DNYANSAGAR ARTS AND COMMERCE COLLEGE, BALEWADI,PUNE-45

2019 Pattern

Subject: Object Oriented Programming Using C++

sub code: CA 402

Class: SYBBA(CA) Sem-IV

By :Prof Santosh B. Potadar

www.dacc.edu.in

UNIT 1

Introduction to C++

C++ Object Oriented Programming (Oops)

HISTORY OF C++ :-

- C++ is an object oriented programming language.
- C++ was developed by “Bjarne Stroustrup” at AT & T’s (American Telecom and Telegraph) Bell Laboratories in USA 1980.
- C++ is an extension of C with major feature addition of the class construct feature.
- The most important feature that C++ adds to C are Classes, Inheritance, Function Overloading and Operator Overloading
- These features enables to create Abstract Data Type . Inherits properties from existing Data Type & support Polymorphism

FEATURES OF OBJECT ORIENTED PROGRAMMING :-

- Emphasis / Focus on data rather than procedure.
- Programs are subdivided into Objects.
- Data is hidden and can't be accessed by external functions.
- Object may communicate through each other with functions .
- New data and functions are easily added.
- It facilitate reusable code that can save lot of time.

1 C++ an overview

- **What is c++ ?**

  C++ is an object-oriented programming language.

  It was developed by Bjarne Stroustrup at AT&T Bell Laboratories in 1980's.

  C++ is an extension of C with a addition of class.

  C++ Design has been influenced from languages Simula67, Algol68, and SmallTalk.

  Initially C++ was called as 'C with classes'.

1 C++ an overview

C++ adds on to 'C' are :

i) classes
ii) inheritance
iii) function overloading
iv) operator overloading. &
v) polymorphism

C++ provides additional data types :

i) classes
ii) templates
iii) exceptions
iv) inline functions
v) operator overloading
vi) function name overloading
vii) references
viii) free storage management operators and additional library facilities.

1 C++ an overview

Object-oriented features in C++ which benefited for

- Build large programs with clarity and decreases complexity.
- Programs are easily maintainable and expandable.
- incorporating the efficiency of C.
- Programming approaches provides the techniques of modular programming,
- top-down programming, bottom-up programming and structured programming.

1 C++ an overview

- Run-time or memory overheads even when not used were avoided the design of C++.
- Type-checking and data-hiding features relay on compile-time analysis of programs to prevent accidental corruption of data.
- Numerical, scientific and engineering computation is done in C++.
- Heavily used in areas of graphics and user interfaces.

1 C++ an overview

Application area of C++

- local and wide-area networks.
- embedded systems.
- numerical computation
- graphics
- artificial intelligence
- user interaction
- data processing
- system programming

1 C++ an overview

Major release of C++

- 1980 C with classes
- 1983 Name C++ is given
- 1985 Version 1.0 (Basics)
- 1987 Standardization Process
- 1989 ANSI C++
- 1989 Version 2.0
  (Multi-inheritance, Operator Overloading)
- 1991 Version 3.0
  (Templates)
- 1995 Standard Template Library (STL),
  namespace, RTTI

1 C++ an overview

Program Features :

- C++ is a collection of function. Every C++ program must have a main().
- C++ is a Free-form language.
- The C++ statement terminates with semicolon.

Structure of C++ program :

Include files
Class declaration
Member functions definitions
Main function program

1 C++ an overview

Structure of C++ program :

- C++ program contains four sections.
- These sections may be placed in a separate code file and then compiled independently.
- It is more flexible to organize a program into three separate files.
- The class declarations are placed in a header file and the definition of member functions go into another file.
- This approach enables the programmer to separate the abstract specification of the interface (class definition) from implementation details (member function definition)
- The main program that uses the class is placed in third file in which ‘include’ the previous two files as well as any other files required.

UNIT 2

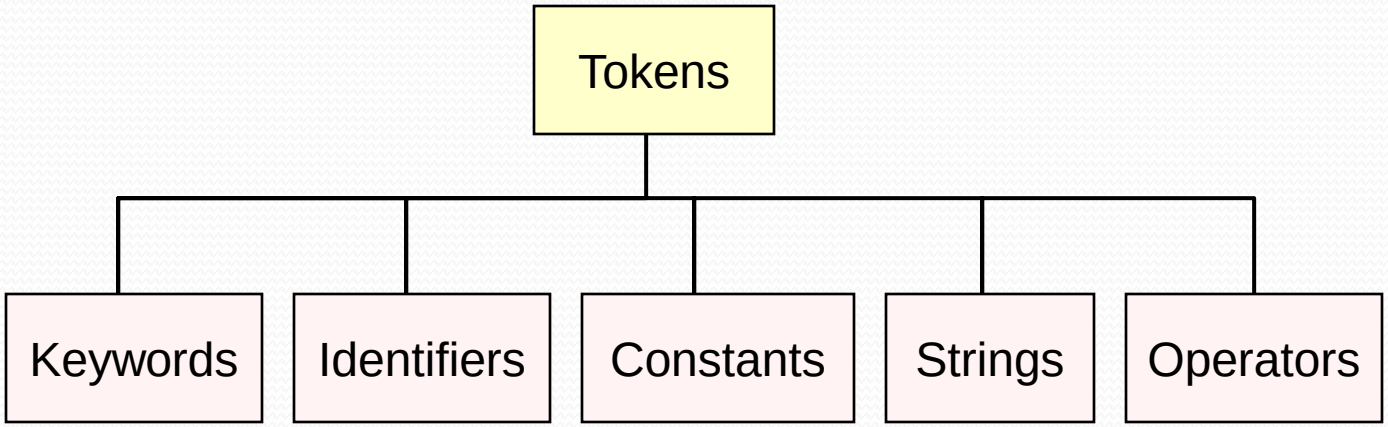
Beginning with C++

C++ an overview

Data types in C++

Data type	Size	Range
char	1 byte	-128 to 127
unsigned char	1 byte	0 to 255
int (short / signed)	2 bytes	-31768 to 32767
unsigned int	2 bytes	0 to 65535
long int	4 bytes	-2147483648 to 2147483647
long unsigned int	4 bytes	0 to 4294967295
float	4 bytes	3.4×10^{-38} to $1.7 \times 10^{+38}$
double	8 byte	1.7×10^{-308} to $1.7 \times 10^{+308}$

C++ an overview



1 C++ an overview

C++ Keywords :

Type	Keywords
Data Declaration keywords	char, float, int, double, long, short, signed, unsigned, enum, sizeof, union
Conditional keywords	case, else, if, switch,
Exception keywords	catch, try, throw, finally
Loop keywords	break, continue, do, for, while, goto
Modifier and Access keyword	new, delete, private, protected, public, static, void, volatile, const, auto
Miscellaneous keywords	return, this, operator, register, typedef
Structure keywords	class, default, struct, friend, inline, template, virtual,

C++ an overview

Identifiers :

Identifiers refers to names of data type, constant, variable, functions, arrays, class etc.

