Unit 5:

ESSENTIALS OF OBJECT ORIENTED PROGRAMMING:

Outline

1. Operator overloading
2. Inheritance
    1. Single
    2. Multiple
    3. Multilevel
    4. Hybrid
3. Pointers to Objects
4. Assignment of an Object to another Object
5. Polymorphism through dynamic binding
6. Virtual Functions
7. Overloading, overriding and hiding
8. Error Handling

Topic -1:

OPERATOR OVERLOADING

Operator Overloading

- **Why is Operator Overloading?**
  - Readable code
  - Extension of language to include user-defined types
    - I.e., classes
  - Make operators sensitive to context
  - Generalization of function overloading

Operator Overloading

- **Why is Operator Overloading?**
  - Examine what operators make sense for a "new data type" we are creating and implement those that make sense as operators:
  - Examples:
    - input_data() function can be replaced by >>
    - Display() function can be replaced by <<
    - Assignment() function or copy() function can be replaced by =

Operator Overloading

- **What is Operator Overloading?**
  - Allows us to define the behavior of operators when applied to objects of a class, it enabling C++'s operators to work with class objects
  - Using traditional operators with user-defined data types (objects)
- Operators may not look like functions but can hide function invocations.
- You cannot change the precedence or associativity of operators when uses with primitive data type

- **What is Operator Overloading?**

Example:

1. Obj1.add(Obj2); // Invoking function for additions
2. Obj1 + Obj2; // used operator function for addition
3. Obj3 = Obj1 + Obj2; //used operator function for addition and returning result

Operator Overloading

- Operator Overloading does not allow us to alter the meaning of operators when applied to built-in types
  - one of the operands must be an object of a class
- Operator Overloading does not allow us to define new operator symbols
  - we overload those provided for in the language to have meaning for a new type of data...and there are very specific rules!

Operator Overloading

- It is similar to overloading functions
  - except the function name is replaced by the keyword operator followed by the operator's symbol
  - the return type represents the type of the residual value resulting from the operation
  - the arguments represent 1 or 2 operands expected by the operator

Expression has two types of values:

- lvalue: (locator value) represents an object that occupies some identifiable location in memory (i.e. has an address).
- rvalue: (residual value)is an expression that does not represent an object occupying some identifiable location in memory.

Operator Overloading

- Requires great care
  - **When overloading misused, program will be difficult to understand**
- We cannot change the....
  - number of operands an operator expects
  - precedence and associativity of operators
  - or use default arguments with operators

- We should not change...
  - the meaning of the operator
    - (+ does not mean subtraction)
  - the nature of the operator (3+4 == 4+3)
  - the data types and residual value expected
  - provide consistent definitions (if + is overloaded, then += should also be)

Operator Overloading:
Understanding the Syntax : It's a member function

```
returnType operator*(parameters);
```

any type *keyword* *operator symbol*

Return type may be whatever the operator returns

- Including a reference to the object of the operand

Operator symbol may be any *overloadable* operator from the list.

❖ Unary Operator

```
returnType operator ++ () {      }
```

Or

```
returnType operator ++ (class_name object) {          }
```

❖ Binary Operator

```
returnType operator + (class_name object) {          }
```

Or

```
returnType operator + (class_name object1, class_name object2)
{

}
```

Example:

```
class complex {
      double real, imag;

public:

complex() { }
complex(double r, double i)
{ real = r ; imag = i;}

complex operator+(complex a);
complex operator*(complex a);

void printComplex()
{

 cout<<real<<"+I" <<imag;

}

};

complex complex :: operator + (complex c)

{       complex c1;

        c1.real = real + c.real;

        c1.imag = imag + c.imag;

        return c1; }
```

```
complex complex :: operator * (complex c)
{     complex c1;
      c1.real = real * c.real + imag * c.imag;
      c1.imag = real * c.imag + imag * c.imag;
      return c1;
}
```

```
// invoking operator functions like mathematical expression //
int main()
{
complex a (2.3, 3.4);
complex b (2.8, 3.6);
complex c, d;

c = a + b;  //invoking + operator function
c.printComplex();

d = c * b;  //invoking * operator function

d.printComplex();
}
```

Operator Overloading

- An overloaded operator's operands are defined the same as arguments are defined for functions.
- The arguments represent the operator's operands.
- Unary operators have a single argument and binary operators have two arguments.
- When an operator is used, the operands become the actual arguments of the "function call".
- Therefore, the formal arguments must match the data type(s) expected as operands or a conversion to those types must exist.
- It is recommended that unary operators always be overloaded as members, since the first argument must be an object of a class

Operator Overloading

- The return type of overloaded operators is also defined the same as it is for overloaded functions.
- The value returned from an overloaded operator is the residual value of the expression containing that operator and its operands.
- It is extremely important that we pay close attention to the type and value returned.
- It is the returned value that allows an operator to be used within a larger expression.
- It allows the result of some operation to become the operand for another operator.
- A return type of void would render an operator useless when used within an expression.

