<val;
}
int main()
{
        show (5);
        show (“Hello World”);
        return 0;
}
```

Output:

```
Integer: 5
String: Hello World
```

Function Overloading:

- ✓ It is interesting to note the way in which the C++ compiler implements function overloading.
- ✓ Although the functions share the same name in the source text, the compiler uses different names.
- ✓ The conversion of a name in the source file to an internally used name is called *name mangling*.
- ✓ For e.g. in the previous program, the C++ compiler might convert the name void show(int) to the internal name VshowI, while an analogous function void show (char *) with a character argument might be called VshowCP.
- ✓ The actual names which are used internally depend on the compiler and are not relevant to the programmer.
- ✓ ***Advantage:*** Function overloading helps in reducing the need for unusual function names, making code easier to read.

Function Overloading:

- Overloading of the functions should be done with cautions. We should not overload unrelated functions.
- Function overloading can lead to surprises. For e.g.
-
  ```
  void show (float val)
  {
          cout<<“Integer: “<< val <<endl;
  }
  void show (char *val)
  {
          cout<<“String: “<<val;
  }
  int main()
  {
          show (0);  // Error: Ambiguity between ‘show(float)’ and ‘show(char *)’
          return 0;
  }
  ```
- Here zero could be interpreted as a NULL pointer to a char i.e. (char *) 0 or as a float with the value zero.

Default Arguments:

- C++ allow us to call a function without specifying all its arguments. In such cases, the function assigns a default value to the parameter which does not have a matching argument in the function call.
- Default values are specified when the function is declared.
- The compiler looks at the prototype to see how many arguments a function uses and supply the default values when they are not specified by the programmer explicitly. For e.g.

default value

```
#include<iostream.h>
float amount (float principal, int period, float rate= 0.15);
int main()
{
   float value;
   value= amount(5000,7);   // Here the function uses default value of 0.15 for rate.
   value= amount(5000,5,0.12); // Function passes an explicit value of 0.12 for rate
   return 0;
}
```

Default Arguments:

- ✓ Default argument is checked for type at the time of declaration and evaluated at the time of call.
- ✓ Default arguments must be known to the compiler prior to the invocation of a function with default arguments.
- ✓ It reduces the burden of passing arguments explicitly at the point of a function call.
- ✓ One important point to note is that, only trailing arguments can have default values and therefore, we must add defaults from right to left.
- ✓ We cannot provide a default value to a particular argument in the middle of an argument list.
- ✓ int mul (int i, int j=5; int k=10); *// legal*
- ✓ int mul (int i=5, int j); *// illegal*
- ✓ int mul (int i=0, int j; int k=10); *// illegal*
- ✓ int mul (int i=2, int j=5; int k=10); *// legal*

Classes & Objects

Ms. Sharmistha Roy
Assistant Professor,
School of Computer Engineering,
KIIT University

Classes

- A class is an expanded concept of a data structure, instead of holding only data, it can hold both data and functions.
- It is a mechanism for creating user-defined data types. It defines a new type like int, float, char etc.
- It is similar to the C language structure data type. In C, a structure is composed of a set of data members. Whereas in C++, a class is composed of a set of data members and a set of operations that can be performed on the class.
- Classes are generally declared with the keyword ‘*class*’.

```
class class_name
{
access_specifier:
          data_member;
access_specifier:
          member_function();
};
```

Class consists of 3 things:

a) Data member
b) Member function
c) Access specifier

Access control specifier:

- ✓ Access specifier specify the access rights that the data_members & member_functions following them acquire. An access specifier is one of the following three keywords:
  - ✓ private:
  - ✓ public:
  - ✓ protected:
- ✓ Mechanism to achieve *Information Hiding* i.e. to prevent the internal representation from direct access from outside the class
- ✓ Public
  - – Can be accessible from anywhere in the program
- ✓ Private (default)
  - – Can be accessible only from member functions and friend functions*
- ✓ Protected
  - – Acts as *public* for objects of its own class and derived classes
  - – Acts as *private* to rest of the program
- ✓ By default the access specifier of data members & member functions in a class is private. And that of structure is public.

* Friend function is a function which can access private data members of a class, even though the function itself is not a member of the class

Difference between Structure & Class

Structure	Class
Structure is composed of data only	Whereas class is composed of both data and member functions.
By default the access specifier of all the data members of structure is public	By default the access specifier of data members & member function is private.

Objects

- An object is an instantiation of a class. In terms of variables, a class would be the type, and an object would be the variable.
- For e.g. object can be created in two ways:

```
class person
        {
         private:
               char name[20];
               int age;
         public:
               display();
         }p1;
```

```
int main()
{
person p1;
p1.display();
return 0;
}
```

- Now the relationship between class and object is same as int and age. Here person is the class(i.e. type) and p1 is the object(i.e. variable) of this class.
- A class can have multiple objects. So we can say that a class is a collection of objects of same type.

How to access data members:

✓ *Accessing Public data members:*

Public data members can be accessed using direct member access operator(.) e.g.:

```
class box
{
public:
int height;
int length;
};

int main()
{
box b1;
b1.height=5;
b1.length=6;
return 0;
}
```

✓ *Accessing Private data members:*

Private data members can be accessed using the member function of that class. e.g.:

```
class box
{
private:
int height;
int length;
public:
void set()
{
height=5;
length=6;
}
};
```

```
int main()
{
box b1;
b1.height=10;// not allowed
b1.set();
return 0;
}
```

Member functions:

- ✓ Member functions can be defined inside a class & outside a class.
- ✓ *Defining a member function inside a class:*

```
class Rectangle
{
      private:
         int width, length;
      public:
       void set(int w, int l)
                {
                     width = w;
                     length = l;
                }
      int area()
      {
           return width*length;
      }
};
```

```
int main()
{
      Rectangle rect;
      rect.set(3,5);
     cout<<rect.area();
     return 0;
}
```

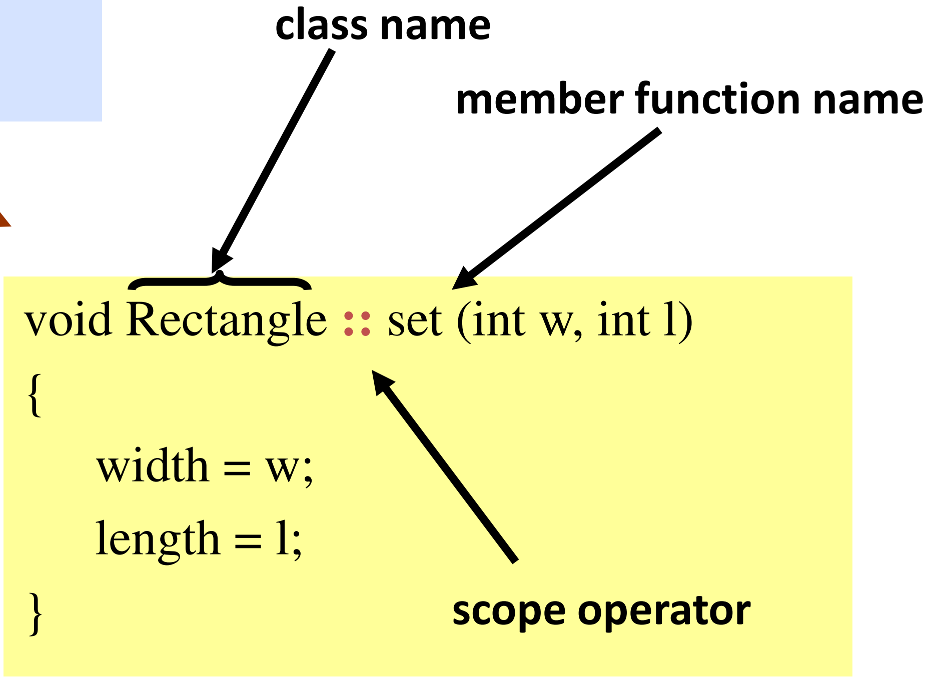
Member functions:

✓ *Defining a member function outside a class:*

```
class Rectangle
{
    private:
        int width, length;
    public:
        void set(int w, int l);
        int area(){return width * length;}
};
```

```
int main()
{
    Rectangle rect;
    rect.set(3,5);
    set(3,5); // not allowed
    return 0;
}
```

Scope operator(::) is used to define a member of a class from outside the class definition itself. The function that is defined with :: is a member of class & not a global function.



```
void Rectangle :: set (int w, int l)
{
    width = w;
    length = l;
}
```

Ms. Sharmistha Roy,
Assistant Professor, SCE , KIIT

Difference between member function inside & outside a class:

- The scope properties of both the member functions is same. Both the member functions can access the private data members.
- The only difference is that the member function defined inside a class will automatically be considered as *inline* member function by compiler. While the other will be considered as normal(not-inline) class member function, which in fact supposes no difference in behavior.
- *Inline function* is a function which means that the code generated by the function body is inserted at each point the function is called, instead of declaring a function once and performing a regular call to the definition somewhere else.
- Advantage of using inline function is to reduce the additional overhead in running time.

Nesting of a member functions:

- ✓ We know that, a member function of a class can be called only by an object of that class using a dot operator.
- ✓ However, there is an exception to this. A member function can be called by using its name inside another member function of the same class. This is known as ***nesting of member functions***.

Nesting of member function:

```
class Code
{
 int m, n;
 public:
   void input();
   void display();
   int largest();
};

 int main()
 {
     Code c;
     c.input();
     c.display();
     return 0;
 }
```

```
int Code :: largest()
  {
    if(m>=n)
     return m;
    else
     return n;
   }
void Code :: input()
   {
     cout<< “Enter values of m & n:”;
     cin>> m >> n;
    }
void Code :: display()
    {
      cout<<“Largest value:” << largest( ) << “\n”;
     }
```

Output:

```
Enter values of m & n:
25 15
Largest value: 25
```

Constant Member Function:

- ✓ A member function which is declared with the *const* keyword and which cannot make any modification to the class data members is known as *constant member function*.
- ✓ ***Declaration:***
  - ✓ return_type func_name (para_list) const;
- ✓ ***Definition:***
- ✓ *Inside the class:*
  - ✓ return_type func_name (para_list) const { … }
- ✓ *Outside the class:*
  - ✓ return_type class_name :: func_name (para_list) const { … }

Constant Member Function:

```
class A
{
    private:
      int a,b;
    public:
      void set( );
       {
           a=5;b=6;
        }
      void show( ) const;
};
```

```
int main()
{
     A a;
     a.set( );
     a.show( );
     return 0;
}
```

```
void A :: show( ) const
{
     a++; // Error: Cannot modify a const object
     cout<< “a= “<<a<<endl;
     cout<<“b=“ <<b<<endl;
}
```

```
Output:
a= 5
b= 6
```

Memory Utilization by class & object:

- ✓ When we declare a class it does not allocate memory to the class data members.
- ✓ Memory allocated, for data members only by the statement of object creation but not for member functions.
- ✓ So, different objects of a class have different set of data members.
- ✓ Member function are created and allocated only when a class declared.
- ✓ All the objects of a class share same location to access the member functions.
- ✓ Hence there is no separate copy of member functions for separate objects of a class.

How to prove that different objects of a class have different data members but same member function?

```
class Rectangle
{
    private:
      int width, length;
    public:
      void set(int w, int l);
      int area(){
    cout<<“address width=“<<&width;
    cout<<“address length=“<<&length;
    return width * length;}
};
```

```
void Rectangle :: set (int w, int l)
{
    width = w;
    length = l;
}
```

```
int main()
{
    Rectangle r1, r2;
    r1.set(3,5);
    cout<<“area of r1=”<<r1.area();
    r2.set(4,6);
    cout<<“\narea of r2”<< r2.area();
    return 0;
}
```

Output:

```
Address width= 0x8fd5fff2
Address length= 0x8fd5fff4
area of r1=15
Address width= 0x8fd5ffee
Address length= 0x8fd5fff0
area of r2=24
```

Static Class Members:

- ✓ In C++, static member of a class can be categorised into two types: static data member & static member function.
- ✓ Whenever, a data or member function is declared as a static type, it belongs to a class, not to the instances or objects of the class.
- ✓ Both data member & member function can have the keyword static.
- ✓ ***Static Data member:***
- ✓ Static data members are data objects that are common to all the objects of a class. They exist only once in all objects of this class.
- ✓ The access rule of the data member of a class is same for the static data member also.
- ✓ The static data member should be defined outside the class definition and before the main() function control block.

Static Data Member:

- ✓ For e.g.

```
class sample
{
private:
        static int sum;
public:
        void display();
};
int sample :: sum= 9;
void sample :: display()
{
        cout<<“Content of static data member is “<< sum;
}
```

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

Output:

```
Content of static data member is 9
```

Static Data Member:

```
class sample
{
static int count;
public:
        sample();
        void display();
};
int sample :: count= 0;
sample :: sample()
{
        ++count;
}
void sample :: display()
{
        cout<<“Counter value is “<< counter;
}
```

```
int main()
  {
    sample obj1, obj2,obj3,obj4;
    obj4.display();
    return 0;
   }
```

Output:

```
Counter value is 4
```

Static Class Members:

- ✓ ***Static Member Function:***
- ✓ The keyword static is used to precede the member function to make a member function static.
- ✓ The static function is a member function of the class and can manipulate only on static data member of the class.
- ✓ A static member function is not part of objects of a class.
- ✓ A static member function cannot have an access to ‘this’ pointer of the class.
- ✓ A static and nonstatic member function cannot have the same name and the same arguments type.
- ✓ It cannot be declared with the keyword const.
- ✓ It can be directly called by using the class name and the scope resolution operator.
- ✓ The main usage of static function is when the programmer wants to have a function which is accessible even when the class is not instantiated.

Static Member Function:

```
class sample
{
static int count;
public:
        sample();
        static void display();
};
int sample :: count= 0;
sample :: sample(){
++count; }
void sample :: display()
{
        count=count+1;
        cout<<"Counter value is "<< count;
}
```

```
int main()
  {
    sample obj1, obj2,obj3,obj4;
    obj4.display();
    return 0;
   }
```

Output:

```
Counter value is 5
```

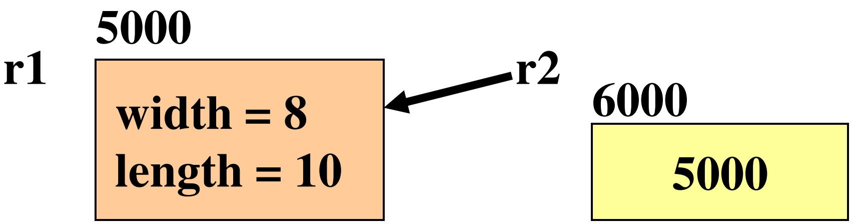
Declaration of an object:

```
class Rectangle
{
   private:
      int width;
      int length;
   public:
      void set(int w, int l);
      int area();
}
```

r2 is a pointer to a Rectangle object

```
main()
{
     Rectangle r1;
     r1.set(5, 8);          //dot notation

     Rectangle *r2;
     r2 = &r1;
     r2->set(8,10);         //arrow notation
}
```



Array of Objects:

- ✓ We know that array can be of any data type including struct. Similarly, we can also have arrays of variables that are of the type ***class***. Such variables are called as ***array of objects***.
- ✓ For e.g.

```
class employee
{
        char name[30];
        int age;
public:
        void getdata( );
        void putdata( );
};
```

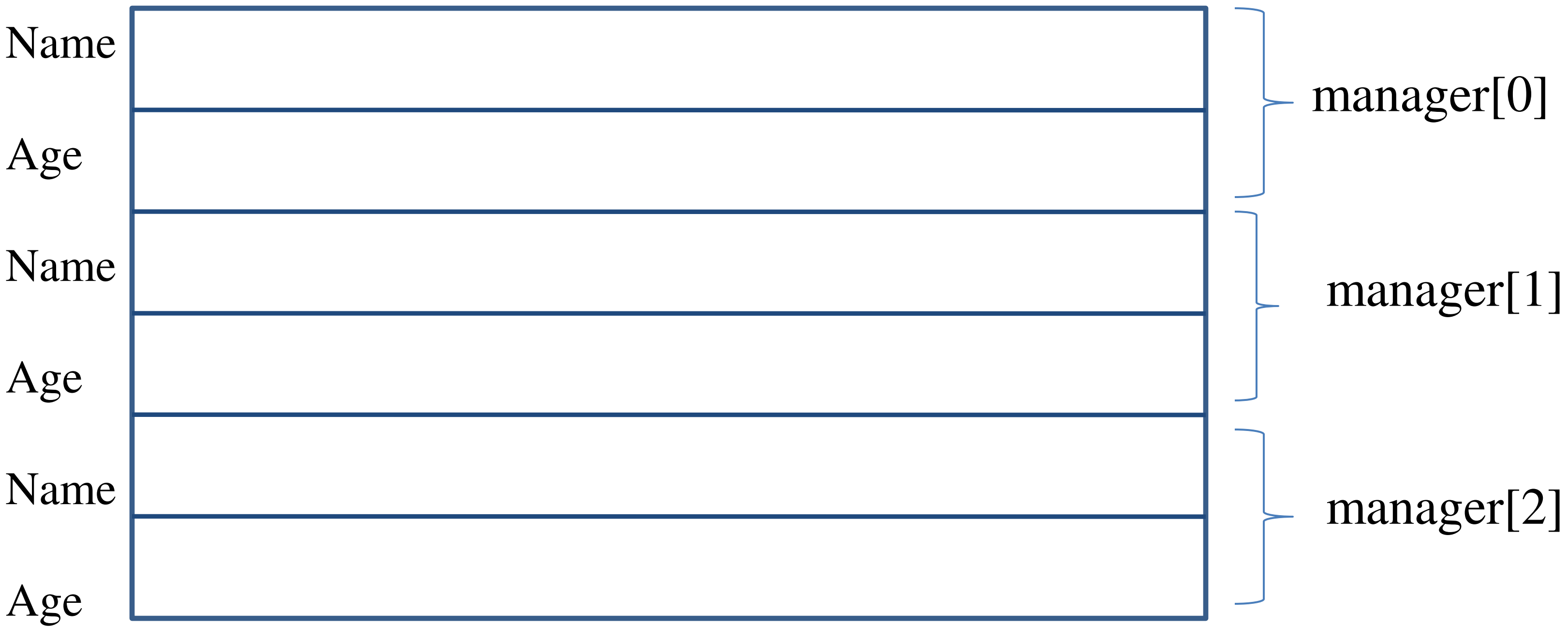
```
employee manager[3];
employee worker[20];
```

Array accessing method to access individual elements & the dot operator to access the member function

```
manager[i].putdata( );
```

Memory allocation of array of objects:

- ✓ An array of objects is stored inside the memory in the same way as a multi-dimensional array.
- ✓ The array manager is represented as:



Name
Age
Name
Age
Name
Age

manager[0]
manager[1]
manager[2]

Fig: Storage of data items of an object array

Object as function argument:

✓ Similar to other variables, object can be passed on to functions. As in normal case object can be passed in two ways:

1) pass by value

2) pass by reference

Pass by Value:

```
class sample
{
int a, b, sum, mul;
public:
void getdata (int c, int d)
{ a=c; b=d; }
void show(){ cout<<a<<endl<<b;}
void display(sample x, sample y)
{ x.a= 10;
sum=x.a+y.a;
mul=x.b*y.b;
Cout<<“Sum=“<<sum<<“\n Multiply=“<<mul;
}};
```

```
int main()
{
sample s1,s2;
s1.getdata(2,3);
s1.show();
s2.getdata(4,5);
s2.show();
s1.display(s1,s2);
s1.show();
}
```

Output:

```
2   3
4   5
Sum= 14
Multiply= 15
2   3
```

✓ The object will be *copied* into the function parameter so, much like basic types, any attempt to modify the parameter will modify only the parameter in the function and won't affect the object that was originally passed in.

Pass by Reference:

```
class sample
{
int a, b, sum, mul;
public:
void getdata (int c, int d)
{ a=c; b=d; }
void display (sample &x, sample &y)
{
x.a=10;
sum=x.a+y.a;
mul=x.b*y.b;
cout<<“Sum=“<<sum<<“\nMultiply=“<<mul;
}
void show(){
cout<<“A=“<<a << “B=“<<b;
} };
```

```
int main()
{
sample s1,s2;
s1.getdata(2,3);
s1.show();
s2.getdata(4,5);
s2.show();
s1.display(s1,s2);
s1.show(); }
```

Output:

```
A= 2       B=3
A=4        B=5
Sum= 14  Multiply= 15
A= 10     B=3
```

- ✓ This syntax means that any use of object in the function body refers to the actual object which was passed into the function and not a copy. All modifications will modify this object and be visible once the function has completed.

new Operator:

- The new operator offers dynamic storage allocation similar to the standard library function malloc.
- It throws an exception if memory allocation fails.
- Syntax:

pointer_variable = new DataType ;

- The operator new allocates a specified amount of memory during runtime and returns a pointer to that memory location.

$$\textit{pointer\_variable = new DataType [}\overbrace{\textit{size in integer}}^{\text{Number of items to be allocated}}\textit{];}$$

- **p= new int;** // here p is a pointer of type int
- **int *p = new int;** // declaration of pointer & their assignment

  ***p = 25;** // assign 25 to the newly created int object
- **int *p = new int(25);** // directly we can initialize the memory
- **int *a= new int[100];** // 100 is the size of the array.
- **int *a;**

  a = new int[100];

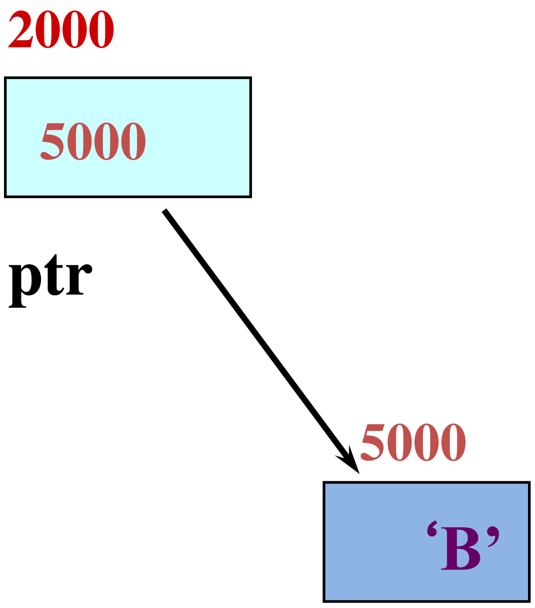
Operator new

```
char*  ptr;

ptr = new char;

*ptr = ‘B’;

cout  <<  *ptr;
```



2000

5000

ptr

5000

‘B’

delete Operator:

- The new operator's counter part, delete ensures the safe and efficient use of memory. It is similar to free in C.
- This operator is used to return the memory allocated by the new operator back to the memory pool.
- The memory allocated using the new operator should be released by the delete operator.
- Syntax:

  Pointer returned through new operator

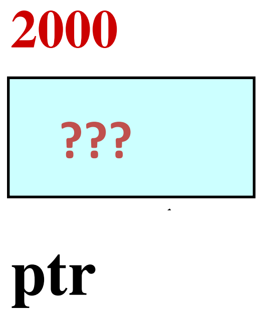
  delete PointerVariable; **or** ***delete [] pointer;*** *[delete an array]*
- Although the memory allocated is returned automatically to the system, when the program terminates, it is safer to use this operator explicitly within the pointer. For e.g. in situations, where local variables pointing to the memory get destroyed when the function terminates, leaving memory inaccessible to the rest of the program.

Operator delete

```
char*  ptr;


ptr = new char;
*ptr = ‘B’;
cout  <<  *ptr;


delete  ptr;
```



2000

???

ptr

NOTE:

delete deallocates the memory pointed to by ptr

Friend Function:

- A friend function is a function which precedes the keyword ‘friend’ and it can access the private and protected data members of a class even if it is not a member function of that class.
- Syntax: ***friend return_type function_name (parameters);***
- A friend function may be either declared or defined within the scope of a class definition.
- If the friend function is declared within the class, it must be defined outside the class, but should not be repeated the keyword friend.
- The keyword ‘friend’ inform the compiler that it is not a member function of the class.

Friend Function:

Friend function within Class definition:

```
class alpha
{
int x;
public:
void getdata();
friend void display (alpha abc)
 {
  cout<<“value of x=“<<abc.x;
 }
};
```

```
void alpha :: getdata()
{
cout<<“Enter the value of x”;
cin>>x;
}
int main()
{
alpha al;
al.getdata();
display(al);
}
```

Friend function outside Class definition:

```
class alpha
{
int x;
public:
void getdata();
friend void display (alpha abc);
};
void display (alpha abc)
 {
  cout<<“value of x=“<<abc.x;
 }
```

Friend Function:

- ✓ Member function of one class can be friend functions of another class.
- ✓ In such cases, they are defined using the scope resolution operator as shown below:
- ✓ class X

```
class X
{
        ……
        public:
        int func();              // member of class X
};
class Y
{
        ……
        friend int X :: func();       // func() of X is friend of Y
};
```

- ✓ Means func() of X can access the private data members of class Y, although it is not a member function of Y.

Granting Friendship to another Class:

- ✓ We can also declare all the member functions of one class as the friend functions of another class. In such cases, the class is called a **friend class**.
- ✓ A class can have friendship with another class.
- ✓ For example, let there be two classes, first and second.
- ✓ If the class first grants its friendship with the other class second, then the private data members of the class first are permitted to be accessed by all public member functions of the class second.
- ✓ But on the other hand, the public member functions of the class first cannot access the private members of the class second.

Granting friendship with another class

```
class first
{
friend class second;
int x;
public:
void getdata();
};
class second
{
public:
void display(class first temp);
};
void first :: getdata()
{
cout<<“enter a number”;
cin>>x;
}
void second :: display( class first temp)
{
cout<<“Entered number is-”<<temp.x;
}
```

```
int main()
{
first obj1;
second obj2;
obj1.getdata();
obj2.display(obj1);
return 0;
}
```

Output:

```
Enter  a number
5
Entered number is-5
```

Constructors
&
Destructors

Ms. Sharmistha Roy
Assistant Professor,
School of Computer Engineering,
KIIT University

Constructor:

- A constructor is a special member function whose main operation is to allocate the required resources such as memory and initialize the objects of its class.
- A constructor is distinct from other member functions of the class, and it has the same name as its class.
- It is the first member function to be executed automatically when an object of the class is created.
- It is used to initialize the data members of a class.
- ***Syntax:***

Constructor inside class definition

```
class ClassName
{
…… // private members
public:
ClassName ( )   // Constructor prototype
{
// Constructor body definition
}
};
```

Constructor outside class definition

```
class ClassName
        {
        …… // private members
        public:
        ClassName ( );       // Constructor prototype
        };
ClassName : : ClassName ()
        {
         // Constructor body definition
        }
```

Constructor:

- ✓ The constructor has no return value specification, not even void.
- ✓ It is of course possible to define a class which has no constructor at all; in such case, the run-time system calls a dummy constructor (i.e. which performs no action) when its object is created.
- ✓ It can access any data member like all other member functions but cannot be invoked explicitly and must have public status to serve its purpose.
- ✓ The constructor which does not take arguments explicitly is called *default constructor.*
- ✓ Since a default constructor takes no argument, it follows that each class can have only one default constructor.
- ✓ The operation of default constructor is usually to initialize data used subsequently by other member functions. It can also be used to allocate the necessary resources such as memory, dynamically.

Parameterized Constructor:

- ✓ Constructor can be invoked with arguments, just as in the case of functions. Thus, constructors with arguments are called parameterized constructors.
- ✓ The argument list can be specified within braces similar to the argument-list in the function.
- ✓ ***Syntax:***

```
class Test
        {
        …..
        public:
        Test (int data)
            {……}
        };
Test t1(2);      // 2 is passed as parameter
Test t2= 3;      // 3 is passed as parameter
```

Constructor with parameter

Example:

```
Class Box
  {
   int height;
   public:
   Box ( int h)
     { height= h; }
  void show()
     {
      cout<<“Height= “<< height;
     }
  };
int main()
  {
   Box b= 4; OR Box b(4) ;
   b.show();          // Height= 4
  }
```

Constructor:

- ✓ If any constructor with any number of parameters is declared, no default constructor will exist, unless you define it.

```
class Rectangle
{
    private:
       int width;
       int length;
    public:
       Rectangle(int w, int l)
         {width =w; length=l;}
       void set(int w, int l);
       int area();
}
```

```
Rectangle r4;     // error
```

- Initialize with constructor

```
Rectangle r5(60,80);              // implicit call
Rectangle r6 = Rectangle(60,80);        // explicit call
```

Constructor Overloading:

✓ An interesting feature of the constructors is that a class can have multiple constructors. This is called as *constructor overloading*.

✓ Since C++ allows function overloading, a constructor with arguments can co-exist with another constructor with/without arguments.

✓ Here all the constructors have the same name as the corresponding class, and they differ only in terms of their signature (in terms of the number of arguments, or data types of their arguments, or both).

✓ In case of a class having multiple constructors, a constructor is invoked during the creation of an object depending on the number and type of arguments passed.

✓ ***E.g. of Constructor Overloading***

```
class Box
{
int height; float length;
public:
Box()              // Default constructor
{ cout<<“Enter height & length”; cin>>height>>length;}
Box (int h)        // Parameterized constructor
{ height= h; length= 8.5;}
Box ( int h, float l)  // Parameterized constructor
{ height= h; length = l;}
void show()
{
cout<<“Height= “ << height << “\tLength=“ << length ;}
};
```

```
int main()
{
Box b1;
Box b2= 4;
Box b3 ( 5, 6.5);
b1.show();
b2.show();
b3.show();
}
```

Output:

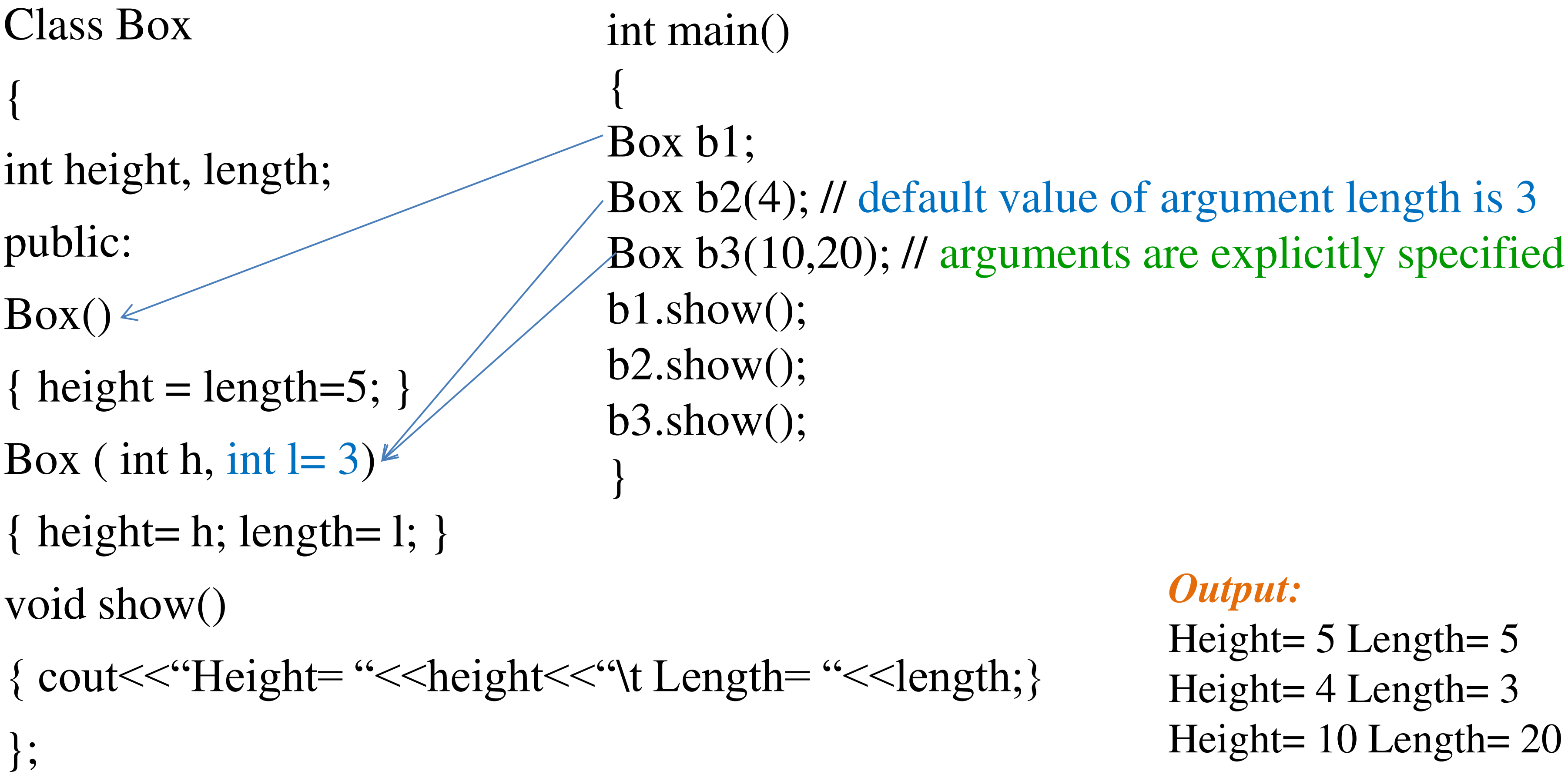
```
       Enter height & length
       2
b1     3
       Height= 2          Length= 3.0

b2     Height= 4          Length= 8.5

b3     Height= 5          Length= 6.5
```

Constructor with Default Arguments:

- ✓ Like any other functions in C++, constructors can also be defined with default arguments.
- ✓ If any arguments are passed during the creation of an object, the compiler selects the suitable constructor with default arguments.



```
Class Box
{
int height, length;
public:
Box()
{ height = length=5; }
Box ( int h, int l= 3)
{ height= h; length= l; }
void show()
{ cout<<“Height= “<<height<<“\t Length= “<<length;}
};
```

```
int main()
{
Box b1;
Box b2(4); // default value of argument length is 3
Box b3(10,20); // arguments are explicitly specified
b1.show();
b2.show();
b3.show();
}
```

Output:

```
Height= 5 Length= 5
Height= 4 Length= 3
Height= 10 Length= 20
```

Destructor:

- ✓ When an object is no longer needed it can be destroyed.
- ✓ A class can have another special member function called the *destructor,* which is invoked when an object is destroyed.
- ✓ This function complements the operation performed by any of the constructors, i.e. it is invoked when an object ceases to exist.
- ✓ Syntax: class ClassName

```
                  {
                  …….
                  public:
                    ~ ClassName ( );          // Destructor prototype
                  };
             ClassName : : ~ ClassName( )   // Destructor definition
                  {
                            // destructor body definition
                  }
```

Destructor:

✓ The destructor function has the same name as the class but prefixed by a tilde (~). The tilde distinguishes it from a constructor of the same class.

✓ Similar to constructors, a destructor must be declared in the public section of a class so that it is accessible to all its users.

✓ Destructors have no return type, not even void. Since it is invoked automatically whenever an object goes out of scope.

✓ A class cannot have more than one destructor.

✓ Unlike the constructors, the destructor does not take any arguments. This is because there is only one way to destroy an object.

Difference between Constructor & Destructor:

✓ Arguments cannot be passed to destructors.

✓ Only one destructor can be declared for a given class.

✓ As a consequence, destructors cannot have arguments and hence, destructors cannot be overloaded.

✓ Destructors can be virtual, while constructors cannot be virtual.

Dynamic Initialization through Constructor:

- ✓ Objects data members can be dynamically initialized during runtime, even after their creation.
- ✓ The advantage is that it supports different initialization formats using overloaded constructors.
- ✓ It provides flexibility of using different forms of data at runtime depending upon user’s need.

Dynamic Initialization through Constructor:

```
class name
{
char first[20], last[20];
public:
name(){first[0]= last[0]= ‘\0’;}
name (char *fname);
name (char *fname, char *lname);
void show( char *msg);
};
```

```
name:: name(char *fname)
{
strcpy(first, fname);
last[0]=‘\0’;
}
name:: name(char *fname, char *lname)
{
strcpy(first, fname);
strcpy(last, lname);
}
void name::show(char *msg)
        {
        cout<<msg<<endl;
        cout<<“First name=“<<first;
        cout<<“\nLast name=“<<last;
        }
```

```
int main()
{ name n1;
char nm[20], srn[20];
cout<<“Enter name”; cin>> nm;
n1= name(nm);
cout<<“enter name & sirname”; cin>>nm>>srn;
name n2=name(nm,srn);
n1.show(“First person..”);
n2.show(“Second person..”);}
```

Order of Construction & Destruction:

- ✓ The possibility of defining constructors with arguments, offer an opportunity to monitor the exact moment at which an object is created or destroyed during the execution of a program.

```
class Test
{
char name[20];
public:
Test();
Test(char *msg);
~Test();
};
```

```
Test :: Test()
{
strcpy(name,”unnamed”);
cout<<“test object ‘unnamed’ created”<<endl;
}
Test :: Test(char *msg)
{
strcpy(name, msg);
cout<<“test object <<name<< created”<<endl;
}
Test :: ~Test()
{
Cout<<“test object” <<name<< “destroyed”<<endl;
}
```

Order of Construction & Destruction:

```
Test g1(“global1”);
Test g2(“global2”);
void func()
{
Test l(“func1”);
cout<<“function”<<endl;
}
int main()
{
Test x(“main1”);
Func();
Cout<<“main() function termination”<<endl;
}
```

```
Output:
Test object global1 created
Test object global2 created
Test object main1 created
Test object func1 created
Function
Test object func1 destroyed
main() function termination
Test object main1 destroyed
Test object global2 destroyed
Test object global1 destroyed
```

Copy Constructor:

- ✓ The parameter of a constructor cannot be an object of its own class as a value parameter. i.e.

```
class X
{
        …..
        public:
        X ( X obj);            // Error: obj is value parameter
};
```

- ✓ However, a class’s own object can be passed as a reference parameter.

```
class X
{
        …..
        public:            Reference to an object of the class X
        X ( );
        X (X &obj);            // Copy Constructor
};
```

- ✓ Such a constructor having a reference to an instance of its own class as an argument is known as ***copy constructor***.

Copy Constructor:

- ✓ Copy constructor is used when the compiler has to create a temporary object of a class object.
- ✓ The copy constructor is used to initialise an object by another object of the same class.
- ✓ The copy constructor can be invoked as:

  X obj1(obj2);   OR   X obj1 = obj2;

  the above statement define the object obj1 and at the same time initialize it to the values of obj2.
- ✓ However, the statement obj1 = obj2; will not invoke the copy constructor. It will simply assigns the values of obj2 to obj1, member-by-member. This is the task of overloaded assignment operator (=).

Copy Constructor:

```
class Code
{
int id;
public:
  Code() { }
  Code (int a) { id=a;}
  Code (Code &x)

    {
        id = x.id;
     }
  void display()
    {
     cout<<id;
    }
};
```

```
int main()
{
   Code A(100);   // object A is created & initialized
   Code B (A);    // copy constructor called
   Code C = A;    //copy constructor called again
   Code D;        // D is created, not initialized
   D = A;         //copy constructor not called
   cout<<"\n id of A: "<< A.display( );
   cout<<"\n id of B: "<< B.display( );
   cout<<"\n id of C: "<< C.display( );
   cout<<"\n id of D: "<< D.display( );
   return 0;
}
```

Output:

```
id of A: 100
id of B: 100
id of C: 100
id of D: 100
```

Dynamic Constructor:

- ✓ The constructors can also be used to allocate memory while creating objects. This will enable the system to allocate the right amount of memory for each object when the objects are not of the same size, thus resulting in the saving of memory.
- ✓ Allocation of memory to objects at the time of their construction is known as ***dynamic construction of objects***.
- ✓ The memory is allocated with the help of the new operator.

Dynamic Constructor:

```
class Code
{
char *name;
int length;
public:
 Code()
   {
     length= 0;
     name = new char[length +1];
    }
  Code (char *s)
   {
     length = strlen(s);
     name = new char[length +1];
     strcpy(name, s)
    }
  void display()
   { cout<< name <<"\n";  }
  void join (Code &a, Code &b);
};
```

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

int main()
{
   char *first = "Ravi";
   Code A(first), B("Kumar"), C("Singh"), D,E;
   D.join (A,B);
   E.join (D,C);
   A.display();
   B.display();
   C.display();
   D.display();
   E.display();
   return 0;
}
```

Output:

```
Ravi
Kumar
Singh
Ravi Kumar
Ravi Kumar Singh
```

Returning Objects:

- ✓ A function cannot only receive objects as arguments but also can return them.
- ✓ The example illustrates how an object can be created (within a function) and returned to another function.
- ✓ Syntax:

Return type of the function

```
classname function_name(arguments)
        {
        }
```

Returning Objects:

```
class sample
{
int m[3][3];
int i,j;
public: void getdata();
void display();
friend sample trans(sample);   };
```

```
void sample :: getdata ()
{cout<<“enter the elements: ”;
for(i=0;i<3;i++)
for(j=0;j<3;j++)
cin>>m[i][j];}
void sample :: display ()
{
for(i=0;i<3;i++)
for(j=0;j<3;j++)
 cout<<m[i][j];}
```

```
sample trans(sample s)
{
sample ss;
int i, j;
for(i=0;i<3;i++)
for(j=0;j<3;j++)
ss.m[i][j]= s.m[j][i];
return (ss);
}
```

```
int main()
{
sample s1,s2;
s1.getdata();
s1.display();
s2=trans(s1);
s2.display();
return 0;
}
```

OPERATOR OVERLOADING

Ms. Sharmistha Roy
Assistant Professor,
School of Computer Engineering,
KIIT University

Introduction :

- ✓ The operator overloading feature of C++ is one of the methods of realizing polymorphism.
- ✓ The word *Polymorphism* is derived from the Greek words ‘poly’ means many or multiple and ‘morphism’ refers to actions, i.e. *performing many actions with single operator.*
- ✓ For example, let us consider the ‘+’ symbol. It can be used for performing addition if the operands are of integer type. Whereas, if the operands are of character type this ‘+’ symbol is used for concatenation of two words. Thus here, ‘+’ operator performs two different task in two different situation, which satisfies ***operator overloading*** concept.
- ✓ Operator overloading concepts are applied to the following two principle areas:
  - ➢ Extending capability of operators to operate on user defined data.
  - ➢ Data conversion
- ✓ Operator overloading extends the semantics of an operator without changing its syntax. Thus the grammatical rules such as no. of operands, precedence & associativity of the operator remain the same for overloaded operators.