Valid identifiers	Invalid identifiers
area, volume, tax, interest, sum, a1, t1, b32, xy9 etc.	area of circle, rate of interest, x+y, student's, 2 xy, 231 etc.

C++ an overview

Constants :

Constants refer to fixed values that do not change during the execution of a program.

Primary constants	Secondary constants
integer	**Array**
float	**Pointer**
character	**Structure**
	Union
	Enum

123	decimal integer	“OOP”	string constant
12.34	floating point integer	‘A’	character constant
037	octal integer		
0x2	hexadecimal integer		

C++ an overview

Strings :

1. A sequence of characters is called string.
2. The string is a character array stored in contiguous memory locations.
3. Strings are used for manipulating words and sentences.

Ex. static char name[20]; static char str[10];
static char password[10]; static char line[20];

strlen	**Find length of string**
strupr	**Converts a string to uppercase**
strlwr	**Converts a string to lowercase**
strrev	**Reverses string**
strcpy	**Copies a string into another**
strcat	**Append one string at the end of another**
strcmp	**Compares two strings**

C++ an overview

Operators :

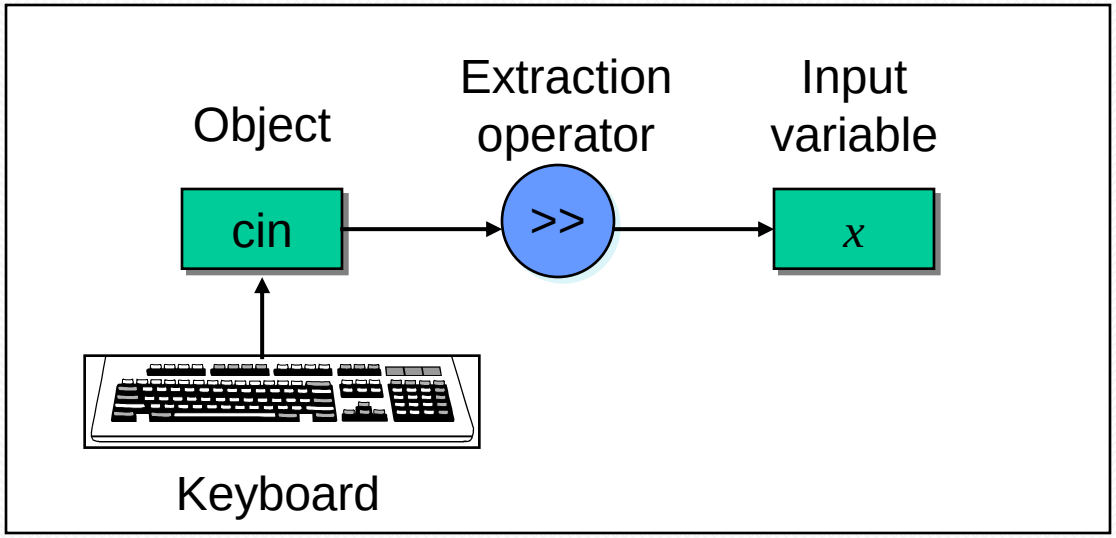
Operators	
Arithmetic	+, -, *, /, %
Relational	<, <=, >, >=, ==, !=
Logical	&&, ||, !
Assignment	=
Increment / decrement	++, --
Conditional	? :
Bitwise	&, |, ^, <<, >>, ~
Special	. ->

C++ Operators	
::	Scope Resolution operator
::*	Pointer-to-member declarator
->*	Pointer-to-member operator
.*	Pointer-to-member operator
delete	Memory release operator
endl	Line feed operator
new	Memory allocation operator
setw	Filed width operator

C++ an overview

Input statement :

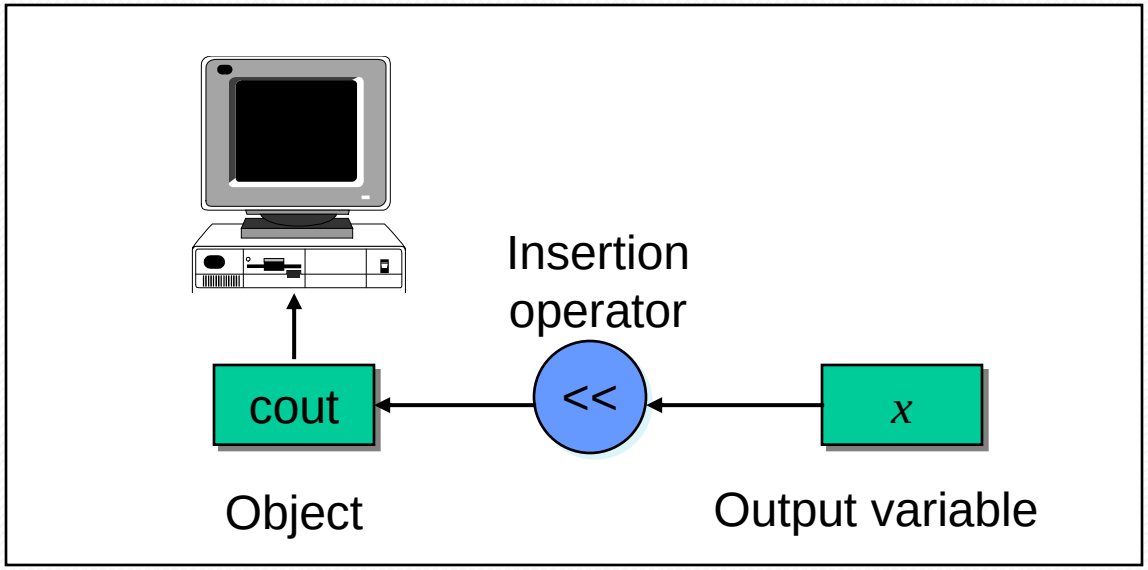
e.g. : cin>>x;



C++ an overview

Output statement :

e.g. : cout<<x;



UNIT 3

Class and Objects

OOP concepts

1. Object :

An object is a collection of properties, i.e. the collection of data members and member functions is called an object. An object can be a person, a place, or a thing with which the computer must deal.

2. Class :

Classes are user-defined data types and behave like the built-in types of a programming language.

Ex: *Class person*
Attributes: *Name, Age, Sex*
Operations: *Speak( ), Listen( ), walk( )*

3. Data Encapsulation :

Encapsulation is a mechanism that associates the code and the data it manipulates and keeps them safe from external interference and misuse. The use of encapsulation is protecting the members of a class from unauthorized access.

2 OOP concepts

4. Data abstraction :

- Creating new data types using encapsulated-items, that are well suited to an application to be programmed, is known as data abstraction.
- The data types created by the data abstraction process are known as Abstract Data Types (ADTs).
- Data abstraction is powerful technique, and its proper usage will result in optional, more readable, and flexible programs.
- The class declaration, allows encapsulation and creation of abstract data types.

5. Inheritance :

- Inheritance is the process, by which objects of one class acquire the properties of objects of another class.
- In OOP, the concept of inheritance provides the idea of reusability.