Operator Overloading

- Binary operators have either a single argument if they are overloaded as members (the first operand corresponds to the implicit object and is therefore an object of the class in which it is defined)
- Or, binary operators have two operands if they are overloaded as non-members
  - (where there is no implicit first operand)
- In this latter case, it is typical to declare the operators as friends of the class(es) they apply to -- so that they can have access privileges to the private/protected data members without going through the public client interface.

Operator Overloading

- All arithmetic, bitwise, relational, equality, logical, and assignment operators can be overloaded.
- In addition, the address-of, dereference, increment, decrement, and comma operators can be overloaded.

- Operators that cannot be overloaded include:

::	scope resolution operator
.	direct member access operator
?:	conditional operator
sizeof	size of object operator

- Operators that must be overloaded as members:

=	assignment operator
[]	subscript operator
()	function call operator
->	indirect member access operator

Binary relational operators

```
class Age
{
int a;
public:
Age()
{ cout<<"Enter age of the person"<<endl;
  cin>>a;   }

int operator<(Age p)
{   int x;
    x = (a < p.a);
    return x;   }

int operator>(Age p)
{   int x;
    x = (a > p.a);
    return x;   }

int operator==(Age p)
{   int x;
    x = (a == p.a);
    return x;   }
};
```

```
int main()
{
Age P1;
Age P2;
if(P1<P2)
cout<<"P1 is less than P2"<<endl;
else
cout<<"P1 is greater than P2"<<endl;
}
```

Unary operator overloading (++, -- , -)

```
class Unaryop  //pre fixed
{
int a, b, c;
public: Unaryop() { }
Unaryop (int a1,int b1, int c1)
{ a=a1; b=b1; c=c1; }
void operator ++ ()
{ a++; b++; c++; }
void display()
{ cout<<a<<"  "<<b<<"
"<<c<<endl; }
};

int main()
{
Unaryop u(2,3,4);
++u;
u.display();
}
```

```
class Unaryop  //post fixed
{
int a, b, c;
public:  Unaryop() { }
Unaryop (int a1,int b1, int c1)
{ a=a1; b=b1; c=c1; }
void operator ++ (int)
{ a++; b++; c++; }
void display()
{ cout<<a<<"  "<<b<<"
"<<c<<endl; }
};

int main()
{
Unaryop u(2,3,4);
u++;
u.display();
}
```

```
int main()
{
Unaryop U1(10,20,30);
Unaryop U2;
U2=U1++;
U1.display();
U2.display();
}
```

```
int main()
{
Unaryop U1(10,20,30);
Unaryop U2;
U2=++U1;
U1.display();
U2.display();
}
```

As Non-members – friend function

- Overloading operators as non-member functions is like defining regular C++ functions.
- Since they are not part of a class' definition, they can only access the public members. Because of this, non-member overloaded operators are often declared to be friends of the class.
- When we overload operators as non-member functions, all operands must be explicitly specified as formal arguments.
- For binary operators, either the first or the second must be an object of a class; the other operand can be any type.

Example: using friend function

```
class complex {
      double real, imag;

public:

complex() { }
complex(double r, double i)
{ real = r ; imag = i;}

friend complex operator+(complex a);
friend complex operator*(complex a);

void printComplex()
{

 cout<<real<<"+I" <<imag;

}

};

complex operator +(complex c1, complex c2)

{       complex c3;

        c3.real = c1.real + c2.real;

        c3.imag = c1.imag + c2.imag;

        return c3; }
```

```
complex operator * (complex c1, complex c2)
{     complex c3;
      c1.real = c1.real * c2.real + c1.imag *
c2.imag;
      c1.imag = c1.real * c2.imag + c1.imag *
c2.imag;
      return c3;
}
```

```
int main()
{
complex a (2.3, 3.4);
complex b (2.8, 3.6);
complex c, d;

c = a + b;  //invoking + operator function
c.printComplex();

d = c * b;  //invoking * operator function

d.printComplex();
}
```

Real use of friend function in Operator Overloading

1. Assume we have a Vector (sequence of value) : 2, 4, 5, 6, 7, 3, 4, 2, 6, 10.
2. Multiply vector by scalar value: 10.
3. Result of multiplication will be: 20, 40, 50, 60, 70, 30, 40, 20, 60, 100.