Overloadable Operators:

- ✓ C++ provides a wide variety of operators to perform operations on various operands. The operators are classified into unary and binary operators based on the number of arguments on which they operate.
- ✓ C++ allows almost all operators to be overloaded in which case atleast one operand must be an instance of a class (object).
- ✓ The precedence relation of overloadable operators and their expression syntax remains the same even after overloading.
- ✓ The following table shows the list of operators which can be overloaded:

Operator Category	Operators
Arithmetic	+, -, *, /, %
Bit-wise	&, |, ~, ^
Logical	&&, ||, !
Relational	>, <, ==, !=, <=, >=
Arithmetic Assignment	+=, -=, *=, /=, %=, &=, |=, ^=
Unary	++, --
Shift	<<, >>, <<=, >>=

Defining Operator Overloading :

- To define an additional task to an operator, we must specify what it means in relation to the class to which the operator is applied.
- This is done with the help of a special function, called **operator function**, which describes the task. The general form of an operator function outside class is:

```
return type Classname :: operator op(arglist)
        {
                    function body
        }
```

- Here *return type* is the type of value returned by the specified operation and *op* is the operator being overloaded. The *op* is preceded by the keyword ***operator***. ***Operator*** *op* is the function name.
- Operator functions must be either member functions (non-static) or friend functions.
- The basic difference is that a friend function will have only one argument for unary operators and two for binary operators, while a member function has no argument for unary operators and only one for binary operators. This is because the object used to invoke the member function is passed implicitly and therefore available for member function. But this is not the case with friend functions. Arguments may be passed either by value or by reference.

Defining Operator Overloading :

Types of Operator	Member Function	Friend Function
Unary Operator	operator op()	operator op(x)
Binary Operator	x.operator op(y)	operator op(x,y)

- ✓ *The process of operator overloading involves the following steps:*
- ✓ Declare a class (that defines the data type) whose objects are to be manipulated using operators.
- ✓ Declare the operator function, in the public part of the class. It can be either a normal function or a friend function.
- ✓ Define the operator function either within the body of a class or outside the body of the class.

Overloading Unary operators:

- Unary operators overloaded by member functions take no formal arguments, whereas when they are overloaded by friend functions they take a single argument.
- Let us consider the unary minus operator. A minus operator when used as a unary, takes just one operand. We know that this operator changes the sign of an operand when applied to a basic data item.
- ***Syntax:***

```
ReturnType operator OperatorSymbol ()
{
        // body of Operator function
}
```

Overloading Unary operators:

✓ *Example of Overloading of Unary Minus Operator using member function*

```
class space
{
  int x;
  int y;
public:
  void getdata (int a, int b);
  void display ();
  void operator –();        // overload unary minus
};
void space :: getdata (int a, int b)
{
  x= a;
  y= b;
}
void space :: display()
{
  cout << “x=“ <<x;
  cout << “\ty=“ <<y;
}
```

```
void space :: operator –()
{
  x= -x;
  y= -y;
}

int main()
 {
  space s;
  s.getdata (10, -20);
  s.display();
  –s;
  s.display();
 }
```

Output:

```
x= 10    y= -20
x= -10   y= 20
```

Overloading Unary operators:

- ✓ *Example of Overloading of Unary Minus Operator using friend function*

```
class space
{
  int x;
  int y;
public:
  void getdata (int a, int b);
  void display ();
  friend void operator –(space &s);
};
void space :: getdata (int a, int b)
{
  x= a;
  y= b;
}
void space :: display()
{
  cout << “x=“ <<x;
  cout << “\ty=“ <<y;
}
```

```
void operator –(space &s)
{
  s.x= -s.x;
  s.y= -s.y;
}

int main()
 {
  space s;
  s.getdata (10, -20);
  s.display();
  –s;
  s.display();
 }
```

Output:

```
x= 10    y= -20
x= -10   y= 20
```

Operator Return Values:

- ✓ The operator function can be used to return values, the return type of the function is specified depending upon the value it returns. This program is for **pre-increment operator overloading**

```
class Index
{
 int value;
public:
 Index() { value= 0; }
 int getdata ()
 { return value; }
 Index operator ++()
 {
  Index temp;
  temp.value= ++value;
  return temp;
 }
};
```

```
int main()
{
Index id1, id2;
cout<<“\nIndex1=“<<id1.getdata();
cout<<“\nIndex2=“<<id2.getdata();
id1=++id2;
cout<<“\nIndex1=“<<id1.getdata();
cout<<“\nIndex2=“<<id2.getdata();
return 0;
}
```

Output:

```
Index1= 0
Index2= 0
Index1= 1
Index2= 1
```

Operator Return Values:

- ✓ This program is for **post-increment operator overloading**. Here, the ***int*** inside bracket of *operator* function indicates the compiler that it is post-fix version of the operator.

```
class Index
{
 int value;
public:
 Index() { value= 0; }
 int getdata ()
 { return value; }
 Index operator ++(int)
 {
  Index temp;
  temp.value= value++;
  return temp;
 }
};
```

```
int main()
{
 Index id1, id2;
 cout<<"\nIndex1="<<id1.getdata();
 cout<<"\nIndex2="<<id2.getdata();
 id1=id2++;
 cout<<"\nIndex1="<<id1.getdata();
 cout<<"\nIndex2="<<id2.getdata();
 return 0;
}
```

Output:

```
Index1= 0
Index2= 0
Index1= 0
Index2= 1
```

Nameless Temporary Objects:

- ✓ In the previous program, an intermediate (a temporary) object temp is created as a return object.
- ✓ A convenient way to return an object is to create a *nameless object* in the return statement itself.

```
class Index
{
 int value;
public:
 Index() { value= 0; }
 Index (int val)
 {
  value= val;
 }
 int getdata ()
 { return value; }
 Index operator ++()   // returns nameless object
 {
  value= value+1;
  return Index(value);
 }};
```

```
int main()
{
 Index id1, id2;
 cout<<“\nIndex1=“<<id1.getdata();
 cout<<“\nIndex2=“<<id2.getdata();
 id1=id2++;
 cout<<“\nIndex1=“<<id1.getdata();
 cout<<“\nIndex2=“<<id2.getdata();
 return 0;
}
```

Output:

```
Index1= 0
Index2= 0
Index1= 1
Index2= 1
```

Overloading Binary operators:

- ✓ The syntax for overloading binary operator is shown below:

```
ReturnType operator OperatorSymbol (arg)
{
        // body of Operator function
}
```

- ✓ The binary overloaded operator function takes the first object as an implicit operand and the second operand must be passed explicitly.
- ✓ The data members of the first object are accessed ***without*** using the dot operator whereas, the second argument members can be accessed ***using*** the dot operator if the argument is an object, otherwise it can be accessed directly.

Overloading Binary operators:

- ✓ *Example of Overloading of + Operator using member function*

```
class complex
{
 float x, y;
public:
 complex() {  }
 complex(float real, float imag)
  {
   x= real;  y= imag;
  }
 complex operator + (complex);
 void display();
};
void complex :: display()
{
 cout<<x<<“ + i”<<y<<“\n”;
}
```

```
complex complex :: operator + (complex c)
{
   complex temp;
   temp.x = x + c.x;
   temp.y = y + c.y;
   return temp;
}

int main()
 {
  complex c1, c2, c3;
  c1= complex(2.5, 3.5);
  c2= complex(1.6, 2.7);
  c3= c1 + c2;
  cout<<“C3= “<<c3.display();
 }
```

Output:

```
C3= 4.1 + i6.2
```

Overloading Binary operators:

```
complex complex :: operator + (complex c)
{
complex temp;
temp.x = x + c.x;
temp.y = y + c.y;
return temp;
}
```

✓ *Features of this function are described below:*

✓ It receives only one complex type argument explicitly.

✓ It returns a complex type value.

✓ It is a member function of complex.

Overloading Binary operators:

- ✓ The statement c3 = c1 + c2 is equivalent to:

  c3= c1.operator +(c2)

- ✓ Thus, in the operator +() function, the data members of c1 are accessed directly and the data members of c2 (that is passed as an argument) are accessed using the dot operator. Thus, both the objects are available for the function.
- ✓ For example, the statement **temp.x = x + c.x;** in the function body refers to:
- ✓ **c.x** refers to the object **c2** and **x** refers to the object **c1**. **temp.x** is the real part of **temp** that is used to hold the results of addition of **c1** & **c2**. The function returns the complex **temp** to be assigned to **c3**.
- ✓ We can avoid the creation of the temp object by replacing the entire function body by the following statement:
  - ✓ return complex ((x+c.x), (y+c.y));
- ✓ As a rule, in overloading of binary operators, the *left-hand* operand is used to invoke the operator function and the *right-hand* operand is passed as an argument.

Overloading Binary operators using Friends:

- ✓ *Example of Overloading of + Operator using friend function*

```
class complex
{
 float x, y;
public:
 complex() {  }
 complex(float real, float imag)
 {
  x= real;  y= imag;
 }
 friend complex operator + (complex, complex);
 void display();
};
void complex :: display()
{
 cout<<x<<" + i"<<y<<"\n";
}
```

```
complex operator + (complex a, complex b)
{
   complex temp;
   temp.x = a.x + b.x;
   temp.y = a.y + b.y;
   return temp;
}
```

```
int main()
 {
  complex c1, c2, c3;
  c1= complex(2.5, 3.5);
  c2= complex(1.6, 2.7);
  c3= operator + (c1,c2);
  cout<<"C3= "<<c3.display();
 }
```

Output:

```
C3= 4.1 + i6.2
```

Overloading Binary operators:

- ✓ Now a question arise that we can get the same results by the use of either a friend function or a member function. Why then an alternative is made available?
- ✓ This is because there are certain situations where we would like to use a friend function rather than a member function.
- ✓ For instance, consider a situation where we need to use two different types of operands for a binary operator, say one an object and another a built-in type data as shown below:
- ✓ A= B+2; (or A= B*2;). This will work for a member function but the statement A= 2+B; (or A= 2*B;) will not work.
- ✓ This is because the left hand operand which is responsible for invoking the member function should be an object of the same class.
- ✓ However, friend function allows both approaches. This is because an object need not be used to invoke a friend function but can be passed as an argument. Thus we can use a friend function with a built-in type data as the left-hand operand and an object as the right-hand operand. This is shown with the help of an example in next slide.

Overloading Binary operators using Friends:

- ✓ *Special Example of Overloading of + Operator using friend function*

```
class complex
{
 int a;
public:
 complex() {  }
 complex(int x)
  {
   a=x;
  }
 friend complex operator + (int, complex);
 void display();
};
void complex :: display()
{
 cout<<“a=“<<a<<“\n”;
}
```

```
complex operator + (int x, complex y)
{
   complex temp;
   temp.a= x+ y.a;
   return temp;
}
```

```
int main()
 {
  complex c1, c2;
  c1= complex(2);
  c1.display();
  c2= operator + (5,c1);
  c2.display();
 }
```

Output:

```
a= 2
a= 7
```

Rules for Overloading operators:

1. Only existing operators can be overloaded. New operators cannot be created.
2. The overloaded operator must have at least one operand that is of user-defined type.
3. We cannot change the basic meaning of an operator. That is to say, we cannot redefine the plus (+) operator to subtract one value from the other.
4. Overloaded operators follow the syntax rules of the original operators. They cannot be overridden.
5. There are some operators that cannot be overloaded. They are listed below:

Symbol	Meaning
.	Membership operator
.*	Pointer-to-member operator
::	Scope resolution operator
?:	Conditional operator
sizeof()	Size operator

Rules for Overloading operators:

6. We cannot use friend functions to overload certain operators. They are listed below. However, member functions can be used to overload them.

=	Assignment operator
( )	Function call operator
[ ]	Subscripting operator
->	Class member access operator

7. Unary operators, overloaded by means of a member function, take no explicit arguments and return no explicit values, but, those overloaded by means of a friend function, take one reference argument (the object of the relevant class).
8. Binary operators, overloaded by means of a member function, take one explicit argument and those overloaded by means of a friend function, take two explicit arguments.
9. When using binary operators overloaded through a member function, the left hand operand must be an object of the relevant class.
10. Binary arithmetic operators such as +, -, *, and / must explicitly return a value. They must not attempt to change their own arguments.

Type Conversion:

- ✓ C supports automatic type conversion for built-in data types only.
- ✓ For e.g.

```
int m;
float q= (float) m;
```

- ✓ The same thng is done in C++ as:

```
int m;
float q= float (m);
```

- ✓ Similarly, an assignment operation also causes the automatic type conversion. The type of data to the right of an assignment operator is automatically converted to the type of the variable on the left. For e.g.

```
int m;
float q= 3.14159;
m= q;
```

- ✓ These are all for built-in data types. Now what happens for user-defined data type. Consider the statement v3= v1 + v2; where v1, v2 & v3 are the objects of the same class, then the operation of addition and assignment are carried out smoothly and the compiler does not make any complaints.
- ✓ Now the problem arise if one of the operand is an object & the other is a built-in type variable or if the objects belong to two different classes.

Type Conversion:

- ✓ Since, the user-defined data types are designed by us to suit our requirements, the compiler does not support automatic type conversions for such data types. For these, there are some conversion routines:
- ✓ Conversion from basic type to class type.
- ✓ Conversion from class type to basic type.
- ✓ Conversion from one class type to another class type.

Basic to Class Type:

- ✓ Conversion takes place with the help of **Constructor**.
- ✓ It performs a defacto type conversion from the argument's type to the constructors's class type.
- ✓ Here we used a constructor to build a string type object from a char* type variable.

```
string :: string (char *a)
{
  length = strlen(a);
  P = new char[length + 1];
  strcpy(P, a);
}
```

- ✓ This constructor builds a string type object from a char* type variable a. The variables length and P are data members of the class string. Once this constructor has been defined in the string class, it can be used for conversion from char* type to string type.

Basic to Class Type:

```
string s1, s2;
char *name1 = “KIIT”;
char *name2 = “Computer Science”;
s1 = string (name1);
s2 = name2;
```

- ✓ The statement s1 = string(name1); first converts name1 from char* type to string type and then assigns the string type values to the object s1.
- ✓ The statement s2 = name2; also does the same job by invoking the constructor implicitly.
- ✓ Note: the constructor used for the type conversion take a single argument whose type is to be converted.

Basic to Class Type:

```
class convert
{
      private:
              int x;
       public:
              convert()
                  {
                      x= 0;
                   }
              convert(int a)
                  {
                      x=a;
                  }
              void show()
                    {
                       cout<<“x=“<<x;
                     }
               };
```

```
void main()
 {
    int a;
    convert  ob;
     cout<<“enter the value of a”;
     cin>>a;
     ob=a;             // Int to Class
     ob.show();
}
```

Here for the statement **ob=a;** the compiler searches for overloaded assignment operator but since the above is not found it invokes the one-argument constructor.

Class to Basic Type:

- ✓ The constructor did a fine job in type conversion from a basic to class type.
- ✓ Now for the conversion from a class to basic type the constructor functions do not support this operation.
- ✓ Luckily, C++ allows us to define an overloaded casting operator that could be used to convert a class type data to a basic type.
- ✓ The general form of an ***Overloaded Casting Operator Function***, usually referred to as a conversion function is:

```
operator typename ( )
{
  ……… (Function statements)
}
```

- ✓ This function converts a class type data to typename. For example, the operator double ( ) converts a class object to type double.
- ✓ The operator function takes no argument but returns a basic data item.

Class to Basic Type:

- When the compiler encounters a statement that requires the conversion of a class type to a basic type, it quietly calls the casting operator function to do the job.
- The casting operator function should satisfy the following conditions:
- **It must be a class member.**
- **It must not specify a return type.**
- **It must not have any arguments.**
- Since, it is a member function, it is invoked by the object & therefore, the values used for conversion inside the function belong to the object that invoked the function. This means that the function does not need an argument.

Class to Basic Type:

```
class convert
{
    private:
            int x;
     public:
            convert()
               {
                  x= 0;
                }
            void get(int s)
                 {
                   x=s;
                  }
            operator int()     //Conversion function
             {
               int a;
               a = x;
               return (a);
              }
 };
```

```
void main()
 {
   int a;
   convert  ob;
    ob.get(10);
    a=ob;              // Class to Int
    cout<<a;
}
```

One Class Type to Another Class Type:

- ✓ We have worked for *Basic to Class Type* with the help of *Constructor* and *Class to Basic Type* with the help of Conversion function i.e. *Operator function*.
- ✓ Now we need the conversion from one Class Type to another Class Type.

What we need to achieve?

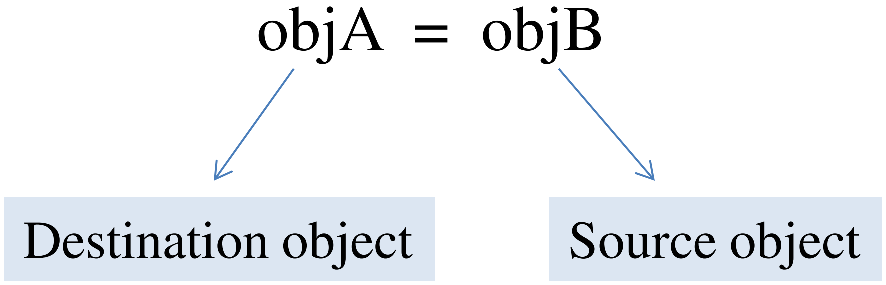
```
classA   objA;
classB   objB;

objA=objB;
```

Need to specify the Conversion Routine.