2 OOP concepts

6. Polymorphism :

- Polymorphism means the ability to take more than one form.
- The “move” operation, for example, behaves differently on the class person, and on the class polygon on the screen.

Different ways of achieving polymorphism in a C++ program :-

- Function Name Overloading
- Operator Overloading
- Dynamic Binding

7. Dynamic Binding :

- Binding refers to the linking of a procedure call to the code to be executed in response to the call.
- Dynamic binding means that the code associated with a given procedure call is not known until the time of the call at run-time.
- It is associated with polymorphism and inheritance.

2 OOP concepts

8. Message passing :

- OOP program consists of a set of objects that communicate with each other.
- The processing of programming involves following basic steps
  i) Creating objects that defines objects and their behaviour
  ii) Creating objects from class definitions.
- Establishing communication among objects.
- Objects communicate with one another by sending receiving information.
- A message for an object is a request for execution of procedure, and therefore
  will invoke a function (procedure) in receiving object that generates the results.
- Message passing involves specifying the name of object, the name of the function (message) and information to be sent.
- Objects have a life cycle.
- The can created and destroyed.
- Communication with an object is feasible as long as it is alive.

3 Functions in C++

A function is a block of instructions that is executed when it is called from some other point of the program.

The following is its format :

type name (argument1, *argument2*, ……..) *statement*

Where:

- type is the type of data returned by the function.
- name is the name by which it will be possible to call the function.
- arguments (as many as wanted can be specified). Each argument consists of a type of data followed by its identifier, like in a variable declaration (for example, int x) and which acts within the function like any other variable. They allow to pass parameters to the function when it is called.
- The different parameters are separated by commas.
- statement is the function's body. It can be a single instruction or a block of instructions.
  In the latter case it must be delimited by curly brackets { }.

Functions in C++

Recursion and linear function :

When a function calls itself it is called recursion.

```
void main()
{
        int num, ans;
        cin>>num;
        ans=fact (num);
        cout<<“Answer =”<<ans<<“\n”;
}
 int fact (int k)
{
        int prod;
        if (k<=1)
         return (1);
        else
        prod=k*fact(k-1);
        return (prod);
}
```

3 Functions in C++

An example using function

```
#include <iostream.h>
#include <conio.h>
void  fun( );                    // prototype declaration
void main( )
{
        clrscr( );
        cout<<"We are in main\n";
        fun( );                  // function call
        cout<<"Back in main\n";
        getch();
}
void  fun( )                     // function definition
{
         cout<<"We are in function\n";
}
```

```
O/p :   We are in main
        We are in function
        Back in main
```

3 Functions in C++

An example on Function

```
#include <iostream.h>
int addition (int a, int b)
{
        int r;
        r=a+b;
       return (r);
}
void main ()
{
       int z;
       z = addition (5,3);
       cout << "The result is " << z;
}
```

O/p : The result is 8

3 Functions in C++

Arguments passed *by value* and *by reference*:

Parameters passed to the functions have been passed by value.
This means that when calling to a function with parameters, what we have passed to the function were values but never the specified variables themselves.

For example : int x=5, y=3, z;
z = addition (x , y);

this case was to call function addition passing the values of x and y, that means 5 and 3 respectively, not the variables themselves.

```
int addition  (int a, int b)
                5 ↑     ↑ 3
z  =  addition ( x,     y );
```

3 Functions in C++

But there might be some cases where we need to manipulate from inside a function the value of an external variable.
For that purpose we have to use *arguments passed by reference*,

```
// passing parameters by reference
#include <iostream.h>
void duplicate (int &a, int &b, int &c)
{
  a*=2;
  b*=2;
  c*=2;
}
void main ()
{
  int x=1, y=3, z=7;
  duplicate (x, y, z);
  cout << "x=" << x << ", y=" << y << ", z=" << z;
}
```

O/p : x=2, y=6, z=14

3 Functions in C++

When passing a variable *by reference* we are passing the variable itself and any modification that we do to that parameter within the function will have effect in the passed variable outside it.

```
int duplicate  (int &a,   int &b,  int &c)
                  x         y       z
z  =  duplicate  (  x,        y,      z );
```

Variables a, b and c with associated with the parameters used when calling the function (x, y and z) and any change that we do on a within the function will affect the value of x outside.
Any change that we do on b will affect y, and the same with c and z.
without the *ampersand* (&) signs, it can not passed the variables *by reference*, but their values, and therefore, the output of program is the values of x, y and z without having been modified.
This type of declaration "by reference" using the ampersand (&) sign is exclusive of C++.

3 Functions in C++

Passing by reference is an effective way to allow a function to return more than one single value.
For example, the function that returns the previous and next numbers of the first parameter passed.

```
// more than one returning value
#include <iostream.h>
void prevnext (int x, int& prev, int& next)
{
    prev = x-1;
    next = x+1;
}
void main ()
{
     int x=100, y, z;
    prevnext (x, y, z);
    cout << "Previous=" << y << "Next=" << z;
}
```

```
O/p :    Previous=99
         Next=101
```

Functions in C++

An example using function with arguments

```
#include <iostream.h>
#include <conio.h>
int add (int, int);    // prototype declaration
void main( )
{
        int a, b, c;
        clrscr( );
        cout<<"Enter two numbers\n";
        cin>>a>>b;
        c=add (a,b);
        cout<<"Ans = "<<c<<"\n";
        getch();
}
int  add (int x, int y )
{
        int z;
        z = x + y;
        return (z);
}
```

```
O/p :
Enter two numbers
  10 20
Ans = 30
```

Functions in C++

An example using recursion

```
#include <iostream.h>
#include <conio.h>
void  fun( int );   // prototype declaration
void main( )
{
        int n;
        clrscr( );
        n = 5;
        fun(n);
        getch();
}
void  fun( int k)
{
        if ( k > 0)
        {
                cout<<k<<"\n";
                fun(k-1);
                cout<<k<<"\n";
        }
}
```

O/p :

```
5
4
3
2
1
1
2
3
4
5
```

3 Functions in C++

Function overloading :

Two different functions can have the same name if the prototype of their arguments are different.
In this case put the same name to more than one function if they have either a different number of arguments or different types in their arguments.
Using this concept, design a family of functions with one function name but with different argument lists.

Functions in C++

```
// overloaded function
#include <iostream.h>
int divide (int a, int b)
{
       return (a/b);
}
float divide (float a, float b)
{    return (a/b);
}
void main ()
{
    int x=5,y=2;
    float n=5.0,m=2.0;
    cout << divide (x,y);
   cout << "\n";  cout << divide (n,m);
   cout << "\n";
}
```

```
O/p :    2
         2.5
```

3 Functions in C++

Inline function :

```
inline Returntype  FunctionName(parameters)
{
        // body of main function
}
```

To eliminate the cost of calls to small function, C++ proposes a new function called as inline function.
A inline function is a function, that is expanded in a line when it is invoked.
When we prefix the keyword inline, the function becomes an inline function.
All inline functions must be defined before they are called.
Inline keyword just sends a request, not a command to the compiler. The compiler ma ignore this request if the function definition is too long too complicated and compiles the function as a normal function.
Inline expansion makes a program to run faster.
Inline expansion may not work with :
loop, a switch, or goto exists.
For functions not returning values, if a statement exists.
If function contains static variables.
If inline function are recursive.