Member operator function to multiply

Example: with member function

```
class Vector
{
int a[10],n;
public:
Vector() {}

Vector(int n1) {
n=n1;
for(int i=0;i<n;i++)
cin>>a[i]; }

Vector operator *(int x)
{
Vector V;
for(int i=0;i<n;i++)
V.a[i]=a[i]*x;
V.n=n;     return V;
}
```

```
void print() {
for(int i=0;i<n;i++)
cout<<a[i]<<endl;  }

};



int main()
{
Vector V1;
cout<<"input the values of vector"<<endl;
Vector V2(3);
V1 = V2 * 10;
cout<<endl<<"vector values after multiplication"<<endl;
V1.print();
}
```

Using friend function

```
class Vector
{
int a[10],n;
public:

Vector() {}

Vector(int n1) {
n=n1;
for(int i=0;i<n;i++)
cin>>a[i]; }

friend Vector operator *(int x, Vector V1);

void print() {
for(int i=0;i<n;i++)
cout<<a[i]<<endl;  }
};
```

```
Vector operator *(int x, Vector V1)
{
Vector V2;
for(int i=0;i<V1.n;i++)
V2.a[i]=V1.a[i]*x;
V2.n=V1.n;
return V2;
}

int main()
{
Vector V1;
cout<<"input the values of vector"<<endl;
Vector V2(3);
V1 = 10 * V2;
cout<<endl<<"vector values after multiplication"<<endl;
V1.print();
}
```

Difference between member and friend function with respect to operator overloading

Member Function	Friend Function
Number of parameters to be passed is reduced by one, as the calling object is implicitly supplied as an operand.	Number of parameters to be passed is more.
Unary operators takes no explicit parameters.	Unary operators takes one explicit parameter.
Binary operators takes only one explicit parameter.	Binary operators takes two explicit parameters.
Left-hand operand has to be the calling object.	Left-hand operand need not be an object of the class.
Writing Obj2 = Obj1 + 10 is allowed but Obj2 = 10 + Obj1 is not allowed	Writing either Obj2 = Obj + 10 or Obj2 = 10 + Obj1 is allowed.

Example: Assignment operator overloading

```
class Student
{
char name[40];
int rn;
char div;
char prog[20];
char dept[20];
public:
void input1()
{
cout<<"Enter student details name, roll no:
"<<endl;
cin>>name>>rn;              }
void input2()
{
cout<<"Enter student details div, prog and dept:
"<<endl;
cin>>div>>prog>>dept;     }
```

```
void print()
{
cout<<“Name of student is
:”<<name<<endl;
cout<<“Roll no is :”<<rn<<endl;
cout<<“Division is: ”<<div<<endl;
cout<<“Program is
:”<<prog<<endl;
cout<<Department is
:“<<dept<<endl;
}
void operator = (Student s)
{
div = s.div;
strcpy(prog,s.prog);
strcpy(dept,s.dept);
}
};
```

```
int main()
{
Student s[10];
s[0].input1();
s[0].input2();
for(int i=1;i<5;i++)
{
s[i].input1();
s[i]=s[0];
}
for(int i=0;i<5;i++)
{
s[i].print();
}
}
```

Guidelines to use operator overloading:

- Determine if any of the class operations should be implemented as overloaded operators: does an operator exists that performs behavior similar in nature to our operations?
- If so, consider overloading those operators. If not, use member functions.
- Consider what data types are allowed as operands.
- what conversions can be applied to the operands.
- whether or not the operands are modified by the operation that takes place.
- what data type is returned as the residual value
- whether the residual value is an rvalue (an object returned by value)

Guidelines to use operator overloading:

- If the first operand is not an object of the class in all usages:(e.g., +)
  - overload it as a friend non-member function
- If the first operand is always an object of the class: (+=)
  - overload it as a member

Overloading Input / Output operators

- Extraction operator ">>" – to perform input operation
- Insertion operator "<<" – to perform output operation
- They are used with objects to perform I/O operations:
  - Example : cin >> a;
    cout << a;

Where,

- a is variable of built in data type.
- "cin" is the object of predefined class "istream" and
- "cout" is the objects of the predefined class "ostream"

Both these classes are located in "iostream.h" header file.