- ✓ The conversion routine can be specified in either ClassA or ClassB and depending upon which class either of the one way need to be choosen

One Class Type to Another Class Type:



objA = objB

Destination object

Source object

If conversion routine is specified in the source object class :
Operator function

If the conversion routine is specified in the destination object class:
Constructor

One Class Type to Another Class Type:

Conversion routine in Source Object: Operator function

Syntax:

```
class A                    //destination object class
{

}
class B                    //source object class
{
          operator A()     //Conversion Operator function
          {
                    .......
          }

}
```

One Class Type to Another Class Type:

Conversion routine in Source Object: Operator function

```
class A
{
       int a;
       public:
        A()
         {
           a= 0;
         }
       A(int x)
        {
         a = x;
        }
       void show()
       {
          cout<<a;
       }
 };
```

```
class B
{
      int b;
       public:
         B()
         {
           b= 10;
         }
        operator A( )
         {
          A obja(b);
          return (obja);
         }
 };
```

```
void main()
 {
    A a;
    B b;
    a=b;
    a.show();
 }
```

One Class Type to Another Class Type:

Conversion routine in Destination Object: Constructor

Syntax:

```
class B                         //source object class
{
            .......
}

class A                         //destination object class
{
            A( B objb)
            {
                        .......
            }

}
```

One Class Type to Another Class Type:

Conversion routine in Destination Object: Constructor

```
class A
{
      int a;
      public:
       A()
        {
          a= 0;
         }
       A(B objb)
        {
         a = objb.getB();
        }
      void show()
        {
         cout<<a;
        }
 };
```

```
class B
{
      int a1;
       public:
          B()
           {
             a1= 10;
            }
           int getB( )
             {
               return (a1);
              }
 };
```

```
void main()
 {
    A a;
    B b;
    a=b;
    a.show();
}
```

Type Conversion:

<table>
<tr><th colspan="3">Type Conversions</th></tr>
<tr><td></td><td>Routine In Destination</td><td>Routine In Source</td></tr>
<tr><td>Basic-to-Basic</td><td colspan="2">(Built In Conversion Functions)</td></tr>
<tr><td>Basic-to-Class</td><td>Constructor</td><td>N/A</td></tr>
<tr><td>Class-to-Basic</td><td>N/A</td><td>Conversion Function</td></tr>
<tr><td>Class-to-Class</td><td>Constructor</td><td>Conversion Function</td></tr>
</table>

INHERITANCE

Ms. Sharmistha Roy
Assistant Professor,
School of Computer Engineering,
KIIT University

Introduction:

- ✓ Inheritance is a technique of organizing information in a hierarchical form, like a child inheriting the features of its parents.
- ✓ Inheritance allows new classes to be built from older and less specialized classes instead of being rewritten from scratch. Thus, the technique of building new classes from the existing classes is called *inheritance.*

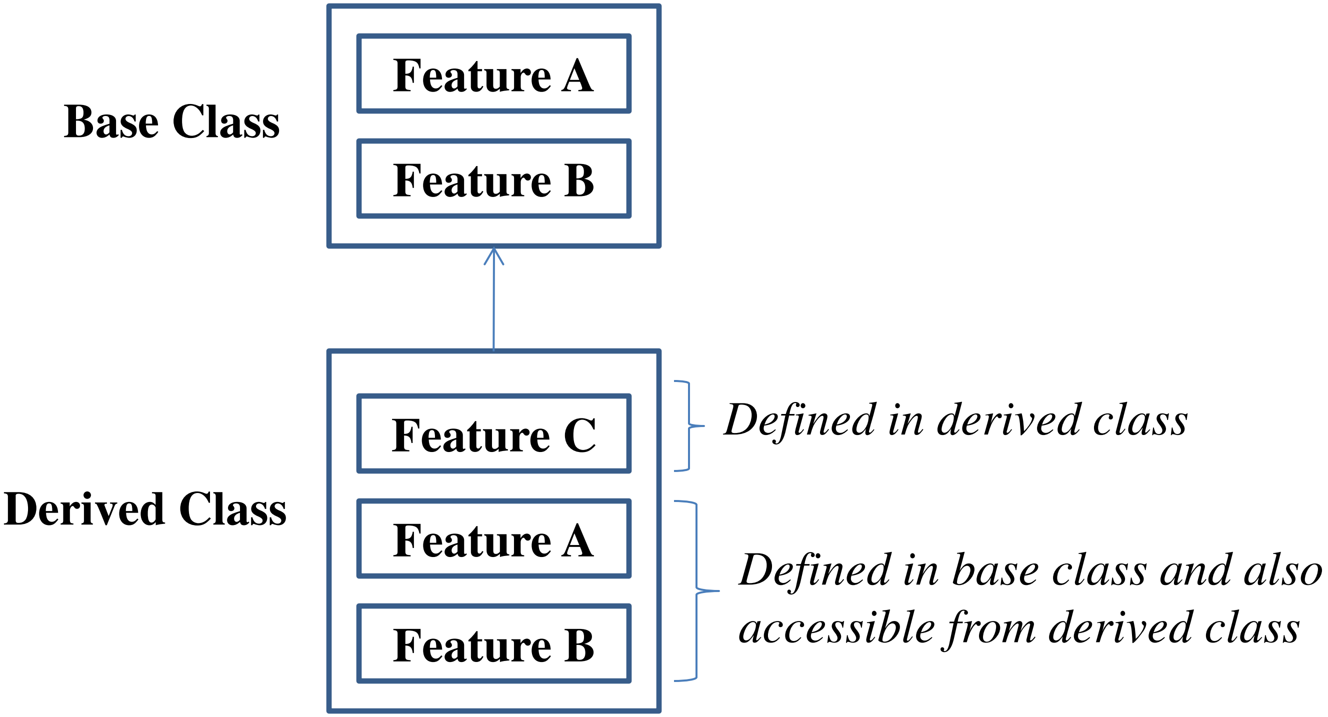


Fig: Base class and derived class relationship

Introduction:

- ✓ A base class is known as ancestor, parent or superclass and a derived class is called the descendent, child or subclass.
- ✓ In inheritance, the base class remains unchanged. Whereas, the derived class inherits all the features of base class and adds its own feature as well.
- ✓ The derived class can access all the features of the base class but the reverse do not occur.
- ✓ As far as the access limit is concerned, within a class or from the objects of a class protected access-limit is same as that of the private specifier.
- ✓ However, in inheritance, the protected data member can be accessed from the base class as well as the derived class.
- ✓ *Private data members*: visible to member functions within its class but not in derived class.
- ✓ *Protected data members*: visible to member functions within its class and derived class.
- ✓ *Public data members*: visible to member functions within its class, derived classes and through object.

Derived Class Declaration:

- ✓ A derived class extends its features by inheriting the properties of another class, called base class and adding features of its own.
- ✓ *Syntax:*

```
class DerivedClass : (VisibilityMode) BaseClass
        {
        // members of derived class
        // and they can access members of the base class
        };
```

the derivation of DerivedClass from the BaseClass is indicated by the colon (:). The VisibilityMode enclosed within the square brackets implies that it is optional. By default, visibility mode is private. Visibility mode specifies whether the features of the base class are publicly or privately inherited.

- ✓ No memory is allocated to the declaration of a derived class, but memory is allocated when it is instantiated to create objects.

Styles of Derivation:

✓ Class D : public B

```
        {
                    // members of D
        };
```

✓ Class D : private B

```
        {
                    // members of D
        };
```

✓ Class D : B   // private derivation by default

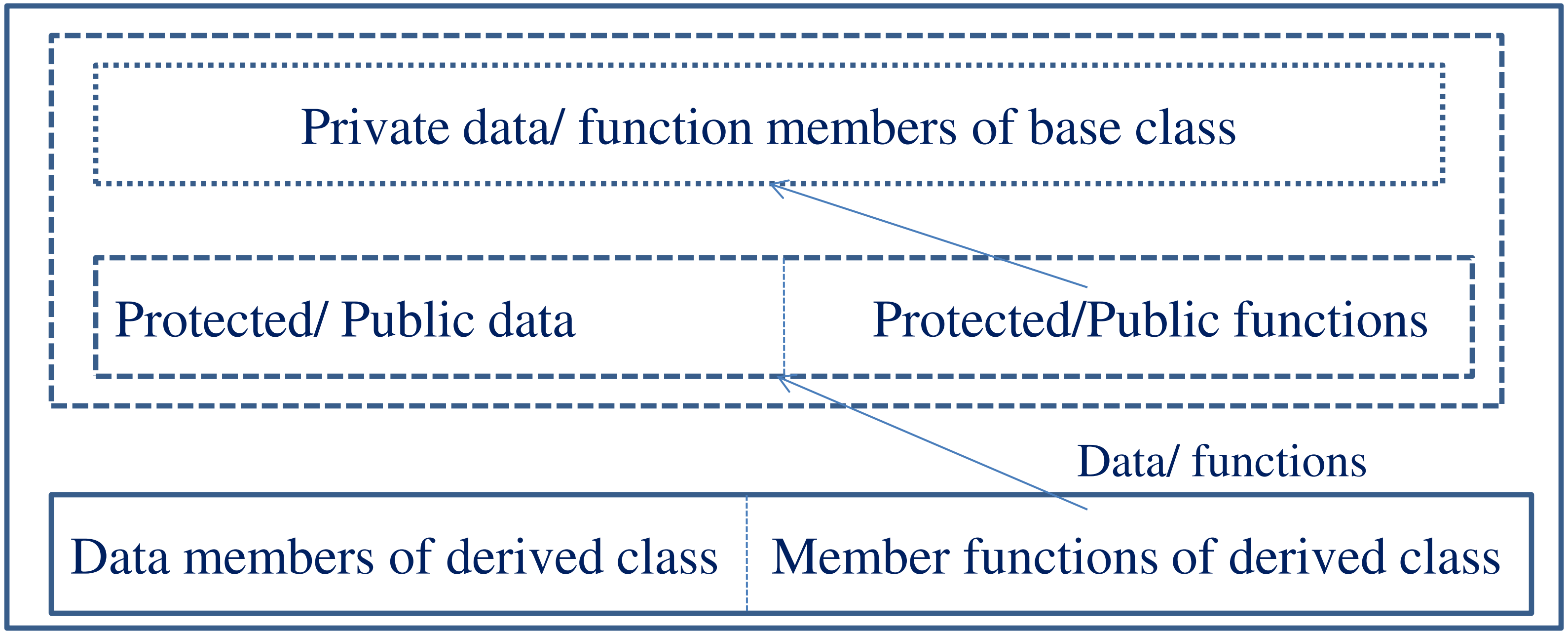
```
        {
                     // members of D
        };
```

✓ Class D: protected B

```
        {
                    // members of D
        };
```

Visibility of class members:

Base class visibility of data	**Derived class visibility of data**		
	Public derivation	Private derivation	Protected derivation
Private	Not Inherited	Not Inherited	Not Inherited
Protected	Protected	Private	Protected
Public	Public	Private	Protected



Private data/ function members of base class

Protected/ Public data

Protected/Public functions

Data/ functions

Data members of derived class

Member functions of derived class

Inherited members of a base class

Derived class's own members

Member Accessibility:

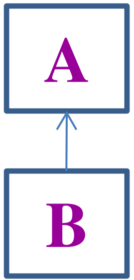
- ✓ A private member is accessible only to members of the class in which the private member is declared. They cannot be inherited.
- ✓ A private member of the base class can be accessed in the derived class through the member functions of the base class.
- ✓ A protected member is accessible to members of its own class and to any of the members in a derived class.
- ✓ If a class is expected to be used as a base class in future, then members which might be needed in the derived class should be declared protected rather than private.
- ✓ A public member is accessible to members of its own class, members of the derived class, and outside users of the class.
- ✓ The private, protected and public sections may appear as many times as needed in a class and in any order.
- ✓ The visibility mode in the derivation of a new class can be either private or public or protected.

Forms of Inheritance:

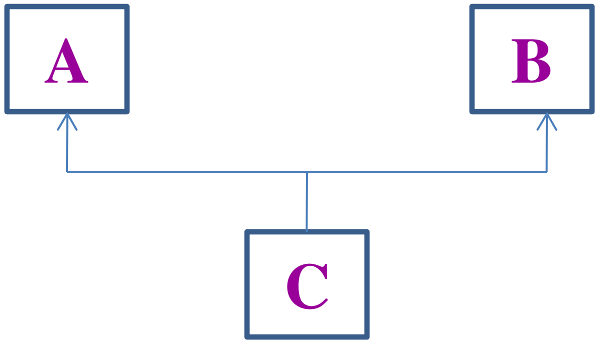
- ✓ The derived class inherits some or all the features of the base class depending on the visibility mode and level of inheritance.
- ✓ Level of inheritance refers to the length of its(derived class) path from the root(top base class).
- ✓ Types of Inheritance based on the levels of inheritance and interrelation among the classes:
- ✓ Single Inheritance
- ✓ Multiple Inheritance
- ✓ Hierarchical Inheritance
- ✓ Multilevel Inheritance
- ✓ Hybrid Inheritance
- ✓ Multipath Inheritance

Forms of Inheritance:

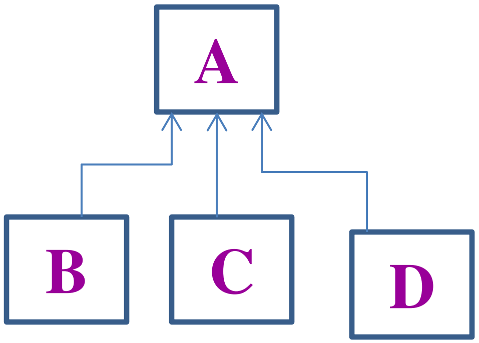
- ✓ ***Single Inheritance***: derivation of a class from only one base class.



- ✓ ***Multiple Inheritance***: derivation of a class from several (two or more) base classes.

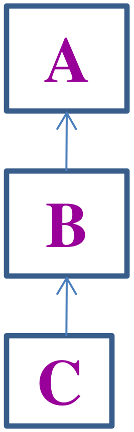


- ✓ ***Hierarchical Inheritance***: derivation of several classes from a single base class i.e. the traits of one class may be inherited by more than one class.

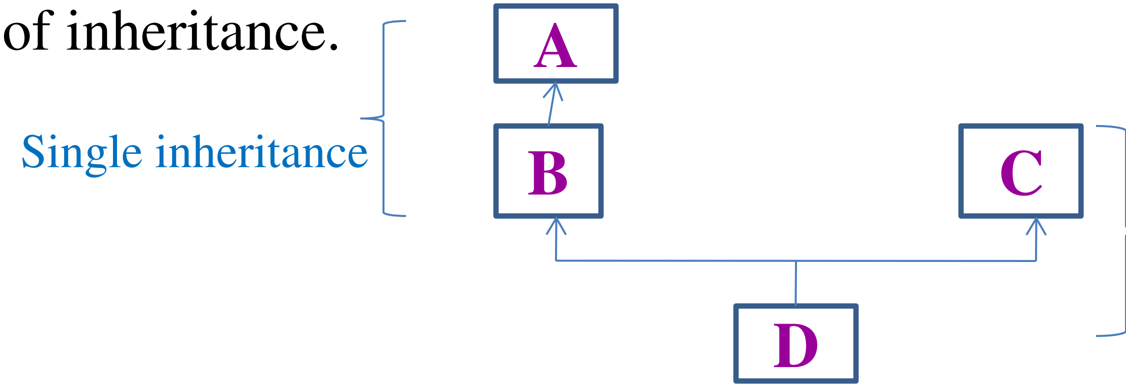


Forms of Inheritance:

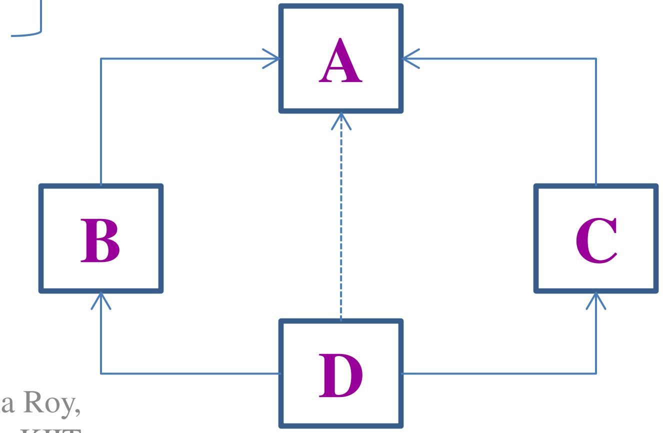
- ✓ ***Multilevel Inheritance***: derivation of a class from another derived class.



- ✓ ***Hybrid Inheritance***: derivation of a class involving more than one form of inheritance.



- ✓ ***Multipath Inheritance***: derivation of a class from other derived classes, which are derived from the same base class.



Example of Single Inheritance:

```
class X
{
protected:
int a,b;
public:
void input();
void display();
};
```

```
void X :: input()
{
cout<<“Enter the values of a & b:”;
cin>> a >> b;
}
void X :: display()
{
cout<<“Values of a & b: ”<<a<<b;
}
```

```
void Y :: swap()
{
t=a;
a=b;
b=t;
}
```

```
class Y : public X
{
int t;
public:
void swap();
};
```

```
int main()
{
Y y;
y.input();
cout<<“Before swapping”;
y.display();
y.swap();
cout<<“After swapping”;
y.display();
return 0;
}
```

Output:

```
Enter the values of a & b:
2
3
Before swapping
Values of a & b: 2 3
After swapping
Values of a & b: 3 2
```

Example of Single Inheritance:

```
class X
{
protected:
int a,b;
public:
void input();
void display();
};
```

```
class Y : private X
{
int t;
public:
void swap();
};
```

```
void X :: input()
{
cout<<"Enter the values of a & b:";
cin>> a >> b;
}
void X :: display()
{
cout<<"Values of a & b: "<<a<<b;
}
```

```
void Y :: swap()
{
input();
cout<<"Before swapping";
X :: display();
t=a;
a=b;
b=t;
cout<<"After swapping";
X :: display();
}
```

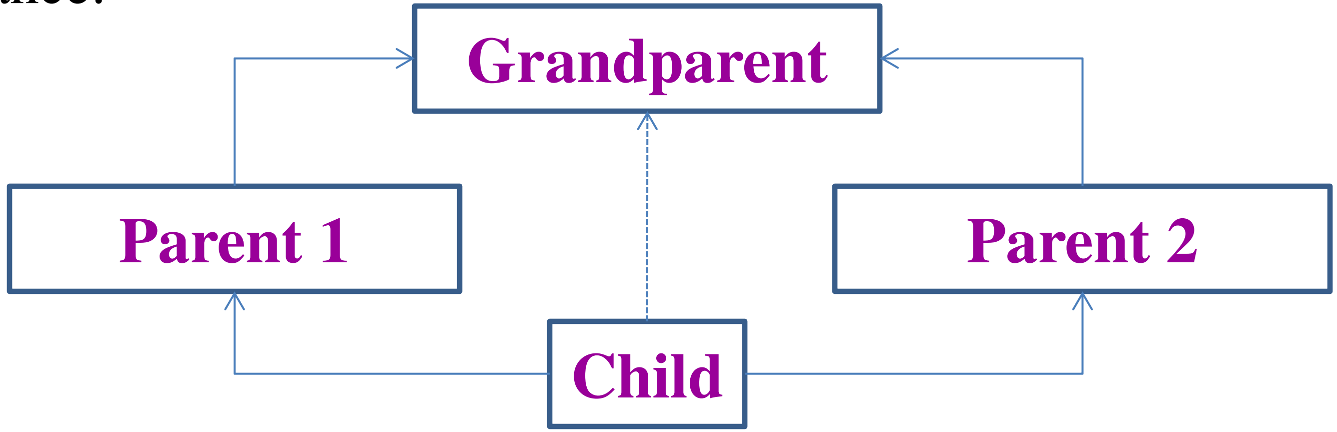
```
int main()
{
Y y;
y.input();  //error
y.swap();
return 0;
}
```

Output:

```
Enter the values of a & b:
2
3
Before swapping
Values of a & b: 2 3
After swapping
Values of a & b: 3 2
```

Virtual Base Classes:

- ✓ The multipath inheritance we have studied seems that there are three kinds of inheritance, namely, multilevel, multiple and hierarchical inheritance.



- ✓ Here the child has two *direct base classes* ‘parent 1’ and ‘parent 2’ which themselves have a common base class ‘grandparent’. The ‘child’ inherits the traits of ‘grandparent’ via two separate paths. It can also inherit directly as shown by the broken line. The grandparent is sometimes referred to as *indirect base class*.
- ✓ Here the child inherits properties of grandparent twice. This introduces ambiguity and should be avoided.

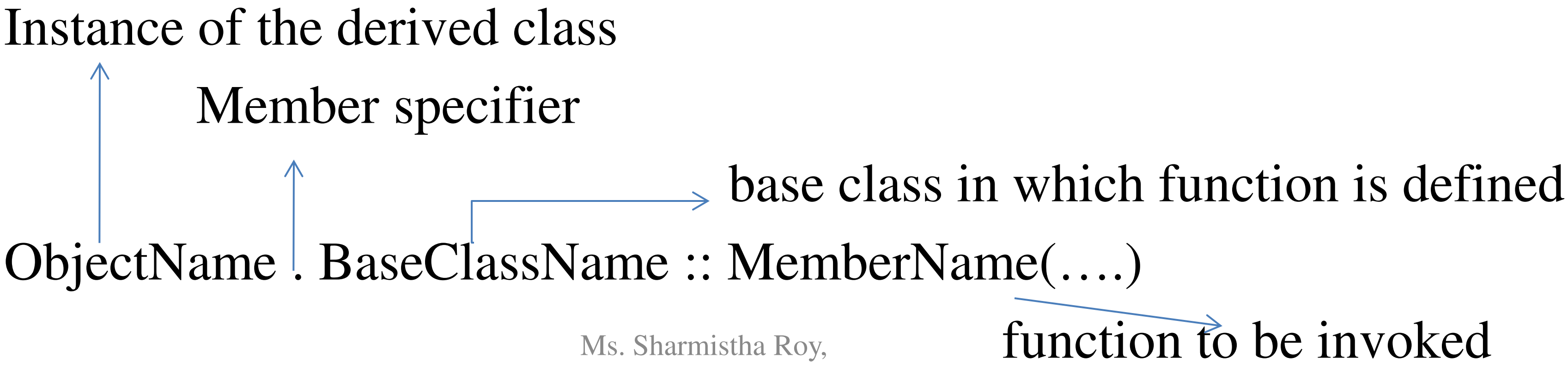
Virtual Base Classes:

✓ The duplication of inherited members due to these multiple paths can be avoided by making the common base class as *virtual base class* while declaring the direct or intermediate base classes as shown below:

```
class A                  // grandparent
{
……
};
class B1 : virtual public A// parent1
{
……
};
class B2 : public virtual A// parent2
{
……
};
Class C : public B1, public B2      // child
{
…..   // only one copy of A will be inherited
};
```

Ambiguity in Inheritance:

- ✓ Ambiguity is a problem that surfaces in certain situations involving single inheritance & multiple inheritance. Consider the following cases:
- ✓ In multiple inheritance, when more than one base classes having functions with the same name.
- ✓ The problem can be resolved by the use of classname and scope resolution operator which informs the compiler which function to call depending on the class name.
- ✓ In single inheritance, if the base class and derived class is having the function with the same name.
- ✓ This problem is resolved using the scope resolution operator as shown below:



Instance of the derived class

Member specifier

base class in which function is defined

ObjectName . BaseClassName :: MemberName(….)

function to be invoked

Overloaded Member Functions:

- ✓ The members of a derived class can have the same name as those defined in the base class.
- ✓ An object of a derived class refers to its own functions even if they are defined in both the base class and the derived class.
- ✓ Objects of the base class do not know anything about the derived class and will always use the base class members.