3 Functions in C++

Prototype function :

```
type name ( argument_type1, argument_type2, ...);
```

It does not include a *statement* for the function.
That means that it does not include the body with all the instructions that are usually enclose within curly brackets { }.
It ends with a semicolon sign (;).
In the argument enumeration it is enough to put the type of each argument.
The inclusion of a name for each argument as in the definition of a standard function is optional, although recommendable.

3 Functions in C++

```
// prototyping
#include <iostream.h>
void odd (int a);
void even (int a);
void main ()
{
     int i;
     do
    {
         cout << "Type a number: (0 to exit)";
         cin >> i;
         odd (i);
    }    while (i!=0);
}
```

```
void odd (int a)
{
    if ((a%2)!=0)
    cout << "Number is odd.\n";
    else even (a);
}
void even (int a)
{
    if ((a%2)==0)
    cout << "Number is even.\n";
   else
    odd (a);
}
```

```
O/p :
Type a number (0 to exit): 9
Number is odd.
Type a number (0 to exit): 6
Number is even.
Type a number (0 to exit): 0
Number is even.
```

Classes & Objects

Class :

- A class is a way to bind the data and its associated functions together.
- It allows the data to be hidden, if necessary, from external use.

The general form of a class declaration is:

```
class   class_name
{
    private:
        Variable declarations;
        Function declarations;
    public:
        Variable declarations;
        Function declarations;
};
```

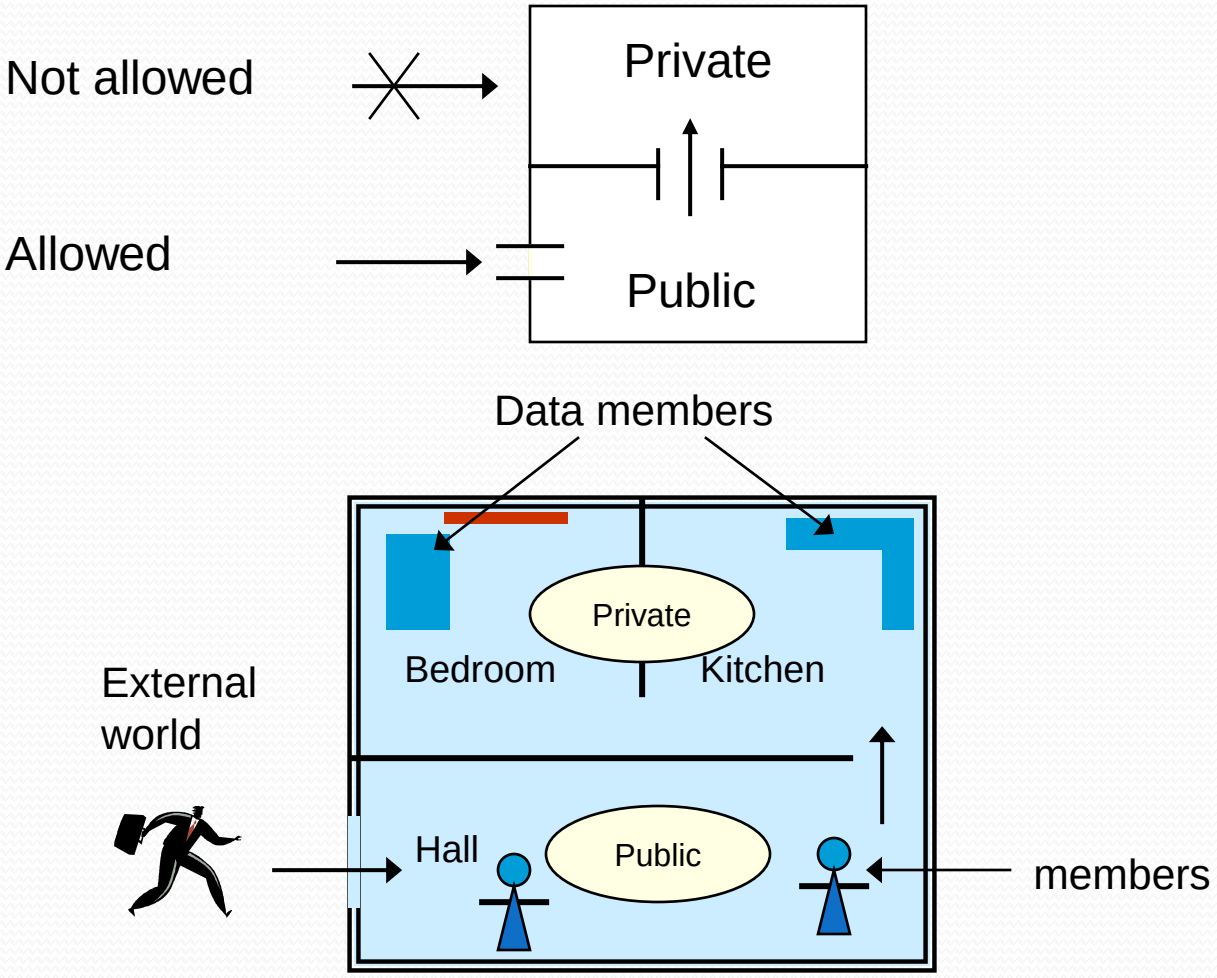
Classes & Objects

- The class declaration is similar to a *struct* declaration.
- The keyword *class* specifies that what follows is an abstract data of type *class_name.*
- The body of a class is enclosed within braces and terminated by a semicolon.
- The class body contains the declaration of variables and functions.
- These functions and variables are collectively called *members.*
- They are usually grouped under two sections, namely, *private* and *public* to denote which of the members are private and which of the are public.
- The keywords *private* and *public* are known as *visibility labels.*
- These keywords are followed by a colon.

Classes & Objects

- The members declared as private can be accessed only from within the class.
- public members can be accessed from outside the class also.
- The data hiding is the key feature of object-oriented programming.
- The use of the keyword *private* is optional.
- By default, the members of a class are *private.*
- If both the labels are missing, then, by default, all the members are private. Such a class is completely hidden from the outside world and does not serve any purpose.
- The variables declared inside the class are known as *data members* and the functions are known as *member's functions.*
- Only the member functions can have access to the private data members and private functions.
- However, the public members can be accessed from outside the class.