Conclusion: If we wish to use I/O operators directly with the objects to perform input / output then we need to write overloading function explicitly in the program.

Continued…

Need non member function -“friend”- Why?

- If an operator is overloaded as member function, then it must be a member of the object on left side of the operator.
- Example 1:- consider the statement “obj1 + obj2”, In this case both the object need to be of same class.
- In the Example-1, obj1 is implicit object and used to member operator function and obj2 is explicit object passed as an argument to member operator function
- Example 2:- Now consider the statement “cin >> obj1”, in this case first object is of different class “istream” and obj1 is of user defined class.
- In the Example 2:- since “cin” object is of different class so it can not be implicitly used as well as obj1 is of different class so need to be passed as an argument.

Conclusion: When we are trying to access objects of two different classes together then non member function (friend function) is the best choice.

Example 1: Input / Output complex number

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

class Complex
{
private:
   float real, imag;
public:
   friend ostream & operator << (ostream &output, Complex &c);
   friend istream & operator >> (istream &input,  Complex &c);
};
```

```
ostream & operator << (ostream &output, Complex &c)
{
   output << c.real;
   output << "+i" << c.imag << endl;
   return out;
}

istream & operator >> (istream &input,  Complex &c)
{
   cout << "Enter Real Part ";
   input >> c.real;
   cout << "Enter Imaginary Part ";
   input >> c.imag;
   return in;
}
```

```
int main()
{
   Complex c1;
   cin >> c1;
   cout << "The complex object is ";
   cout << c1;
   return 0;
}
```

Example 2: I/O of vector data and multiplication - Using friend function

```
class Vector
{
int a[10],n;
public:
friend istream & operator >> (istream &input,  Vector &c);
friend ostream & operator << (ostream &output, Vector &c);
friend Vector operator *(int x, Vector V1);
};
```

```
Vector operator *(int x, Vector V1)
{
Vector V2;
for(int i=0;i<V1.n;i++)
V2.a[i]=V1.a[i]*x;
V2.n=V1.n;
return V2;
}
```

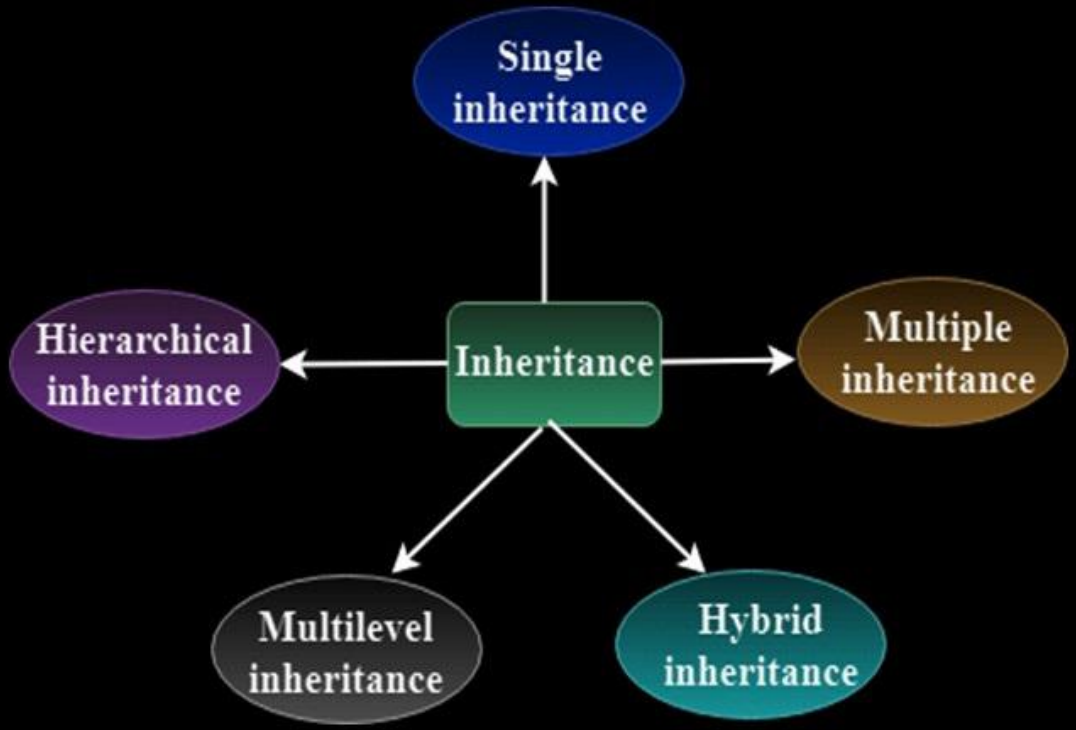
```
istream & operator >> (istream &input,  Vector &v)
{
        input >> v.n;
        for(int i = 0; i<v.n; i++)
        input >> v.a[i];
        return input;       }
ostream & operator << (ostream &output, Vector &v)
{
        for(int i = 0; i<v.n; i++)
        output << v.a[i];
        return output;      }
```