Overloaded Member Functions:

```
class B
{
protected:
  int x,y;
public:
  B() { }
  void read()
  {
  cout<<“Enter X & Y in base class”;
  cin>>x>>y;
 }
  void show()
  {
        cout<<“X=“<<x<<“\tY=“<<y;
  }
};
```

```
class D : public B
{
 protected:
   int  y,z;
 public:
 void read()
  {
   B :: read();
   cout<<“Enter Y& Z in derived class”;
   cin>>y>>z;
  }
void show()
 {
  B :: show();
 cout<<“Y=“<<y<<“\tZ=“<<z;
  cout<<“Y of B from D”<<B::y;
 }
};
```

```
int main()
  {
  D ob;
  ob.read();
  ob.show();
  }
```

Scope Resolution with Overriding Functions:

- ✓ In the above program, the statement in class D ***B :: read();*** refers to the function read() defined in the base class B due to the use of scope resolution operation.
- ✓ Similarly, the statement ***B :: y;*** refers to the data member defined in the base class B and not the one defined in the derived class.
- ✓ The general format of scope resolution for class members is:

    ClassName :: MemberName()

- ✓ Thus prefixing the class name to the member separated by scope resolution operator :: informs the compiler to call the member function specified in that class.

Abstract Classes:

✓ An abstract class is one that is not used to create objects.

✓ An abstract class is designed only to act as a base class (to be inherited by other classes).

✓ It is a design concept in program development and provides a base upon which other classes may be built.

✓ However, we can redefine the definition of abstract class as a class which contains at least one pure virtual function.

Constructors in Derived Classes:

- The constructors role is to initialize an objects data members and allocating required resources such as memory.
- The derived class need not have a constructor as long as the base class has a no-argument constructor.
- However, if the base class has constructors with arguments, then it is mandatory for the derived class to have a constructor and pass the arguments to the base class constructor.
- In the application of inheritance, objects of derived class are usually created instead of base class.
- Hence, it needs to have a constructor of the derived class and pass arguments to the constructor of the base class.
- When an object of a derived class is created, the constructor of the base class is executed first and later the constructor of the derived class.

Constructors in Derived Classes:

✓ *No-constructors in the base class and derived class:*

```
class X          //base class
{
// body of the base class, without constructors
};
class Y : public X        //publicly derived class
{
 public:
 void msg()
 {
 cout<<"No constructors exists in base & derived class";
 }
};

int main()
{
 Y y;    // dummy constructor
 y.msg();
}
```

Output:

```
No constructors exists in base & derived class
```

Constructors in Derived Classes:

✓ *Constructor only in the base class:*

```
class X                //base class
{
 public:
  X()
  {
   cout<<“No argument constructor of base class X”;
  }
};
class Y : public X             //publicly derived class
{
 public:
};

int main()
{
 Y y;      // accesses base constructor
}
```

Output:

```
No argument constructor of base class X
```

Constructors in Derived Classes:

- ✓ *Constructor only in the derived class:*

```
class X                  //base class
{
 //  body of base class, without constructors
};
class Y : public X              //publicly derived class
{
 public:
  Y()
   {
    cout<<“Constructors exist in only in derived class”;
   }
};

int main()
{
 Y y;       // access derived constructor
}
```

Output:

```
Constructors exist in only in derived class
```

Constructors in Derived Classes:

- ✓ *Constructor in both base & derived classes:*

```
class X                    //base class
{
 public:
  X()
   {
    cout<<“Constructors of the base class\n”;
   }};
class Y : public X                  //publicly derived class
{
 public:
  Y()
   {
    cout<<“Constructors of derived class”;
   }};

int main()
{
 Y y;      // access both class constructor
}
```

Output:

```
Constructors of the base class
Constructors of derived class
```

Constructors in Derived Classes:

- ✓ *Multiple constructor in base class & single constructor in the derived class:*

```
class X                    //base class
{
 public:
  X()       { cout<<“No-argument constructor of the base class\n”;}
  X (int a)
   {
    cout<<“one argument constructor of base class\n”;
   }};
class Y : public X              //publicly derived class
{
 public:
  Y(int a)
   {
    cout<<“one argument constructor of derived class\n”;
   }};
int main( )
{
 Y y(3);
 }
```

Output:

```
No-argument constructor of the base class
One argument constructor of derived class
```

✓ *Constructor in base & derived classes without default constructor:*

In general, the compiler looks for the no-argument constructor by default in the base class. If there is a constructor in base class, the following conditions must be met:

✓The base class must have a no-argument constructor.

✓If the base class does not have a default constructor and has an argument constructor, they must be explicitly invoked, otherwise the compiler generates an error.

```
class X                    //base class
{
 public:
  X (int a)
   { cout<<“one argument constructor of base class\n”; }
};
class Y : public X              //publicly derived class
{
 public:
  Y( int a)
   { cout<<“one argument constructor of derived class\n”; }
};
int main( )
{
 Y y(3);
 }
```

Output:

Compilation error: Cannot find ‘default’ constructor to initialize base class ‘X’

Constructors in Derived Classes:

- ✓ *Explicit invocation in the absence of default constructor:*

```
class X                    //base class
{
 public:
  X (int a)
   {
    cout<<“one argument constructor of base class X\n”;
   }
 };
class Y : public X               //publicly derived class
{
 public:
  Y (int a) : X(a)
   {
    cout<<“one argument constructor of derived class Y\n”;
   }
 };
int main( )
{
 Y y(3);
 }
```

Output:

```
One argument constructor of the base class X
One argument constructor of derived class Y
```

Constructors in Derived Classes:

- ✓ *Constructor in a multiple inherited class with default invocation:*

```
class X1                //base class
{
 public:
  X1()   { cout<<“Constructors of the base class X1\n”; }
 };
class X2                //base class
{
 public:
  X2()   { cout<<“Constructors of the base class X2\n”;  }
 };
class Y : public X2, public X1          //publicly derived class
 {
  public:
   Y()     { cout<<“Constructor of the derived class Y\n”; }
 };
int main()
{
 Y y;
}
```

Output:

```
Constructors of the base class X2
Constructors of the base class X1
Constructors of derived class Y
```

Constructors in Derived Classes:

- ✓ *Constructor in a multiple inherited class with explicit invocation:*

```
class X1                //base class
{
 public:
  X1()   { cout<<“Constructors of the base class X1\n”; }
 };
class X2                //base class
{
 public:
  X2()   { cout<<“Constructors of the base class X2\n”;  }
 };
class Y : public X1, public X2          //publicly derived class
 {
  public:
    Y() : X2(), X1()            // explicit call to constructors
     { cout<<“Constructor of the derived class Y\n”; }};
int main()
{
 Y y;      // access both the constructor
}
```

Output:

```
Constructors of the base class X1
Constructors of the base class X2
Constructors of derived class Y
```

Constructors in Derived Classes:

✓ *Constructor in base and derived classes in multiple inheritance:*

```
class X1                //base class
{
 public:
  X1()   { cout<<"Constructors of the base class X1\n"; }
 };
class X2                //base class
{
 public:
  X2()   { cout<<"Constructors of the base class X2\n";  }
 };
class Y : public X1, virtual X2         //public X1, private virtual X2
 {
  public:
    Y() : X1(), X2()
     { cout<<"Constructor of the derived class Y\n"; }};
int main()
{
 Y y;      // base constructor
}
```

Output:

```
Constructors of the base class X2
Constructors of the base class X1
Constructors of derived class Y
```

Constructors in Derived Classes:

✓ *Constructor in multilevel inheritance:*

```
class X                  //base class
{
 public:
  X()   { cout<<“Constructors of the base class X\n”; }
 };
class Y : public X              //derived class
{
 public:
  Y()   { cout<<“Constructors of the base class Y\n”;  }
 };
class Z : public Y              //publicly derived class
 {
  public:
    Z()     { cout<<“Constructor of the derived class Z\n”; }
  };
int main()
{
 Z z;      // base constructor
}
```

Output:

```
Constructors of the base class X
Constructors of the base class Y
Constructors of the derived class Z
```

Order of invocation of constructors:

Method of Inheritance	*Order of Execution*

class Y : public X
{
…..
};
	X() : base constructor
	Y() : derived constructor

class Y : public X1, public X2
{
…..
};
	X1() : base constructor
	X2() : base constructor
	Y() : derived constructor

class Y : public X1, virtual X2
{
…..
};
	X2() : virtual base constructor
	X1() : base constructor
	Y() : derived constructor

class Y1 : public X
{
…..
};
Class Y2 : public Y1
{
…..
};
	X() : super base constructor
	Y1() : base constructor
	Y2() : derived constructor

Destructors in Derived Classes:

- ✓ Unlike constructors, destructors in the class hierarchy are invoked in the reverse order of the constructor invocation.
- ✓ The destructor of that class whose constructor was executed last, while building object of the derived class, will be executed first whenever the object goes out of scope.
- ✓ If destructors are missing in any class in the hierarchy of classes, that class's destructor is not invoked.

Order of invocation of constructors & destructors:

```
class X1                //base class
{
 public:
  X1()   { cout<<“Constructors of the base class X1\n”; }
  ~ X1()  { cout<<“ Destructor in the base class X1\n”;  } };
class X2                //base class
{
 public:
  X2()   { cout<<“Constructors of the base class X2\n”;  }
  ~ X2()  { cout<<“ Destructor in the base class X2\n”;  } };
class Y : public X1, public X2          //publicly derived class
 {
  public:
    Y() :
     { cout<<“Constructor of the derived class Y\n”; }
    ~ Y()  { cout<<“ Destructor in the derived class Y\n”;
  };
int main()
{
 Y y;      // access both the constructor
}
```

Output:

```
Constructors of the base class X1
Constructors of the base class X2
Constructors of derived class Y
Destructor in the derived class Y
Destructor in the base class X2
Destructor in the base class X1
```

Constructor Invocation & Base Initialization:

- ✓ In multiple inheritance, the constructors of base classes are invoked first, in the order in which they appear in the declaration of the derived class, whereas in the case of multilevel inheritance, they are executed in the order of inheritance.
- ✓ It is the responsibility of the derived class to supply initial values to the base class constructor, when the derived class objects are created.
- ✓ Initial values can be supplied either by the object of a derived class or a constant value can be mentioned in the definition of the constructor.
- ✓ The syntax of defining a constructor in a derived class is as:

```
DerivedClass (arg_list) : Base1 (arg_list1), Base2 (arg_list2)
{
        // body of derived class constructor
}
```

- ✓ The parameters are the list of arguments passed to the constructor or they can be any constant value those match with the arguments of the constructor list of base classes.

Data Members Initialization:

- C++ supports another method of initializing the objects of classes through the use of the *initialization list* in the constructor function.
- It facilitates the initializing of data members by specifying them in the header section of the constructor.
- The general form of this method is as:

```
DerivedClass (arg_list) : InitializationSection
{
        // body of derived class constructor
}
```

- Data member initialization is represented by:

  DataMemberName (value)

- The value can be arguments of a constructor, expression or other data members.
- The data member to be initialized must be a member of its own class.

Constructor Invocation & Data Members Initialization:

- ✓ The following rules must be noted about the initialization and order of invocation of constructors:
- ✓ The initialization statements (in the initialization section) are executed in the order of definition of data members in the class.
- ✓ Constructors are invoked in the order of inheritance. However, the following rules apply when class is instantiated:
- ✓ first the constructors of virtual base classes are invoked,
- ✓ second, any non-virtual classes,
- ✓ and finally, the derived class constructor.

Data Members Initialization:

```
class B
{
protected:
        int x,y;
public:
        B (int a, int b) : x(a), y(b) { } // x=a, y=b
};
class D : public B
{
private:
        int a,b;
public:
        D (int p, int q, int r) : a(p), B(p,q), b(r)   { }
void output()
        {
        cout<<“x=“<<x<<“\ty=“<<y<<endl;
        cout<<“a=“<<a<<“\tb=“<<b<<endl;
        }
};
```

```
int main()
 {
  D obj (5, 10, 15);
  obj.output ();
 }
```

Output:

```
x= 5        y= 10
a= 5        b= 15
```

Data Members Initialization:

- ✓ The following examples illustrates the initialization of data members with different formats:
- ✓ **B(int a, int b) : x(a), y(a+b)**

  Here, the data member x is assigned the value a & y is assigned the value of the expression (a+b).
- ✓ **B(int a, int b) : x(a), y(x+b)**

  Here, the data member x is assigned the value a & y is assigned the value of the expression (x+b).
- ✓ **B(int a, int b) : y(a), x(y+b)**

  Here, it produces a wrong result, because, the statement which initializes the data member x is the first one to be executed (since, x is defined first data member in the class B). Hence the computation x(y+b) i.e. x=y+b, produces a wrong result because the data member y is not yet initialized. Thus it is a run-time error not compilation error.

Arrays of Class Objects:

- Once a derived class has been defined, the way of accessing a class member of the array of class objects are same as the ordinary class types.
- The general syntax of the array of class objects is as:

```
class baseA
        {
        // body of base class
        };
class derivedB : public baseA
        {
        // body of derived class
        };
int main()
        {
        derivedB obj[100];
        }
```

Nested Class/ Object Composition:

- ✓ The two most common techniques for reusing functionality in object-oriented systems are class inheritance and object composition.
- ✓ The use of objects in a class as data members is referred to as *object composition* or *container class.*
- ✓ Object composition is an alternative to class inheritance. Here, new functionality is obtained by assembling or composing objects to support more powerful functionality.
- ✓ This new approach takes a view that an object can be a collection of many other objects and the relationship is called a *has-a relationship* or *containership.*
- ✓ In OOP, the has-a relationship occurs when an object of one class is contained in another class as a data member.
- ✓ It is represented as:

```
class B            class D
                    {
{                   B objb;
 ……
                    …..
 };                 };
```

Nested Class/ Object Composition:

- ✓ Creation of an object that contains another object is very different than the creation of an independent object.
- ✓ An independent object is created by its constructor when it is declared with arguments.
- ✓ On the other hand, a nested object is created in two stages: first the member objects are created using their respective constructors and then the other ordinary members are created.
- ✓ This means, constructors of all the member objects should be called before its own constructor body is executed.
- ✓ This is accomplished using an initialization list in the constructor of the nested class.

```
class B
{
 ……
 };
```

```
class D
 {
 B objb;
 public:
  D( arg-list) : objb(arg-list1)
  {        // constructor body        }
 };
```

Nested Class/ Object Composition:

```
class B
{
        // body of class
 };
class D
 {
 B objb;
 public:
  D( arg-list) : objb(arg-list1)
  {    // constructor body        }
 };
```

- ✓ Here arg-list is the list of arguments to be supplied during the creation of objects of class D. These parameters are used in initializing the members of class D. The arg-list1 is used to initialize the members of the class B.
- ✓ Here, the constructor of class B is executed first and then the constructor of class D.

Nested Class/ Object Composition:

Example of Initialization in Object Composition:

```
class X
 {
  int x;
  public:
  X(int a)
   {
    x=a;
    cout<<“x=“<<x;
   }
 };
```

```
class Y
 {
  X obx;
  int y;
  public:
  Y(int b) : obx(5)
   {
    y=b;
    cout<<“y=“<<y;
   }
 };
```

```
int main()
 {
  Y oby(10);
  return 0;
  }
```

Output:

```
x= 5    y= 10
```

Object Slicing:

- ✓ In Inheritance, the attributes of the base class get carried to the derived class. If a superclass instance is assigned its value from a subclass instance, member variables defined in the subclass cannot be copied, since the superclass has no place to store them.
- ✓ When a Derived Class object is assigned to Base class, the base class contents in the derived object are copied to the base class leaving behind the derived class specific contents. This is referred as ***Object Slicing***.

```
class base
{
    public:
        int i, j;
};
class derived : public base
{
     public:
         int k;
};
```

```
int main()
{
      base b;
      derived d;
      b=d;
      return 0;
}
```

- ✓ here b contains i and j where as d contains i, j & k. On assignment only i and j of the 'd' get copied into i and j of 'b'. k does not get copied. On the effect object d got **sliced.**

Object Slicing:

- ✓ Object slicing could be prevented by making the base class function pure virtual there by disallowing object creation.

```
class Base {
int d1;
public:
Base(int a) {
d1 = a;
}
virtual void display() = 0;
};

class Derived : public Base {
int d2;
public:
Derived(int a, int b) : Base(a) {
d2 = b;
}
void display() {
cout << “data1=“<<d1<<endl<<“data2=“<<d2<< endl;
}};
```

```
int main()
{
~~Base b(10);~~        //error
Derived d(100, 200);
~~b=d;~~               // error
~~b.display();~~       // error
d.display();
}
```

Output:

```
data1= 100
data2= 200
```

Delegation:

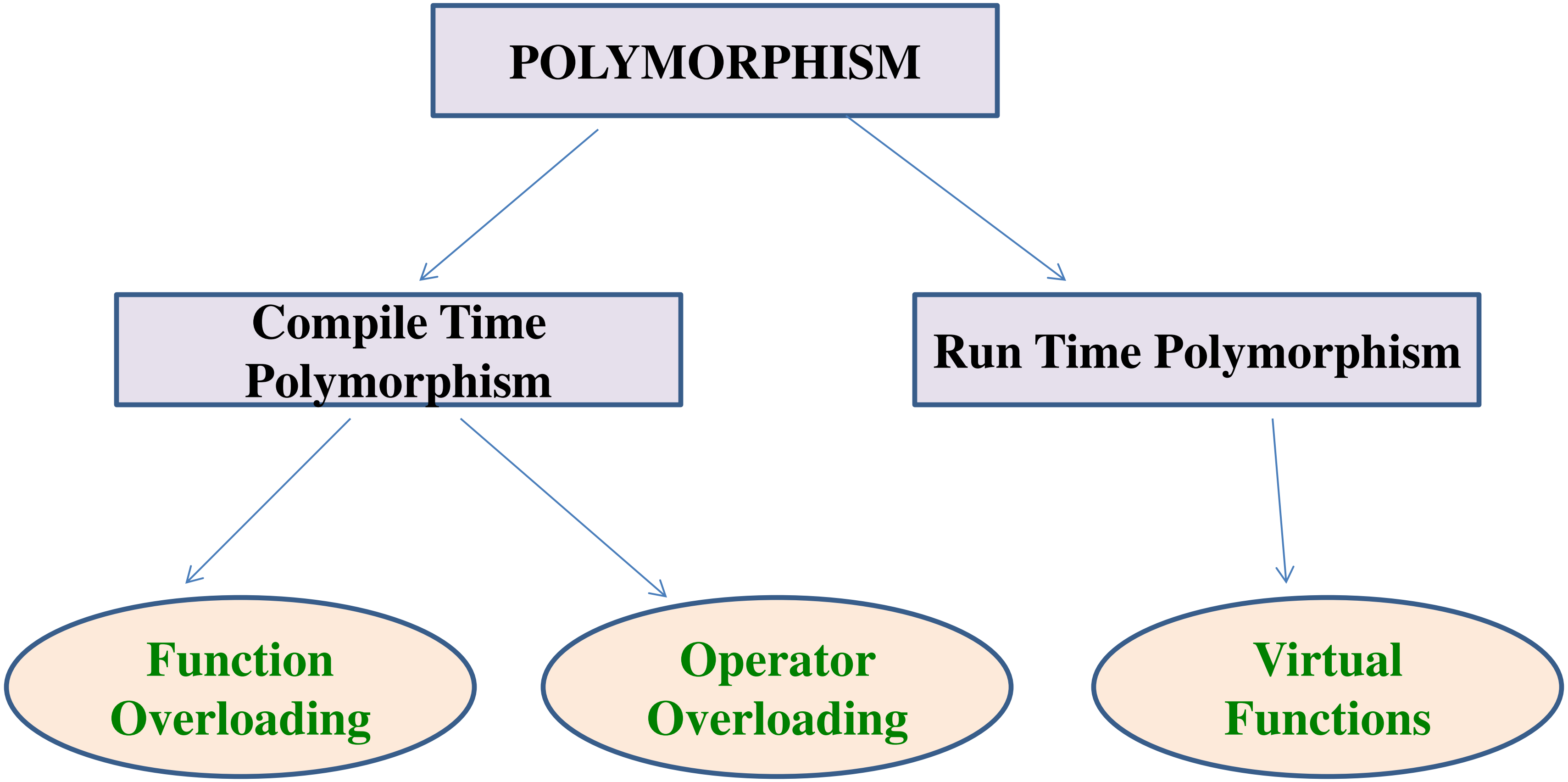
- ✓ ***Delegation*** is a way of making object composition as powerful as inheritance for reuse.
- ✓ Delegation is the simple yet powerful concept of handing a task over to another part of the program.
- ✓ In object-oriented programming it is used to describe the situation where one object defers a task to another object, known as the delegate. This mechanism is sometimes referred to as aggregation, consultation or forwarding.
- ✓ Delegation is dependent upon *dynamic binding*, as it requires that a given method call can invoke different segments of code at runtime.
- ✓ Delegation has the advantage that it can take place at run-time and affect only a subset of entities of some type and can even be removed at run-time.
- ✓ Inheritance on the other hand typically targets the type rather than the instances and is restricted to compile time.
- ✓ Delegation can be termed as "*run-time inheritance for specific objects*".