Classes & Objects



Classes & Objects

Ex: 1.

```
class student
{
        private:
                int roll_no;
                char name[20];
         public:
                void setdata(int rno, char *sn);
                void outdata();
};
```

Ex: 2.

```
class item
{
        private:
                int number;
                float cast;
        public:
                void getdata(int a, int b);
                void putdata();
};
```

Creating objects :

Syntax : *class ClassName ObjectName1, ObjectName2,…….;*

- The declaration of class does not define any objects but only specifies what they will contain.
- Once class has been declared we can create any member of its objects.
- In C++ variables are called as a class.
- Variables of class can be created by using the class name like any other data type variables.
- Objects can be created when the class is defined by placing their names immediately after the closing brace.

Ex: class student s1;
Or
student s1;

It creates the object *s1* of the class student. More than one object can be created with a single statement as follows:

Ex: class student s1, s2, s3, s4;
Or
student s1, s2, s3, s4;

OR :

```
class student
{
      ……..
      ……..
      ……..
} s1, s2, s3, s4;
```

Classes & Objects

Accessing class members :

Accessing class members is achieved by using the *member access operator, dot (.).*

The syntax for accessing members of a class is:

Accessing data members of class :

Syntax : *Objectname.Datamembers;*

Accessing member functions of a class :

Syntax : *ObjectName.Functionname (Actual arguments);*

Classes & Objects

Ex. : Member functions defined inside the body of the student class

```
#include <iostream.h>
#include <string.h>
class student
{
   private:
      int roll_no;
      char name[20];
   public:
      void setdata(int roll_no_in, char *name_in)
      {
          roll_no=roll_no_in;
          strcpy(name,name_in);
      }
      void outdata()
      {
         cout<<"Roll No="<<roll_no<<endl;
         cout<<"Name ="<<name<<endl;
      }
};
```

```
void main()
{
   student s1;
   student s2;
   s1.setdata(1,“Pallavi");
   s2.setdata(10,"Rajesh");
   cout<<"Student details....."<<endl;
   s1.outdata();
   s2.outdata();
}
```

Classes & Objects

- Defining member functions

The data members of a class must be declared within the body of the class, whereas the member functions of the class can be defined in any one of the following ways:

1) *Inside the class specification*
2) *Outside the class specification*

☐ Member functions inside the class body

member function declaration is similar to a normal function definition except that it is enclosed within the body of a class.

All the member functions defined within the body of a class are treated as inline function by default.

4 Classes & Objects

Ex. : date class which member functions defined inside a class

```
#include<iostream.h>
class date
{
   private:
       int day;
       int month;
       int year;
  public:
       void set(int dayin,int monthin,int yearin)
       {
                  day = dayin;
         month=monthin;
         year = yearin;
       }
       void show()
       {
      cout<<day<<"-"<<month<<"-"<<year<<endl;
       }
};
```

```
void main()
{
   date d1,d2,d3;
   d1.set(26,3,1958);
   d2.set(14,4,1971);
   d3.set(1,9,1973);
   cout<<"Birth Date of the 1st student:";
   d1.show();
   cout<<"Birth Date of the 2nd student:";
   d2.show();
   cout<<"Birth Date of the 3rd student:";
   d3.show();
}
```

4 Classes & Objects

member functions outside the class body

- Member functions that are declared inside a class have to be defined separately outside the class.
- Their definitions are very much like the normal functions.
- They should have a function header and a function body.

The general from of a member function definition is:

```
Return-type class-name :: function-name (argument declaration)
{
        function body;
}
```

Classes & Objects

```
#include <iostream.h>
class item
{
     private:
          int number;
          float cost;
    public:
          void getdata(int a, float b);
          void putdata();
};
void item :: getdata(int a, float b)
{
       number = a;
      cost = b;
}
void item :: putdata()
{
      cout<<"number :"<<number<<"\n";
      cout<<"cost:"<<cost<<"\n";
}
```

```
main()
{
     item x;
     x.getdata(100,299.95);
     x.putdata();
     item y;
     y.getdata(200,175.50);
     y.putdata();
}
```

Classes & Objects

- Static data member

➢ A data member of a class can be qualified as *static.*

➢ The properties of a static member variable are similar to that of a C static variable.

A static member variable has certain special characteristics:

- ❑ It is initialized to zero when the first object of its class is created.
- ❑ No other initialization is permitted.
- ❑ Only one copy of that member is created for the entire class and is shared by all the objects or that class, no matter how many objects are created.
- ❑ It is visible only within the class, but its lifetime is the entire program.

Static variables are normally used to maintain values common to the entire class.

Note: The type and scope of each “static” member variable must be defined outside the class definition.

Ex: int item:: count; //definition of static data member

Classes & Objects

- Passing Objects arguments :

It is possible to have functions, which accept objects of a class as arguments, just as there are functions, which accept other variables as arguments.

Like any other data type, an object can be passed as an argument to a function by the following ways:

Pass-by-value : a copy of the entire object is passed to the function.

Pass-by-reference : only the address of the object is passed implicitly to the function.

UNIT 4

Constructor and Destructor

Constructors and Destructors

Constructor :

```
class  classname
{
    private:
        private members;
    public:
       classname( );
}
classname : : classname()
{
                //constructor body definition
}
```

- Constructor :
  - Public function member
  - called when a new object is created (instantiated).
  - Initialize data members.
  - Same name as class
  - No return type
  - Several constructors

Constructor :

A constructor has the following characteristics:

- It has the same name as that of the class to which it belongs.
- It is executed automatically whenever the class is instantiated.
- It does not have any return type.
- It is normally used to initialize the data members of a class.
- It is also used to allocate resources such as memory, to the dynamic data members of a class.

Constructor :

Initialising class constructor

```
#include <iostream.h>
#include <conio.h>
class Time
{
        private :
                int hour;
                int min;
         public :
        Time ( )
        {
                hour = min = 0;
        }
        Time ( int h, int m)
        {
                hour = h;
                min = m;
        }
 void showTime()
{
        cout<<hour<<" : "<<min<<"\n";
}
};
```

```
void main( )
{
    Time t1;
     t1.showTime( );
    Time t2(11,30);
     t2.showTime();
}
```

```
O/p :     0 : 0
          11 : 30
```

Destructor :

- Destructors
  - Special member function
  - Same name as class
    - Preceded with tilde (~)
  - No arguments
  - No return value
  - Cannot be overloaded
  - Before system reclaims object's memory
    - Reuse memory for new objects
    - Mainly used to de-allocate dynamic memory locations

```
Class calssname
{
    private:
        ……
    public:
      ~classname( );      //destructor prototype
};
classname : : ~classname( )
{
                Destructor body definition
}
```

Destructor :

An example for destructor

```
#include <iostream.h>
#include <conio.h>
class Test
{
        private :
                static int count;
        public :
        Test ( )
        {
        cout<<"Object created ="
                <<++count<<"\n";
        }
        ~ Test( )
        {
        cout<<"Object destroyed ="
        <<count--<<"\n";
        }
};
   int Test :: count;
```

```
void main( )
{
  Test t1, t2, t3;
 {
   Test t4, t5;
   cout<<"Leaving block…\n";
  }
  cout<<"Leaving program…\n";
}
```

O/p :