POINTERS, VIRTUAL FUNCTION & POLYMORPHISM

Ms. Sharmistha Roy
Assistant Professor,
School of Computer Engineering,
KIIT University

Introduction to Polymorphism:

- ✓ The word *Polymorphism* is very much familiar to all. It is derived from the Greek words ‘poly’ means many or multiple and ‘morphism’ refers to actions, i.e. *performing many actions with single function or operator.*



Introduction to Polymorphism:

Compile Time Polymorphism:

- ✓ The linking of the appropriate function for a particular call at the compile time itself is known as compile time polymorphism.
- ✓ This is also called early binding or static binding or static linking.
- ✓ Early binding simply means that an object is bound to its function call at compile time.
- ✓ Suppose, consider a situation where both the base and derived class has the same function with same prototype. In that case, we cannot apply static binding since the function is not overloaded.
- ✓ We have seen earlier that in such situations, we may use the class resolution operator (::) to specify the class while invoking the functions with the derived class objects.
- ✓ It would be nice if the appropriate member function could be selected while the program is running. This is known as run time polymorphism.

Introduction to Polymorphism:

Run Time Polymorphism:

- ✓ The linking of the appropriate function for a particular call during run time is known as run time polymorphism.
- ✓ At run time, when it is known what class objects are under consideration, the appropriate version of the function is invoked.
- ✓ It is achieved with the help of virtual function.
- ✓ This is also called late binding or dynamic binding.
- ✓ Dynamic binding is one of the powerful feature of C++. This requires the use of pointers to objects.

Pointers :

- ✓ Pointer is a derived data type that refers to another data variable by storing the variable's memory address rather than data.
- ✓ A pointer variable defines where to get the value of a specific data variable instead of defining actual data.
- ✓ A pointer variable can also refer to (or point to) another pointer in C++.
- ✓ *Syntax of pointer declaration:*
- ✓ data-type *pointer-variable;
- ✓ The pointer variable should contain the memory location of any integer variable.
- ✓ *Initialization of pointer variable:*

  int *ptr, a; // declaration

  ptr= &a;   // initialization
- ✓ The pointer variable, ptr contains the address of the variable a. The '&' or reference operator is used to retrieve the address of a variable.

Pointers to Objects:

✓ Pointers are also used to point to a object of a class.

✓ Syntax: **classname *ptr;**

✓ Here **ptr** is of class type and it can be used to perform the similar features just like an object. That is, it can access the member functions of a class.

✓ Let us consider a class named ***employee***.

```
employee ee;
employee *ptr = &ee;     // ptr is initialized with the address of ee
ptr->show();             // arrow operator and object pointer
(*ptr).show();           // ptr is an alias of ee
```

✓ We can also create the objects using pointers and new operator as follows:

```
employee *ptr = new employee;
employee *ptr = new employee[10];      // array of 10 objects
```

This Pointers :

- ✓ In binary operator overloading using member function, we pass only one argument to the function, the other argument is passed implicitly since it invokes the function.
- ✓ Now, this argument is actually passed using the pointer **this**.
- ✓ For e.g. the statement **return *this;** inside a member function will return the object that invoked the function.

```
person person :: greater (person x)
 {
 if(x.age >age)
   return x;          // argument object
 else
   return *this;      // invoking object
 }

int main()
 {
 person p1, p2;
 person p= p1.greater(p2);
 }
```

Here, the function will return the object p2 if the age of the person p2 is greater than that of p1, otherwise it will return the object p1 using the **pointer this**. Remember, the dereference operator * produces the contents at the address contained in the pointer.

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 e.g. 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 a pointer to D.

```
B *cptr;          // pointer to class B type variable
B b;              // base object
D d;              // derived object
cptr = &b;        // cptr points to object b
```

- ✓ We can make cptr point to the object d as: cptr= &d; because d is an object derived from the class B.
- ✓ However, there is a problem in using cptr to access the members of the derived class D. Using cptr, we can access only those members which are inherited from B and not the members that originally belong to D.
- ✓ In case, a member of D has the same name as one of the members of B, then any reference to that member by cptr will always access the base class member.

Virtual Functions :

✓ We know, polymorphism refers to the property by which objects belonging to different classes are able to respond to the same message, but in different forms.

✓ An essential requirement of polymorphism is the ability to refer to the objects of different classes i.e. the use of a single pointer variable to refer to the objects of different classes.

✓ Here, we use the base class pointer to refer to all the derived objects. But, we discovered that although a base pointer contain the address of derived class, it executes the base class function. That is, the compiler ignores the content of the pointer, it simply chooses the function that matches the type of the pointer.

✓ Thus to achieve polymorphism we use the concept of virtual function.

Virtual Functions :

- ✓ When we use the same function name in both the base and derived classes, the function in base class is declared as *virtual* using the keyword **virtual** preceding its normal declaration.
- ✓ When a function is made **virtual**, C++ determines which function to use at run time based on the type of object pointed to by the base pointer, rather than the type of the pointer.
- ✓ Thus, by making the base pointer to point to different objects, we can execute different versions of the virtual function.
- ✓ One important point to remember is that, we must access *virtual* functions through the use of a pointer declared as a pointer to the base class.

Virtual Functions :

```
class Base
{
 public:
  void display()
  { cout<<“Display Base”; }
  virtual void show()
  { cout<<“Show Base”; }
};

class Derived : public Base
{
 public:
  void display ()
  { cout<<“Display Derived”; }
  void show ()
  { cout<<“Show Derived”; }
};
```

```
int main ()
{
 Base b;
 Derived d;
 Base *bptr;
 bptr = &b;
 bptr -> display();
 bptr -> show();
 bptr = &d;
 bptr -> display();
 bptr -> show();
 return 0;
 }
```

Output:

```
Display Base
Show Base

Display Base
Show Derived
```

Rules for Virtual Functions :

- ✓ The virtual functions must be members of some class.
- ✓ They cannot be static members.
- ✓ They are accessed by using object pointers.
- ✓ A virtual function can be a friend of another class.
- ✓ A virtual function in a base class must be defined, even though it may not be used.
- ✓ The prototypes of the base class version of a virtual function and all the derived class versions must be identical. If two functions with the same name have different prototypes, C++ considers them as overloaded functions, and the virtual function mechanism is ignored.
- ✓ We cannot have virtual constructors, but we can have virtual destructors.
- ✓ While a base pointer can point to any type of the derived object, the reverse is not true. That is to say, we cannot use a pointer to a derived class to access an object of the base type.

Rules for Virtual Functions :

- ✓ When a base pointer points to a derived class, incrementing or decrementing it will not make it to point to the next object of the derived class. It is incremented or decremented only relative to its base type. Therefore, we should not use this method to move the pointer to the next object.
- ✓ If a virtual function is defined in the base class, it need not be necessarily redefined in the derived class. In such cases, calls will invoke the base function.

Virtual Destructor :

- ✓ Virtual destructor are needed for proper deletion of objects of derived class, when pointed to by a base class pointer.
- ✓ If virtual destructors are not defined, only the base class subobject is deleted, and the remaining portion of the derived class object is not deleted.
- ✓ A derived class destructor is the function called when delete is invoked with a base class pointer with the content as the derived class object.

Virtual Destructor :

```
class Base
{
 public:
  Base()
  { cout<<“Base class constructor”; }
  ~Base()
  { cout<<“Base class destructor”; }
};

class Derived : public Base
{
 public:
  Derived()
  { cout<<“Derived class constructor”;
    }
  ~Derived()
  { cout<<“Derived class destructor”; }
};
```

```
int main ()
{
 Base *bptr;
 Derived od;
 bptr = &od;
 delete bptr;
 return 0;
 }
```

Output:

```
Base class constructor
Derived class constructor
Base class destructor
```

Virtual Destructor :

```
class Base
{
 public:
  Base()
  { cout<<“Base class constructor”; }
  virtual ~Base()
  { cout<<“Base class destructor”; }
};

class Derived : public Base
{
 public:
  Derived()
  { cout<<“Derived class constructor”;
    }
  ~Derived()
  { cout<<“Derived class destructor”; }
};
```

```
int main ()
{
 Base *bptr;
 bptr = new Derived();
 delete bptr;
 return 0;
 }
```

Output:

```
Base class constructor
Derived class constructor
Derived class destructor
Base class destructor
```

Pure Virtual Functions :

- A ***pure virtual function*** is a function declared in a base class that has no definition relative to the base class. Such function is also called as “do-nothing” function.
- Syntax: ***virtual <return type> function_name ( ) = 0;***
- A virtual function, equated to zero is called a pure virtual function.
- In such cases, the compiler requires each derived class to either define the function or redeclare it as a pure virtual function.
- A pure virtual function can also be defined outside the class.
- A class containing pure virtual functions cannot be used to declare any objects of its own. Such classes, are called ***Abstract Base Classes***.

Pure Virtual Functions :

```
class B
{
 public:
  virtual void show() = 0;
};
class D1 : public B
{
 public:
  void show()
  {
      cout<<“1st Derived class ”;
  }
};
class D2 : public B
{
 public:
  void show()
  {
      cout<<“2nd Derived class ”;
  }
};
```

```
int main ()
{
 Base *b[2];
 D1 d1;
 D2 d2;
 b[0] = &d1;
 b[1] = &d2;
 b[0]->show();
 b[1]-> show();
 return 0;
 }
```

Output:

```
1st Derived class
2nd Derived class
```

DATA FILE OPERATIONS

Ms. Sharmistha Roy
Assistant Professor,
School of Computer Engineering,
KIIT University

Introduction :

- ✓ 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 performs two kinds of data communication:
- ✓ Data transfer between the console unit and the program.
- ✓ Data transfer between the program and a disk file.

Introduction :

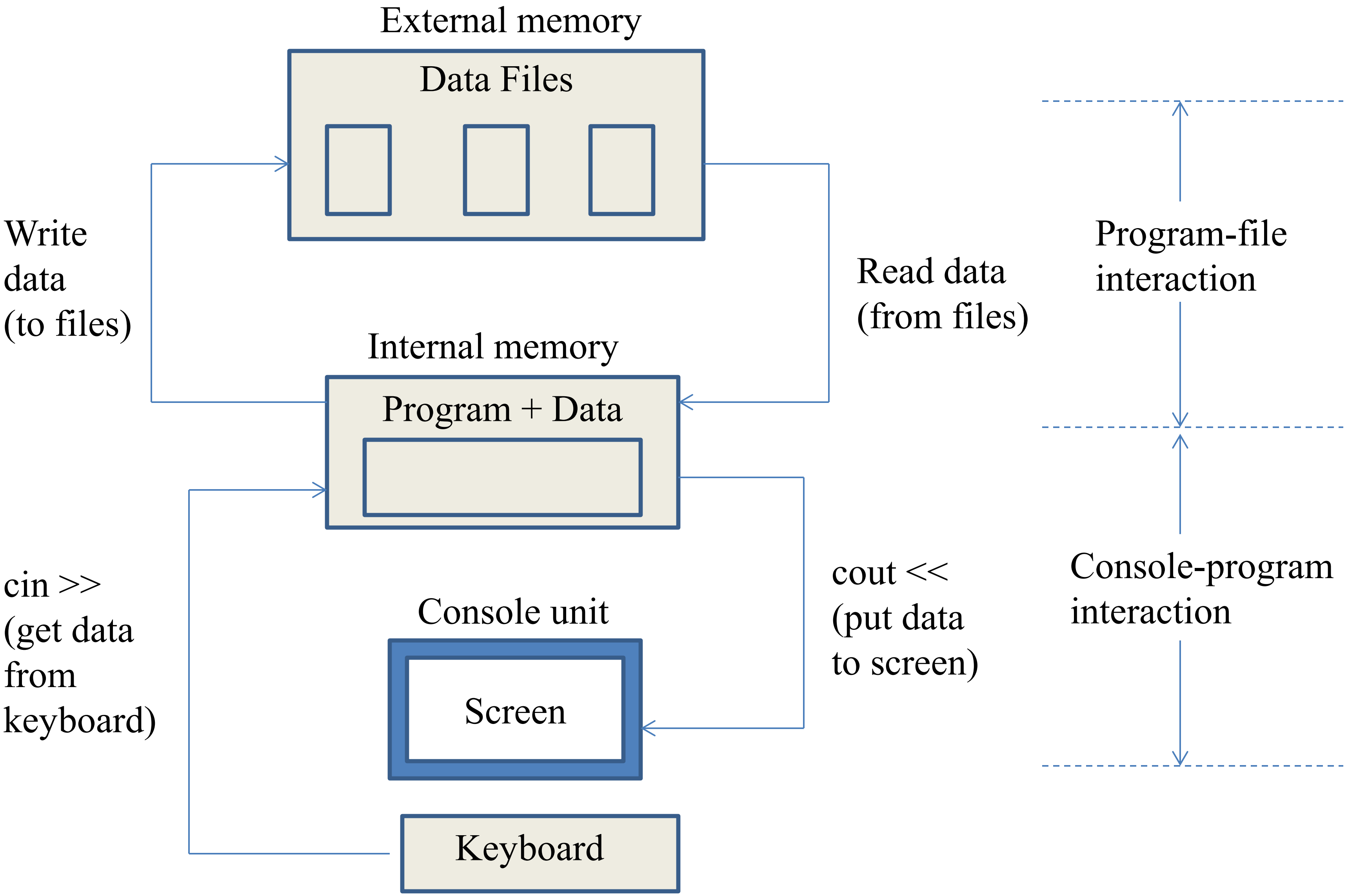


Fig: Console-program-file interaction

Introduction :

- ✓ The I/O system of C++ handles file operations which is 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.*

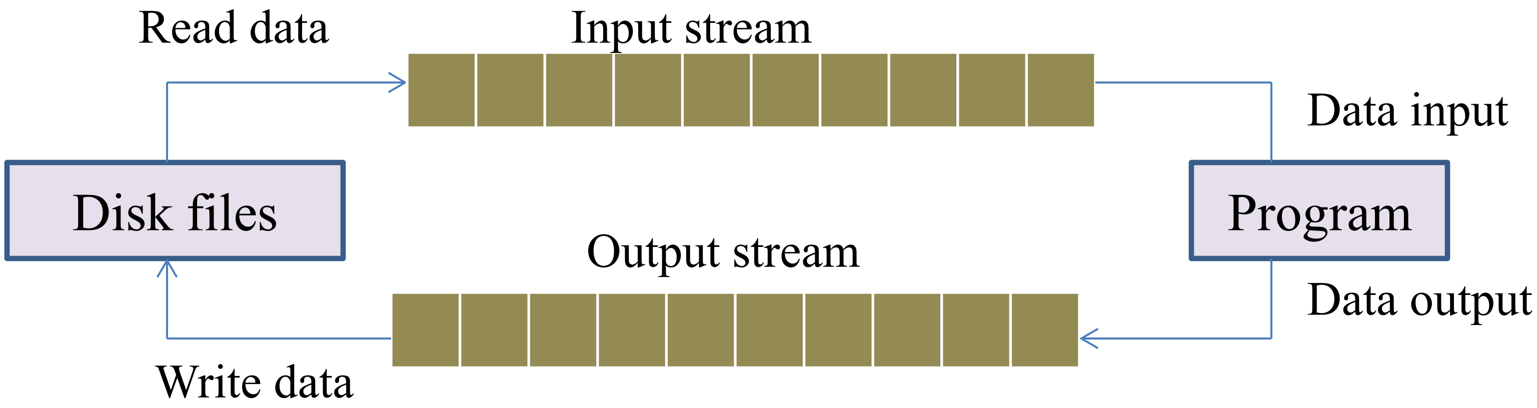


Fig: File input and output streams

Classes for File Stream Operations:

- ✓ The I/O system of C++ contains a set of classes that define the file handling methods.
- ✓ These include ifstream, ofstream and fstream. These classes are derived from fstreambase and from the corresponding iostream class as shown below:

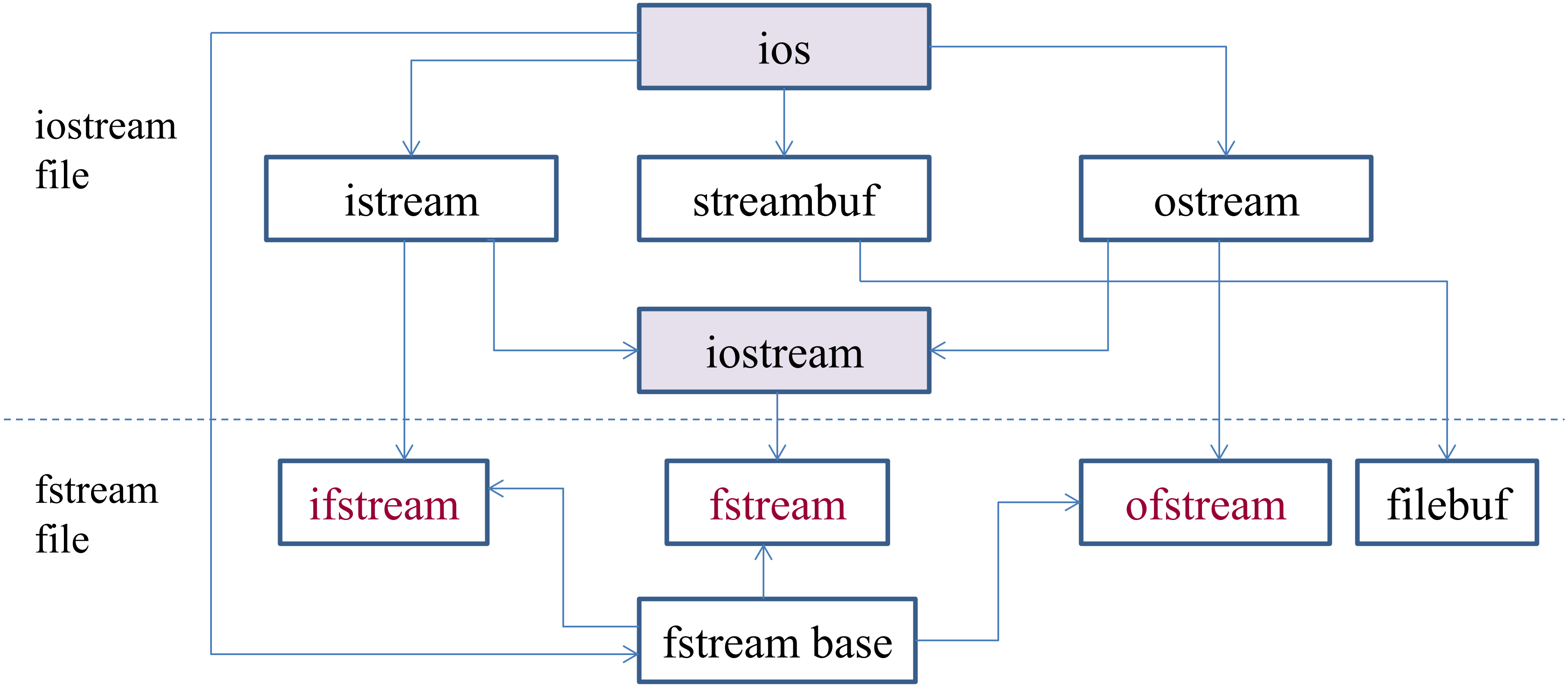


Fig: Stream classes for file operations (contained in fstream file)

- ✓ These hierarchy of classes are declared in the file "***fstream***".

Ms. Sharmistha Roy,
Assistant Professor, SCE , KIIT

Opening and Closing a File :

- ✓ If we want to use a disk file, we need to decide the following things about the file and its intended use:
- ✓ ***Suitable name for the file:*** the filename is a string of characters that make up a valid filename for the operating system. It may contain two parts, a primary name and an optional period extension. Examples: Input.data, Test.doc, student etc.
- ✓ ***Data type and structure:*** for opening a file, we must first create a file stream and then link it to the filename. A file stream can be defined using the classes **ifstream, ofstream,** and **fstream** that are contained in the headerfile *fstream*.
- ✓ **Purpose:** 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.
- ✓ **Opening method:** 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 and Closing a File :

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:
- ✓ Create a file stream object to manage the stream using the appropriate class. That is, the class **ofstream** is used to create the output stream and the class **ifstream** to create the input stream.
- ✓ Initialize the file object with the desired filename.

Opening and Closing a File :

Opening Files Using Constructor

- ✓ For example, the following statements opens a file named “results” for output: **ofstream outfile (“result”);**   // output only
- ✓ This creates *outfile* as an ofstream object that manages the output stream. This object can be any valid C++ name such as o_file, myfile or fout.
- ✓ This statement also opens the file *result* and attaches it to the output stream outfile.
- ✓ Similarly, the statement: **ifstream infile(“data”);**   //input only
- ✓ declares *infile* as an ifstream object and attaches it to the file *data* for reading input.

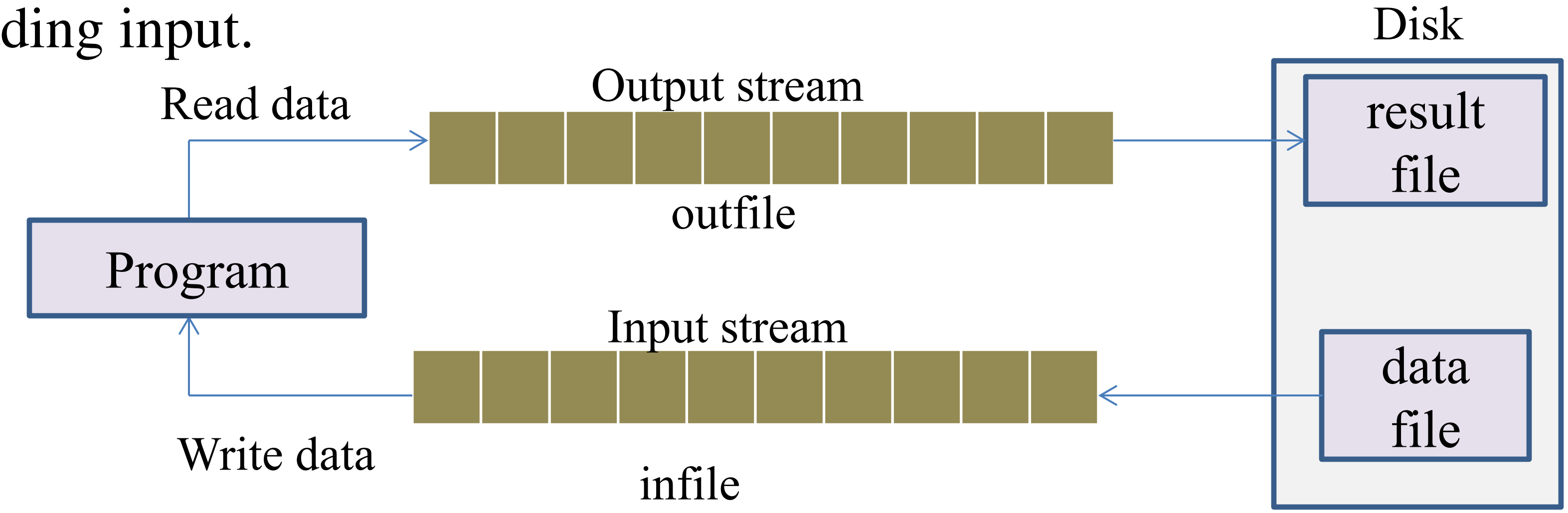


Fig: Two file streams working on separate files

Opening and Closing a File :

Opening Files Using Constructor

- ✓ We can also use same file for both reading and writing data as follows:

Program 1

```
……
……
ofstream outfile("salary"); // creates outfile and
                            // connects "salary to it
……
……
```

Program 2

```
……
……
ifstream infile("salary"); // creates infile and
                           // connects "salary to it
……
……
```

✓The connection with a file is closed automatically when the stream object expires (when the program is terminated).

✓In the above statement, when *program1* is terminated, the salary file is disconnected from the outfile stream.

✓Similar action takes place when the *program 2* terminates.

- ✓ We can manually close a file using the statement is: **outfile.close();** which disconnects salary from outfile.

Opening and Closing a File :

Opening Files Using Constructor

- ✓ Instead of using two programs, we can use a single program for writing data (output) and reading data (input).
- ✓ But we created two file stream objects, *outf* (to put data) and *inf* (to get data from file).

```
#include<iostream.h>
#include<fstream.h>
void main()
{
ofstream outf(“ITEM”);    // connect ITEM file to outf
cout<<“Enter item name”;
char name[30];
cin>>name;
outf<<name<<“\n”;         // write to file ITEM
outf.close();             // disconnect ITEM file from outf
ifstream inf(“ITEM”);     // connect ITEM file to inf
inf>>name;                // read name from file ITEM
cout<<“Item name=“<<name;
inf.close();               // disconnect ITEM file from inf
}
```

Output:

```
Enter item name
CD-ROM
Item name= CD-ROM
```

Opening and Closing a File :

Opening Files Using open()

- ✓ The function **open()** can be used to open multiple files that use the same stream object.
- ✓ Syntax of open():

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

- ✓ For example:

```
ofstream outfile;  // creates stream (for output)
outfile.open("DATA1");     // connect stream to DATA1
……..
outfile.close();      // disconnect stream from DATA1
outfile.open("DATA2);       // connect stream to DATA2
……
outfile.close();      // disconnect stream from DATA2
```

- ✓ The above code opens two files in sequence for writing the data. But the first file should be closed before opening the second one because, a stream can connect to only one file at a time.

More about Open(): File Modes:

- ✓ Till now we have ifstream & ofstream class and the function open() to create new files as well as to open the existing files. Remember, in both these methods, we used only one argument that was the filename.
- ✓ However, these functions can take two arguments, the second one is for specifying the *file mode*.
- ✓ The general form of the function open() with two arguments is:
- ✓ ***stream-object.open("filename", mode);***
- ✓ The second argument *mode* specifies the purpose for which the file is opened.
- ✓ By default the "***ifstream***" open the file in "***reading***" mode and "***ofstream***" open the file in "***writing***" mode only.

More about Open(): File Modes:

✓ The list of file mode parameters are given below:

Parameter	Meaning
ios::app	Append to end-of file
ios::ate	Go to end-of file on opening
ios::binary	Binary file
ios::in	Open file for reading only
ios::nocreate	Open fails if the file does not exist
ios::noreplace	Open fails if the file already exist
ios::out	Open file for writing only
ios::trunc	Delete the contents of the file if it exists

✓ The mode can combine two or more parameters using the bitwise OR operator (|) as shown below:

✓ Fout.open(“data”, ios::app | ios::nocreate)

✓ This opens the file in the append mode but fails to open the file if it does not exist.

Detecting end-of-file :

- Detection of the end-of-file condition is necessary for preventing any further attempt to read data from the file.
- This can be done using the statement: **while(fin)**
- An ifstream object **fin**, returns a value 0 if any error occurs in the file operation including the end-of-file condition.
- Thus the while loop terminates when fin returns a value of zero on reaching the end-of-file condition.
- There is another approach to detect the end-of-file condition.
- **if(fin.eof()!=0)**

  { exit(1);}
- **eof()** is a member function of the ios class. It returns a non-zero value if the end-of-file condition is encountered, and a zero otherwise.
- Therefore, the above statement terminates the program on reaching the end of the file

Working with Multiple files

```
void main()
{
    ofstream fout;                  // create output stream
    fout.open(“Country”);           // connect “Country” to it
    fout<<“INDIA”;
    fout.close();                   // disconnect “Country”
    fout.open(“Capital”);           // connect “Capital”
    fout<<“NEW DELHI”;
    fout.close();                   // disconnect “Capital”
    // Reading the files
    char line[40];
    ifstream fin;                   // create input stream
    fin.open(“Country”);            // connect “Country to it”
    while(fin)                      // check end-of-file
    {  fin.getline(line, 40);       // read a line
     cout<< line;  }                // display it
    fin.close();                    // disconnect “Country”
    fin.open(“Capital”);            // connect “Capital to it”
    while(fin)
    {  fin.getline(line, 40);
     cout<< line;  }
    fin.close();
}
```

Output:

```
INDIA
NEW DELHI
```

Opening and Closing a File :

Opening Files Using open()

Working with multiple files:

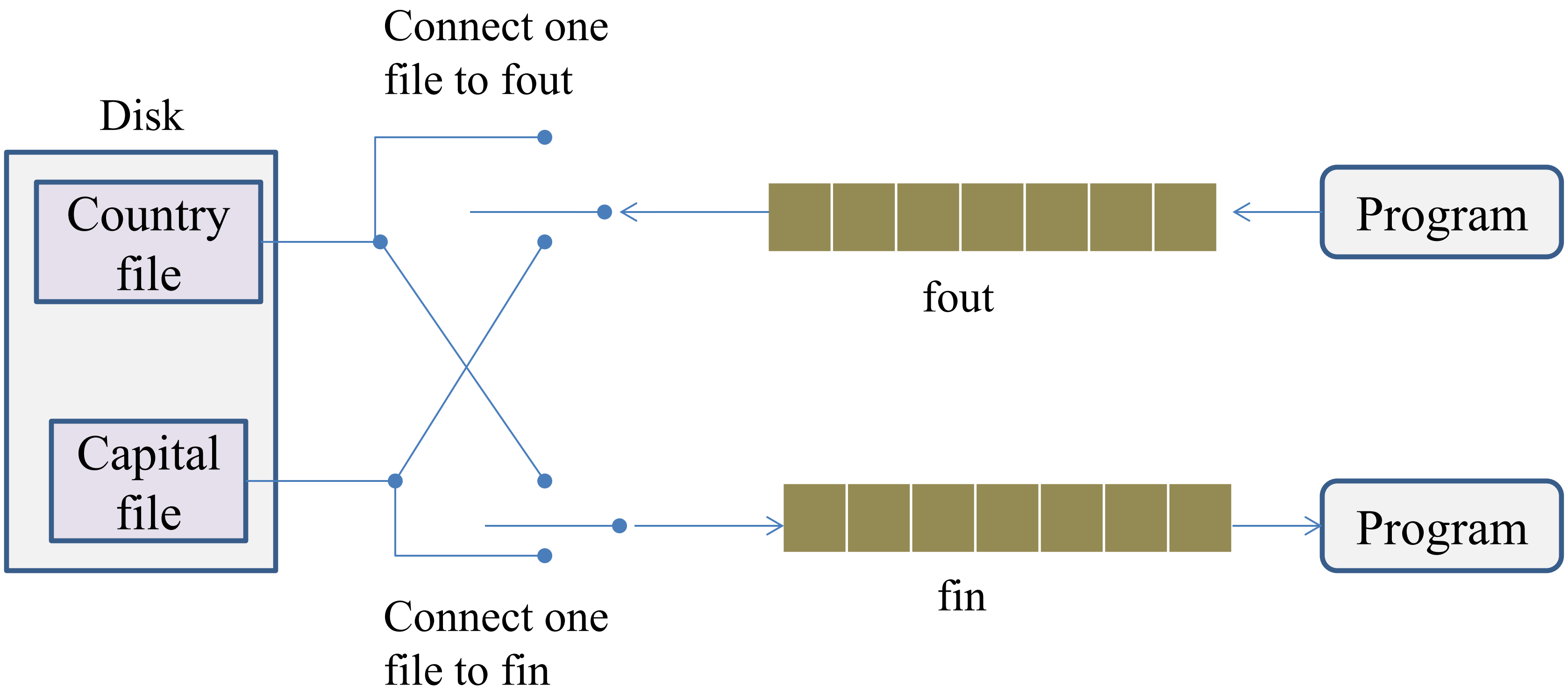


Fig: Streams working on multiple files

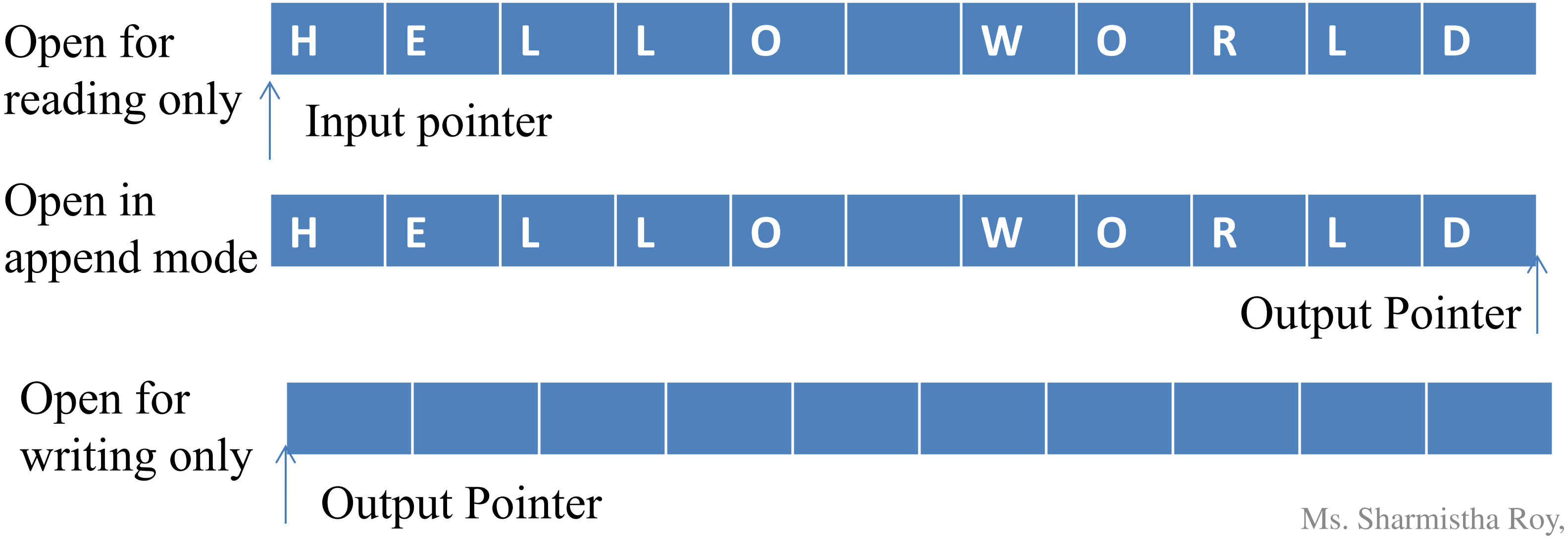
File Pointers and their Manipulations:

- ✓ Each file has two associated pointers known as the *file pointers*.
- ✓ One of them is called the input pointer (*or get pointer*).
- ✓ The other is called the output pointer (*or put pointer*).
- ✓ The input pointer is used for reading the contents of a given file location and the output pointer is used for writing to a given file location.

File Pointers and their Manipulations:

Default Actions:

- ✓ When we open a file in read-only mode, the input pointer is automatically set at the beginning so that we can read the file from the start.
- ✓ Similarly, when we open a file in write-only mode, the existing contents are deleted and the output pointer is set at the beginning.
- ✓ Incase, we want to open an existing file to add more data, the file is opened in '*append*' mode. This moves the output pointer to the end of the file.



Open for reading only

H	E	L	L	O		W	O	R	L	D

Input pointer

Open in append mode

H	E	L	L	O		W	O	R	L	D

Output Pointer

Open for writing only

Output Pointer

Functions for Manipulation of File Pointers:

- ✓ All the actions on the file pointers take place automatically by default.
- ✓ Now, for moving the file pointer to any desired position inside the file we have to take control of the movement of the file pointers using the following functions that are defined in the stream classes:
- ✓ ***seekg():*** moves get pointer (input) to a specified location.
- ✓ ***seekp():*** moves put pointer (output) to a specified location.
- ✓ ***tellg():*** gives the current position of the get pointer.
- ✓ ***tellp():*** gives the current position of the put pointer.
- ✓ For example, the statement infile.seekg(10); moves the file pointer to the byte number 10. That is, the pointer will be pointing to the 11th byte in the file.

```
ofstream fileout;
fileout.open(“hello”, ios::app);
int p= fileout.tellp();
```

- ✓ On execution of these statements, the output pointer is moved to the end of the file “hello” and the value of p will represent the number of bytes in the file.

Functions for Manipulation of File Pointers:

Specifying the offset

- ✓ ‘Seek’ functions seekg() and seekp() can also be used with two arguments as follows:
- ✓ **seekg(offset, refposition);** or **seekp(offset, refposition);**
- ✓ The parameter *offset* represents the number of bytes the file pointer is to be moved from the location specified by the parameter function.
- ✓ The *refposition* takes one of the following three constants defined in the ios class:
- ✓ **ios::beg** start of the file
- ✓ **ios::cur** current position of the pointer
- ✓ **ios::end** end of the file
- ✓ *seekg()* function moves the associated file’s ‘get’ pointer while the *seekp()* function moves the associated file’s ‘put’ pointer.

Sequential Input and Output Operations:

- ✓ There are several member functions supported by file stream classes for performing the input and output operations on files.
- ✓ One pair of functions, **put()** and **get()** are designed for handling a single character at a time.
- ✓ Another pair of functions, **write()** and **read()** are designed to write and read blocks of binary data.

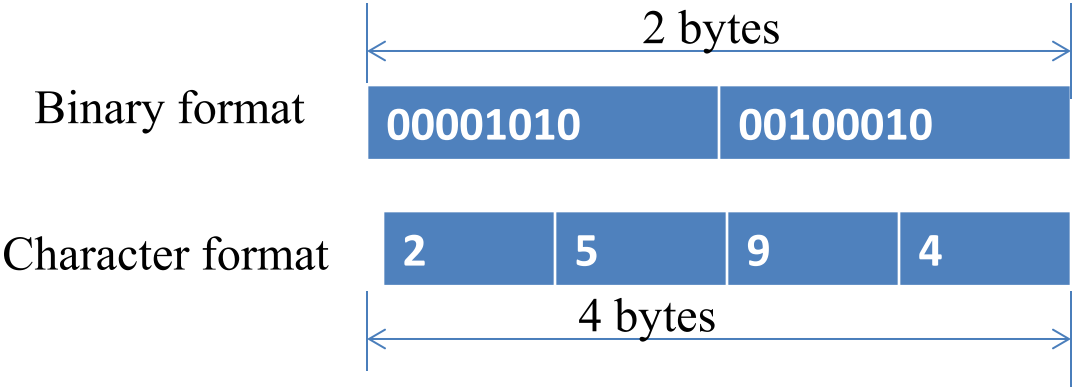
put() and get() functions:

- ✓ The function put() writes a single character to the associated stream.
- ✓ Similarly, the function get() reads a single character from the associated stream.

Sequential Input and Output Operations:

write() and read() functions:

- ✓ The functions **write()** and **read()** handles the data in binary form.
- ✓ This means the values are stored in the disk file in the same format in which they are stored in the internal memory.
- ✓ Figure below shows how an integer value 2594 is stored in binary and character formats:
- ✓ An **int** takes two bytes to store its value in the binary form, irrespective of its size. But a 4 digit int will take four bytes to store it in the character form.



2 bytes

Binary format

00001010 00100010

Character format

2 5 9 4

4 bytes

- ✓ The binary format is more accurate for storing the numbers as they stored in the exact internal representation.
- ✓ There are no conversion while saving the data and therefore saving is much faster.

Sequential Input and Output Operations:

write() and read() functions:

- ✓ The binary input and output functions takes the following form:
- ✓ infile.**read** ((char *) &V, sizeof (V));
- ✓ outfile.**write** ((char *) &V, sizeof (V));
- ✓ These functions take two arguments. The first is the address of the variable V, and the second is the length of that variable in bytes.
- ✓ The address of the variable must be cast to type **char*** (i.e. pointer to character type).

Exception Handling

Ms. Sharmistha Roy
Assistant Professor,
School of Computer Engineering,
KIIT University

Introduction :

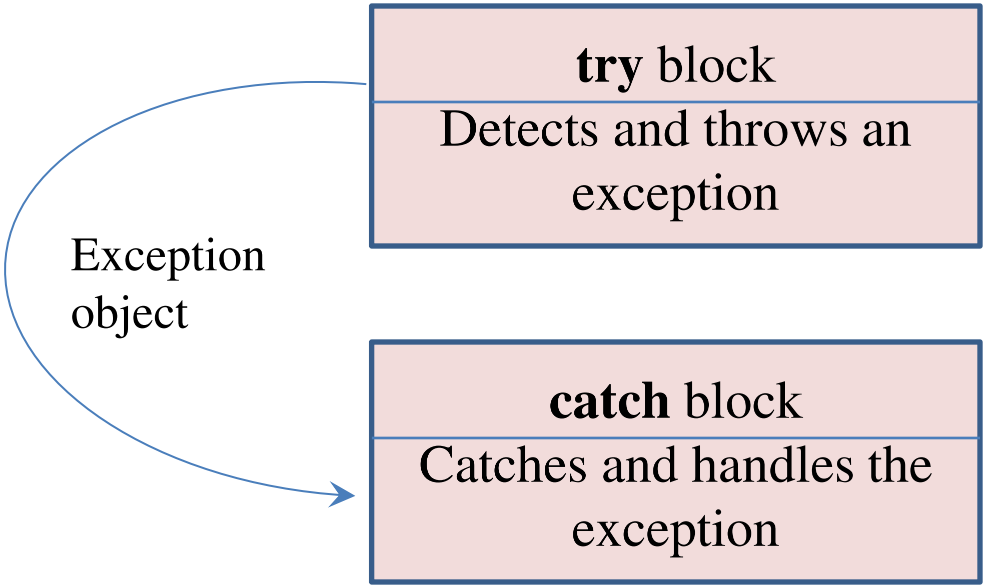
- We know that it is very rare that a program works correctly first time. It might have bugs.
- The two most common types of bugs are *logic errors* and *syntactic errors*.
- We can detect these errors by using exhaustive debugging and testing procedures.
- We often come across some peculiar problems other than logic or syntax errors. They are known as **exceptions**.
- **Exceptions are runtime anomalies or unusual conditions that a program may encounter at runtime or while executing**. Anomalies might include conditions such as division by zero, access to an array outside of its bounds, or running out of memory or disk space.
- When a program encounters an exceptional condition, it is important that it is identified and dealt effectively. This is known as ***Exception Handling***.