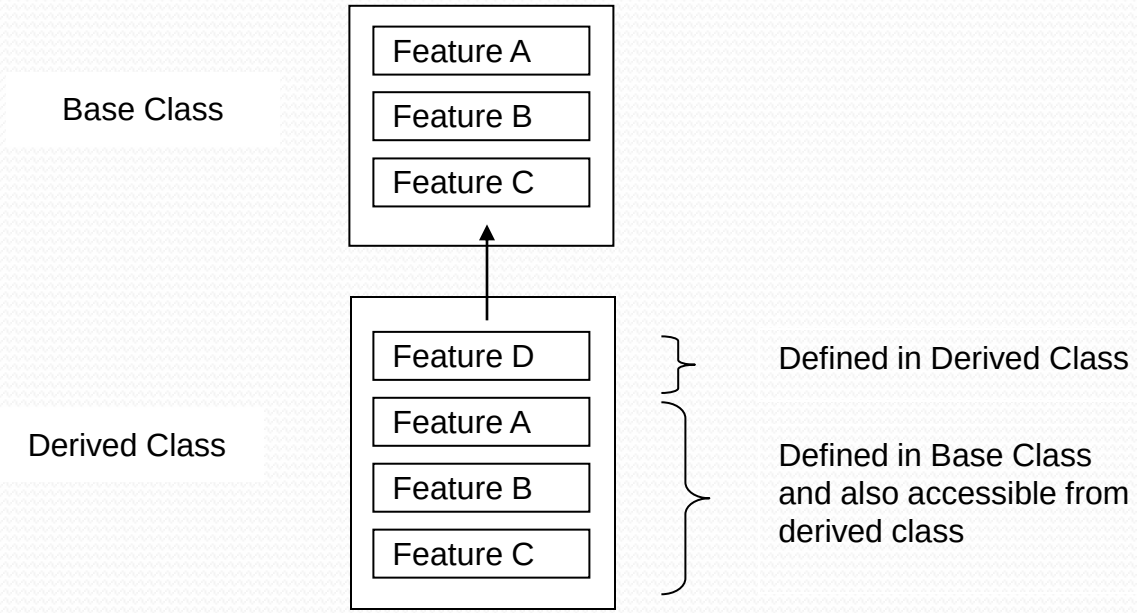
```
Object created = 1
Object created = 2
Object created = 3
Object created = 4
Object created = 5
Leaving block…
Object destroyed = 5
Object destroyed = 4
Leaving program…
Object destroyed = 3
Object destroyed = 2
Object destroyed = 1
```

UNIT 5

Inheritance

Inheritance

- Inheritance is a technique of organizing information in a hierarchical form.
- It is like a child inheriting the features of its parents.
- The technique of building new classes from the existing classes is called inheritance.



Inheritance

- Inheritance, a prime feature of oops can be stated as the process of creating new classes (called derived classes), from the existing classes (called base classes).
- The derived class inherits all the capabilities of the base class and can add refinements and extensions of its own.
- The base class remains unchanged.
- A base class is often called the ancestor, parent, or superclass, and a derived class is called the descendent, child or subclass.

The derivation of a new class from the existing class is represented in the above figure.

The derived class inherits the features of the base class (A, B, and C) and adds its own features (D). The arrow in the diagram symbolizes derived from. Its direction from the derived class towards the base class, represents that the derived class accesses features of the base class and not vice versa.

Inheritance

Derived base class :

```
Class Derivedclass : [VisibilityMode] BaseClass
{
        // members of derived class
        // and they can access members of the base class
}
```

- A derived class extends its features by inheriting the properties of another class, called base class and adding features of its own.
- The declaration of derived class specifies its relationship with the base class in addition to its own features.

Ex: 1.
```
class D : public B      // public derivation
{
        // members of D
};
```

Ex: 2.
```
class D : private B     // private derivation
{
        // members of D
};
```

Ex: 3.
```
class D : B             // private derivation by default
{
    // members of D
};
```

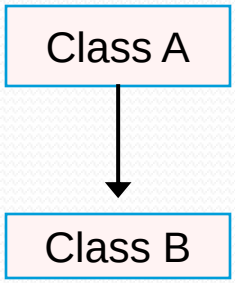
Inheritance

Private, public protected :

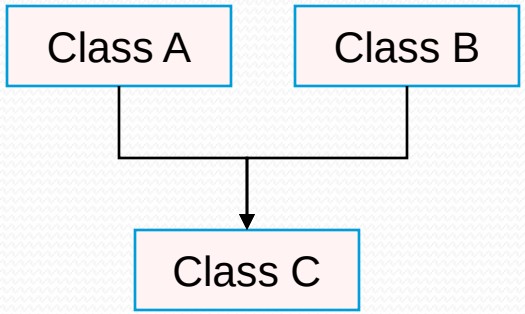
- Inheritance of a base class with visibility mode public, by a derived class, causes public members of the base class to become public members of the derived class
- the protected members of the base class become protected members of the derived class.
- Inheritance of a base class with visibility mode private by a derived class, causes public members of the base class to become private members of the derived class
- the protected members of the base class become private members of the derived class.
- Member functions and objects of a derived class can treat these derived members as through they are defined in the derived class with the private modifier.
- Thus objects of derived class cannot access these members.

Inheritance

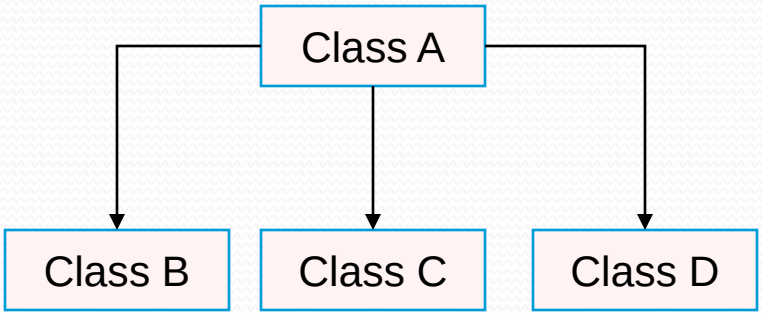
Various form of inheritance :