Basics of Exception Handling :

- Exceptions are of two kinds, namely, Synchronous exceptions and asynchronous exceptions.
- Synchronous exceptions: “out-of-range index” and “over-flow” errors.
- Asynchronous exceptions: error that are caused by events beyond the control of the program (such as keyboard interrupts).
- C++ is designed to handle only synchronous exceptions.
- **Purpose:**
- To provide means to detect and report an “exceptional circumstances” so that appropriate action can be taken. The mechanism suggests a separate error handling code that performs the following tasks:
- Find the problem (*Hit the exception*).
- Inform that an error has occurred (*Throw the exception*).
- Receive the error information (*Catch the exception*).
- Take corrective actions (*Handle the exception*).

Exception Handling Mechanism:

- ✓ Exceptions Handling mechanism is built upon three keywords:
- ✓ **try:** the keyword *try* is used to preface a block of statements which may generate exceptions. This block of statements is known as try block.
- ✓ **throw:** when an exception is detected, it is thrown using a *throw* statement in the try block.
- ✓ **catch:** a ‘catch block’ defined by the keyword *catch* ‘catches’ the exception ‘thrown’ by the throw statement in the try block, and handles it appropriately.



try block

Detects and throws an exception

Exception object

catch block

Catches and handles the exception

Fig: The block throwing Exception

Exception Handling Mechanism:

✓ The catch block that catches an exception must immediately follow the try block that throws the exception.

✓ The general form of these two blocks are as follows:

```
try
{
    throw exception;        // block of statements which detects & throws an exception
}
catch (type arg)            // catches exception
{
    …..                     // block of statements that handles the exception
}
```

✓ When the try block throws an exception the program control leaves the try block & enters the catch statement of the catch block, if the type of object thrown matches the arg type.

✓ If they donot match the program is aborted with the help of ‘abort()’ function invoked by default.

✓ When no exception is detected and thrown, the control goes to the statement immediately after the catch block. That is, the catch block is skipped.

Example of Try-catch Mechanism:

```
int main ()
{
 int a,b;
 cout<<"enter values of a& b";
 cin>>a>>b;
 int x= a-b;
 try
  {
    if(x!=0)
      {
           cout<<"result (a/x)=" << a/x;
       }
    else                // there is an exception
     {
           throw(x);              // throws int object
     }
  }
 catch (int i)          // catches the exception
  {
    cout<<" Exception caught: x="<< i;
   }
cout<<"END";  return 0;
}
```

Output:

```
First Run
Enter values of a& b
20 15
Result(a/x)= 4
END

Second Run
Enter values of a& b
10 10
Exception caught: x=0
END
```

Exception Handling Mechanism:

- ✓ Most often, ***exceptions are thrown by functions*** that are invoked from within the try blocks.
- ✓ The point at which the throw is executed is called the *throw point*.
- ✓ Once an exception is thrown to the catch block, control cannot return to throw point.

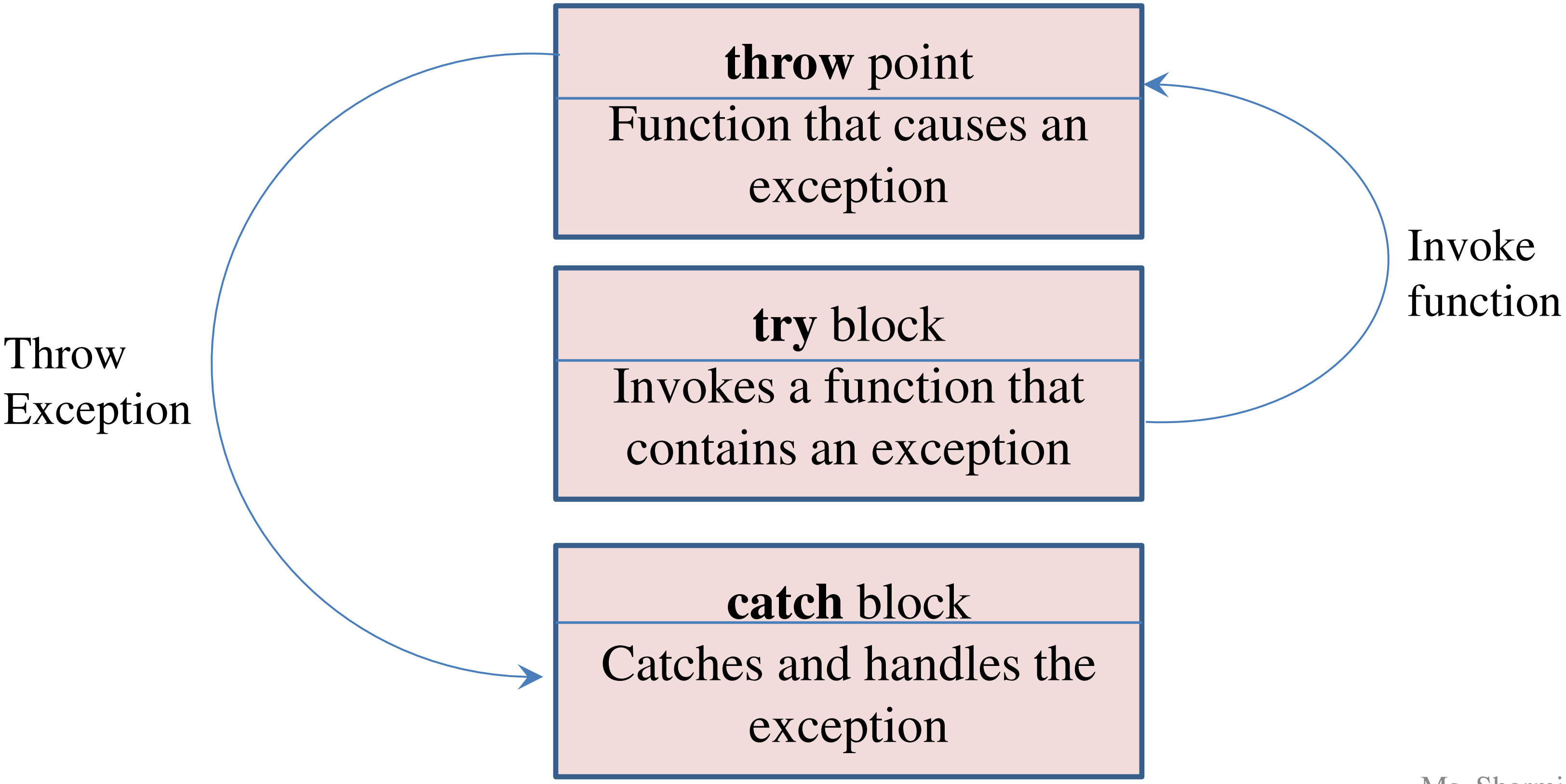

throw point
Function that causes an exception
try block
Invokes a function that contains an exception
catch block
Catches and handles the exception
Invoke function
Throw Exception

Fig: Function invoked by try block throwing exception

Invoking function that generates exception:

```
void divide (int x, int y, int z)
{
 cout <<“Inside the function\n”;
 if ( (x-y) ! = 0)
  {
   int R=z/(x-y);
   cout<<“Result= “<<R;
   }
   else
    throw (x-y);     // Throw point
 }
```

```
int main()
{
try
 {
  cout<<“Inside try block\n”;
  divide (10,20,30);
  divide (10,10,20);
  }
 catch (int i)
  {
   cout<<“Exception caught”;
  }
 return 0;
 }
```

Output:

```
Inside try block
Inside the function
Result = -3
Inside the function
Exception caught
```

Throwing Mechanism:

- ✓ When an exception that is desired to be handled is detected, it is thrown using the throw statement in one of the following forms:
- ✓ throw(exception);
- ✓ throw exception;
- ✓ throw;
- ✓ The operand object exception may be of any type, including constants. It is also possible to throw objects not intended for error handling.
- ✓ When an exception is thrown, it will be caught by the catch statement associated with the try block. That is, the control exits the current try block, and is transferred to the catch block after that try block.
- ✓ Throw point can be in a deeply nested scope within a try block or in a deeply nested function call. In any case, control is transferred to the catch statement.

Catching Mechanism:

- ✓ As stated earlier, code for handling exceptions is included in catch blocks. A catch block looks like a function definition and is of the form:
- ✓ catch (type arg)

```
catch (type arg)
{
        // statements for managing exceptions
}
```

- ✓ The type indicates the type of exception that catch block handles. The parameter arg is an optional parameter name.
- ✓ The exception-handling code is placed between two braces. The catch statement catches an exception whose type matches with the type of catch argument. When it is caught, the code in the catch block is executed. After executing the handler, the control goes to the statement immediately following the catch block.
- ✓ Due to mismatch, if an exception is not caught, abnormal program termination will occur. It is important to note that the catch block is simply skipped if the catch statement does not catch an exception.

Catching Mechanism:

Multiple Catch Statements:

✓ It is possible that a program segment has more than one condition to throw an exception.

✓ In that cases, we can associate more than one catch statement with a try as shown below:

```
try
{
    // try block
}
catch (type1 arg)
{
    // catch block1
}
……….
catch (typeN arg)
{
    // catch blockN
}
```

Catching Mechanism:

Multiple Catch Statements:

- ✓ When an exception is thrown, the exception handlers are searched in order for an appropriate match.
- ✓ The first handler that yields a match is executed. After executing the handler, the control goes to the first statement after the last catch block for that try.
- ✓ When no match is found, the program is terminated.
- ✓ It is possible that arguments of several catch statements match the type of an exception. In such cases, the first handler that matches the exception type is executed.

Catching Mechanism:

Catch All Exceptions:

- ✓ In some situations, we may not be able to anticipate all possible types of exceptions and therefore may not be able to design independent catch handlers to catch them.
- ✓ Is such circumstances, we can force a catch statement to catch all exceptions instead of a certain type alone. This could be achieved by defining the catch statement using ellipses as follows:
- ✓ catch (…)

```
catch (…)
{
        // statements for processing all exceptions
}
```

- ✓ One point to remember is that catch(…) should be placed last in the list of handlers. Placing it before other catch blocks would prevent those blocks from catching exceptions.

Rethrowing an Exception:

- ✓ A handler may decide to rethrow the exception caught without processing it. In such situations, we may simply invoke throw without any arguments as shown below:
- ✓ throw;
- ✓ This causes the current exception to be thrown to the next enclosing try/catch sequence and is caught by a catch statement listed after that enclosing try block.
- ✓ When an exception is rethrown, it will ***not be caught by the same catch statement*** or any other catch in that group. Rather, it will be caught by an appropriate ***catch in the outer try/catch*** sequence only.

Example of Rethrowing an Exception:

```
void divide(int x, int y)
{
 try
{
if(y = = 0)
 throw y;
else
 cout<<“division=“ <<(x/y);
}
catch (int)
{
 cout<<“Caught exception inside function”;
 throw;
}
}
```

```
int main()
 {
 try
 {
 divide(10,2);
 divide(2,0);
 }
 catch (int)
 {
 cout<<“Caught exception inside main”;
 }
 cout<<“END of main”;
 }
```

Output:

```
Division = 5
Caught exception inside function
Caught exception inside main
END of main
```

Specifying Exceptions:

- ✓ It is possible to restrict a function to throw only certain specified exceptions. This is achieved by adding a **throw** *list* clause to the function definition.
- ✓ The general form of using an *exception specification* is:

```
  type function (arg-list) throw (type-list)
  {
       …….
       ……. // function body
       …….
  }
```

- ✓ The *type-list* specifies the type of exceptions that may be thrown. Throwing any other type of exception will cause abnormal program termination.
- ✓ If we wish to prevent a function from throwing any exception, we may do so by making the type-list empty. That is, we must use:
- ✓ **throw();** in the function header-line.

Example of Specifying Exception:

```
void func (int num) throw(int, char)
{
 if(num>0)
    throw num;
 else
    if(num<0)
            throw ‘c’;
    else
            throw 1.2345;
 }
```

Output:

```
enter value of num: 10
Integer Exception
enter value of num: -10
Char Exception
enter value of num:  0
Abnormal program termination
```

```
int main()
 {
  int num;
  cout<<“enter value of num”;
  cin>> num;
  try
  {
  func(num);
  }
  catch (int)
  { cout<<“Integer Exception”; }
  catch (char)
  { cout<<“Char Exception”; }
  catch (double)
  { cout<<“Double Exception”; }
 }
```

TEMPLATES

Ms. Sharmistha Roy
Assistant Professor,
School of Computer Engineering,
KIIT University

Introduction :

- ✓ ***Template*** is a new concept which enable us to define generic classes and functions and thus provides support for generic programming.
- ✓ Generic programming is an approach where generic types are used as parameters in algorithms so that they work for a variety of suitable data types and data structures.
- ✓ A template can be used to create a family of classes or functions.
- ✓ For e.g. a template for an **array** class would enable us to create arrays of various data types such as **int** array and **float** array.
- ✓ Similarly, we can define a template for a function say **mul()**, that would help us to create various versions of mul() for multiplying **int**, **float** and **double** type values.
- ✓ A template can be considered as a kind of macro. When an object of a specific type is defined for actual use, the template definition for that class is substituted with the required data type.
- ✓ Since, a template is defined with a parameter that would be replaced by a specified data type at the time of actual use of the class or function, the templates are sometimes called *parameterized classes or functions*.

Class Templates :

- *Class Template* definition is very similar to an ordinary class definition except the prefix **template <class T>** and the use of type T. This prefix tells the compiler that we are going to declare a template and use T as a type name in the declaration.
- The general format of a class template is:

```
template <class T>
class classname
{
//….
// class member specification with anonymous type T wherever appropriate
//….
};
```

- A class created from a class template is called a ***template class***. The syntax for defining an object of a template class is:
- **classname <type> objectname(arglist);**
- This process of creating a specific class from a class template is called *instantiation.*

Class Templates :

```
template <class T>
class vector
{
T *v;
int size;
public:
vector(int m)
{
v= new T [size =m];
for(int i=0; i<size; i++)
v[i]= 0;
}
vector(T *a)
{
for(int i=0; i<size; i++)
v[i]= a[i];
}
T operator *(vector &y)
{
T sum =0;
 for(int i=0; i<size; i++)
  sum+= this-> v[i] * y . v[i];
return sum;
 }};
```

```
int main()
{
int x[3]= {1,2,3};
int y[3]= {4,5,6};
float m[2] = {2.5, 3.5};
float n[2] = {3.2, 4.6};
vector <int> v1(3);     // 3 element int vector
vector <int> v2(3);
v1= x;
v2= y;
int R= v1*v2;
cout<<"R= "<< R;
vector <float> v3(2);  // 2 element float vector
vector <float> v4(2);
v3=m;
v4=n;
float S = v3*v4;
cout<<"S="<<S;
return 0;
}
```

Output:

```
R= 32
S= 24.1
```

Class Templates with Multiple Parameters:

- ✓ We can use more than one generic data type in a class template.
- ✓ They are declared as a comma-separated list within the template specification as shown below:

```
template <class T1, class T2, ….>
class classname
{
//….
// body of the class
//….
};
```

Two generic data types in a class definition:

```
template <class T1, class T2>
class Test
{
T1 a;
T2 b;
public:
Test (T1 x, T2 y)
{
 a=x;
 b=y;
}
void show()
{
 cout << a “and” << b;
}
};
```

```
int main()
{
Test <float, int> test1 (1.23, 123);
Test <int, char> test2 (100, ‘w’);
test1.show();
test2.show();
return 0;
}
```

Output:

```
1.23 and 123
100 and w
```

Function Templates:

- Like class templates, we can also define function templates that could be used to create a family of functions with different argument types.
- The general format of a function template is:

```
template <class T>
returntype function_name (argments of Type T)
{
//….
// body of the function with Type T wherever appropriate
//….
};
```

- The following example declares a swap() function template that will swap two values of a given type of data.

Function Template:

```
template <class T>
void swap(T &x, T &y)
{
 T temp =x;
 x= y;
 y= temp;
}
void func(int m, int n, float a, float b)
{
 cout<< “m and n before swap: “<<m<<“ “ <<n<<“\n”;
 swap(m,n);
 cout<< “m and n after swap: “<<m<<“ “ <<n<<“\n”;
cout<< “a and b after swap: “<<a<<“ “<<b<<“\n”;
 swap(a,b);
cout<< “a and b after swap: “<<a<<“ “<<b<<“\n”;
}
     int main()
     {
      func(100,200,11.22,33.44);
      return 0;
     }
```

Output:

```
m and n before swap: 100 200
m and n after swap: 200 100
a and b before swap: 11.22 33.44
a and b after swap: 33.44 11.22
```

Function Templates with Multiple Parameters:

- ✓ Like template classes, we can use more than one generic data type in a template statement, using a comma-separated list as shown below:

```
template <class T1, class T2, ….>
returntype function_name (arguments of types T1, T2, ...)
{
//….
// body of the function
//….
 };
```

Function Template with two generic types:

```
template <class T1, class T2>
void display(T1 x, T2 y)
{
 cout << x << “ “ <<y <<“\n”;
}
```

```
int main()
{
 display(2012, “INDIA”);
 display(12.34, 1234);
 return 0;
}
```

Output:

```
2012 INDIA
12.34 1234
```

Overloading of Template Functions:

- ✓ A template function may be overloaded either by template functions or ordinary functions of its name. In such cases, the overloading resolution is accomplished as follows:
  - ✓ Call an ordinary function that has an exact match.
  - ✓ Call a template function that could be created with an exact match.
  - ✓ Try normal overloading resolution to ordinary functions and call the one that matches.
- ✓ An error is generated if no match is found.
- ✓ Note that, no automatic conversions are applied to arguments on the template functions.

Template Function with Explicit Function:

```
template <class T>
void display(T x)
{
 cout<< “Template display:” <<x <<“\n”;
}
void display(int x)              // overloads the generic display
{
 cout<< “Explicit display:” <<x <<“\n”;
}

int main()
{
 display(100);
 display(12.34);
 display(‘C’);
 returrn 0;
}
```

Output:

```
Explicit display: 100
Template display: 12.34
Template display: C
```

Member Function Templates:

- Earlier we have seen that member functions of the template class were defined inside the class only.
- But we can define them outside the class as well.
- For this we have to remember that the member functions of the template classes themselves are parameterized by the type argument and therefore these functions must be defined by the function templates.
- The general format is:

```
template <class T>
returntype classname <T> :: function_name(arglist)
{
//….
// body of the function
//….
 }
```

Non-Type Template Arguments:

✓ We have seen that a template can have multiple arguments. It is also possible to use non-type arguments.

✓ That is, in addition to the type argument T, we can also use other arguments such as string, constant expressions and built-in types.

✓ Consider the following example:

```
template <class T, int size>
class array
{
 T a[size];
// …
//…
 };

array <int, 10> a1;            // array of 10 integers
array <float, 5> a2;           // array of 5 floats
```

✓ This template supplies the size of the array as an argument during compile time.

Namespace:

- ✓ ANSI C++ Standard has added a new keyword ***namespace*** to define a scope that could hold global identifiers.
- ✓ The best example of namespace scope is the **C++ Standard Library**.
- ✓ All classes, functions and templates are declared within the namespace named **std**. That is why we have been using the directive
- ✓ **using namespace std;**
- ✓ The ***using namespace*** statement specifies that the members defined in std namespace will be used frequently throughout the program.

Defining Namespace:

- ✓ We can define our own namespaces in our programs. The syntax is:

```
namespace namespace_name
{
//….
// declaration of variables, functions, classes, etc.
//….
 }
```

- ✓ There is one difference between a class definition and a namespace definition. The namespace is enclosed with a closing brace but no terminating semicolon.

- ✓ *Example:*

```
namespace TextSpace
{
 int m;
 void display(int n)
  {
 cout<<n;
  }
 }
```

Defining Namespace:

- In the previous example we have seen that, the variable **m** and the function **display** are inside the scope defined by the TestSpace namespace. Now, if we want to assign a value to m, we must use the scope resolution operator as:
- **TestSpace :: m = 100;** (here m is qualified using the namespace name)
- This approach becomes cumbersome if the members of a namespace are frequently used. In such cases, we can use a using directive to simplify their access. This can be done in two ways:
- ***using namespace namespace_name;*** // using directive
- ***using namespace_name :: member_name;*** // using declaration
- In the first form, all the members declared within the specified namespace may be accessed without using qualification. In the second form, we can access only the specified member in the program. ***Example:***

```
using namespace TestSpace;
m= 100;           // OK
display(200);     // OK
using TestSpace :: m;
m= 100;           // OK
display(200);     // Not OK, display not visible
```