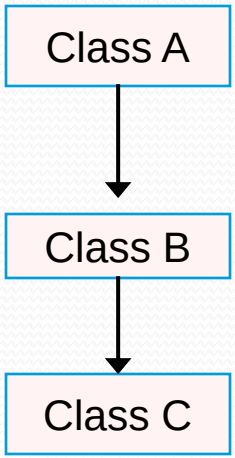
(a) Single Inheritance



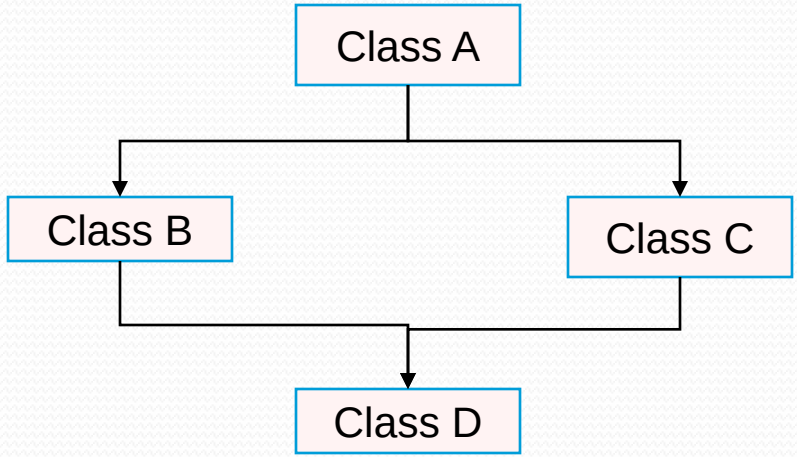
(b) Multiple inheritance



(c) Hierarchical inheritance



(d) Multilevel Inheritance



(e) Hybrid Inheritance

7 Inheritance

Examples on inheritance : Single level

```
#include <iostream.h>
#include <conio.h>
class Data
{
protected :
        int num1;
        int num2;
public :
void getData( int x, int y)
{
        num1 = x;
        num2 = y;
}
};
```

```
#include <iostream.h>
#include <conio.h>
class Data
{
protected :
        int num1;
        int num2;
public :
void getData( int x, int y)
{
        num1 = x;
        num2 = y;
}
};
```

O/p :
Num1 = 10
Num2 = 20
Sum = 30

7 Inheritance

Examples on inheritance :

```
#include <iostream.h>
#include <string.h>
#include <conio.h>
class Person
{
 protected :
        char name[30];
        int age;
 public :
void getPersonData( char *n, int a)
{
        strcpy(name,n);
        age = a;
}
};
```

```
class Typist
{
protected :
        int speed;
public :
void getTypingSpeed( int s)
{
        speed = s;
}
};
class Staff : public Person, public Typist
{
private :
        char department[30];
public :
void getDepartment( char * dpt)
{
        strcpy(department, dpt)
}
```

7 Inheritance

Examples on inheritance :

```
void showDetails( )
{
        cout<<"Name  :"<<name<<"\n";
        cout<<"Age     :"<<age<<"\n";
        cout<<"Speed :"<<speed<<"\n";
        cout<<"Department :"<<department<<"\n";
}
};
void main( )
{
        Staff myStaff;
        myStaff.getPersonData("Ramesh", 32);
        myStaff.getSpeed(60);
        myStaff.getDepartment("Library");
        myStaff.showDetails( );
}
```

```
O/p :   Name : Ramesh
        Age  : 32
        Speed : 60
        Department : Library
```

7 Inheritance

Examples on inheritance :

```
#include <iostream.h>
#include <string.h>
#include <conio.h>
class Person
{
    protected :
      char name[30];
      int age;
    public :
void getPersonData( char *n, int a)
{
      strcpy(name,n);
      age = a;
}
};
```

```
class Manager : public Person
{
protected :
char department[30];
public :
void getDepartment( char * dpt)
{
      strcpy(department, dpt)
}
void showDetails( )
{
cout<<"Manager details :\n";
cout<<"Name  :"<<name<<"\n";
cout<<"Age     :"<<age<<"\n";
cout<<"Department
      :"<<department<<"\n";
}
};
```

7 Inheritance

```
class Scientist : public Person
{
private :
int publications;
public :
void getPublications( int pub)
{
      publications = pub;
}
void showDetails( )
{
cout<<"Scientist details :\n";
cout<<"Name  :"<<name<<"\n";
cout<<"Age     :"<<age<<"\n";
cout<<"Publications
      :"<<publications<<"\n";
}
};
```

```
void main( )
{
      Manager m1;
      Scientist s1;
      m1.getPersonData("Kumar", 36);
      m1.getDepartment("Sales");
      s1.getPersonData("Krishna", 42);
      s1.getPublications(23);
      m1.showDetails( );
      s1.showDetails( );
}
```

O/p :
```
Manager details :
Name : Kumar
Age   : 36
Department : Sales
Scientist details :
Name : Krishna
Age  : 42
Publications : 23
```

Polymorphism

Polymorphism

- Polymorphism refers to identically named methods (member functions) that have different behaviour depending on the type of object they refer.
- Polymorphism plays an important role in allowing objects having different internal structure to share the same external interface.
- Polymorphism extensively used in implementing inheritance.

Types of polymorphism :

- Polymorphism means ability to take more than one form. For examples an operation may exhibit different behaviour in different instances.
- The concepts of polymorphism is implemented by using the overloaded functions and operators.
- In specific cases appropriate member function is selected while program is running. This is called as a runtime polymorphism.
- Virtual functions are used to runtime polymorphism.

Polymorphism

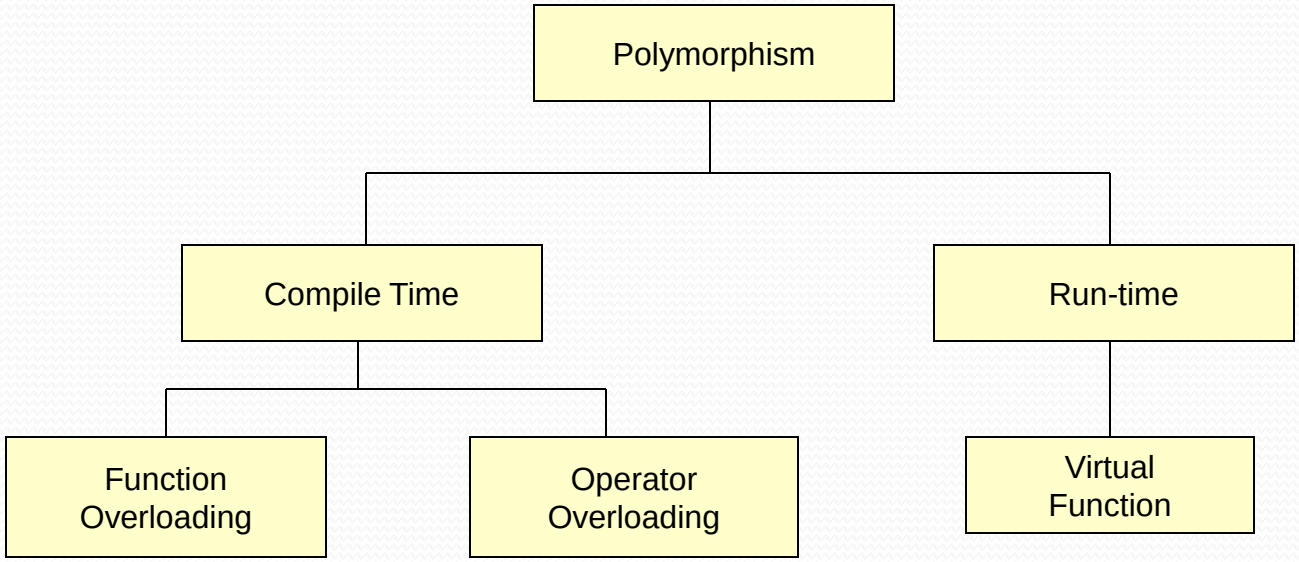


Fig : Achieving polymorphism

Polymorphism

Virtual Function :

When virtual functions are created for implementing late binding, following are some basic rules that should be followed :

1. The virtual functions must be members of some class.
2. They cannot be static members.
3. They are accessed by using object pointers.
4. We can't have virtual friend functions.
5. A virtual function in a base class must be defined, even though it may not be used.
6. 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 prototype, C++ considers them as overloaded functions, and the virtual function mechanism is ignored.
7. We cannot have virtual constructors, but we can have virtual destructors.
8. While a base pointer can point to any type of the derived object, the reverse is not true.
9. 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.
10. 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.

Polymorphism

```
#include <iostream.h>
#include <conio.h>
class Shape
{
   public :
    virtual void display()       // virtual function
   {  }
};
class Circle : public Shape
{
   float rad;
    public :
    void getrad(int r)
    {    rad = r;
    }
    void display( )
    {
        cout<<“Area of circle = “<<3.14*rad*rad<<“\n”;
     }
};
```

```
class Triangle : public Shape
{
float base;
float height;
public :
void getbh(float b, float h)
{ base  =    b;
height  =    h;
}
void display( )
{
cout<<“Area of triangle =“<<0.5*base *height<<“\n”;
}
};
```

Polymorphism

```
void main()
{
   Circle MyCir;
   Triangle MyTri;
   MyCir.getrad(7);
   MyTri.getbh(9,11);
   Shape *sptr;
   sptr=&MyCir;
   sptr->display();  // display area of circle
   sptr=&MyTri;
   sptr->display();  // display area of triangle
}
```

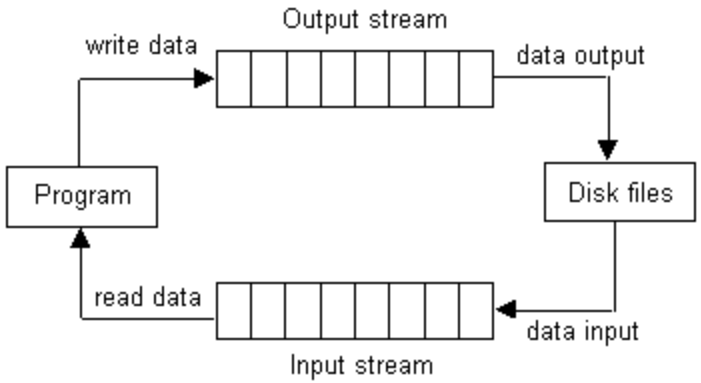
Polymorphism

```
void main()
{
	AddData MyData1(10,30);
	SubData MyData2(40,20);
	Data *Dptr;
	Dptr = &MyData1;
	Dptr->display();
	Dptr = &MyData2;
	Dptr->display();
}
```

Managing Console Operations

9 I/O Streams

- C++ uses the concept of stream and steam class to implement I/O operations with the console, disk files and terminals.
- Each device is very different, the I/O stream supplies an interface to the programmer that is independent of the actual device being accessed.
- A stream is a sequence of bytes. It acts as a source from which the input data can be obtained or as a destination to which the output data can be sent.
- The source of stream that provides data to the program is called the input stream and destination stream that receives output from the program is called the output stream.
- The program extracts the bytes from an input stream and inserts bytes into an output stream.
- Data in input stream can come from the keyword or any other storage device.
- Data in out stream can go to the screen or any other storage device.

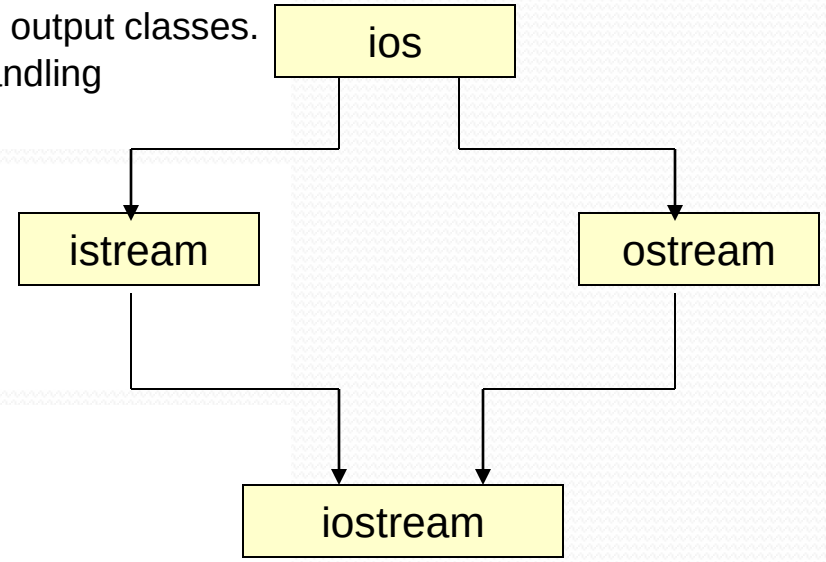


File input and output streams

9 I/O Streams

1. ios:

   Contains basic facilities that are used by all other input and output classes.
   Declares constants and functions that are necessary for handling formatted input and output operations.



2. Istream :

   Inherits the properties of ios.
   Declares input functions such as get(), getline() and read()
   Contains overloaded extraction operator >>

3. ostream :

   Inherits the properties of ios.
   Declares output functions such as put() and write().
   Contains overloaded insertion operator <<.

4. iostream :

   Inherits the properties of ios istream and ostream through multiple inheritance and thus contains all the input and output functions.

I/O Streams

Ex : character io with get() and put()

```
#include <iostream.h>
void main()
{
    int count=0;
    char c;
    cout<<”Input text\n”;
    cin.get(c);
    While (c != ‘\n’)
    {
        cout.put(c);
        count++;
        cin.get(c);
    }
      cout<<”\n number of characters =”<<count<<”\n”;
}
```

Template

10

Templates

- Template is a method for writing a single function or class for a family of similar functions or classes in a generic manner.
- When a single function is written for a family of similar functions, it is called as 'function template'.
- The main advantage of using function template is avoiding unnecessary repetition of the source code.

```
template<class T>
T function_name( T formal arguments)
{
  ---------
  ---------
  return(T);
}
```

10 Templates

Ex : Function template for summing an array

```
#include <iostream.h>
template<class T>
T sum(T array[ ],int n)
{
  T temp=0;
  for(int i=0;i<=n-1;++i)
      temp=temp+array[i];
  return(temp);
}
```

```
void main()
{
    int n=3,sum1=0; float sum2=0.0;
    static int a[3]={1,2,3};
    static float b[3]={1.1,2.2,3.3};
    sum1=sum(a,n);
    cout<<"sum of the  integers="<<sum1<<endl;
     sum2=sum(b,n);
     cout<<"sum of the floating point numbers="<<sum2;
    cout<<endl;
}
```