```
int main()
{
Vector V1, V2;
cout<<“Input the size and values of vector"<<endl;
cin >> V1;
V2 = 10 * V1;
cout<<endl<<"vector values after multiplication"<<endl;
cout << V2;
}
```

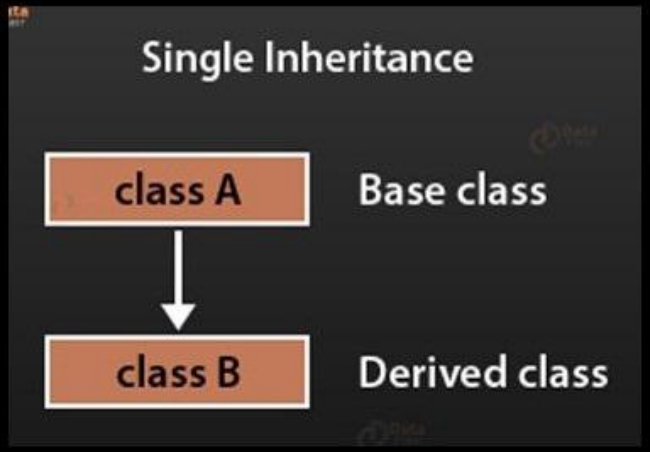
Topic -2:

INHERITANCE

Types of Inheritance:



Single Inheritance




```
#include <iostream>

class BaseClass {
Data members;
public:
Members functions;
};

class DerivedClass : public BaseClass
{
Data members:
public:
Member function;
};

int main( )
{
DerivedClass D1,D2;
}
```

Access Specification: Public

- Public members of Base are public members for Derived class

- Private members of Base remain private members, irrespective of the visibility mode, but can be used through public member functions of the base class.

i.e. "They are invisible to the Derived class"

Base Class Access Specification

or

(Visibility mode)

Example: Single Inheritance

```
#include <iostream>

class AddNo {
int a, b, c;

public:
void getdata() { cin>>a>>b; }
void add() { c = a + b;  }
void print() { cout << c; }
};

class MulNo : public AddNo
{
int a, b, c;
public:
void getdata1() { cin>>a>>b; }
void mul() { c = a*b; }
void print1() { cout << c; }
};
```

```
int main( )
{
MulNo A
A.getdata();
A.add();
A.print();
A.getdata1();
A.mul();
A.print1();
}
```

Use of "protected" access specifier

- A base class in not exclusively "owned" by a derived class. A base class can be inherited by any number of different classes.
- There may be times when you want to keep a member of a base class private but still permit a derived class access to it.

SOLUTION: Designate the data as **protected.**

- ***Protected members of a base class are accessible to members of any class derived from that base.***
- ***Protected members, like private members, are not accessible outside the base or derived classes.***

Note:
Private members of the base class are always private to the derived class regardless of the access specifier.

Example: Single Inheritance with "protected" access specifier

```
#include <iostream>

class AddNo {
protected:
int a, b, c;
public:
void getdata() { cin>>a>>b; }
void add() { c = a + b;  }
void print() { cout << c; }
};

class MulNo : public AddNo
{
public:
void mul() { c = a*b; }
};
```

Access Specification:
protected:

- *data members of the base class remains private for its own implementation*
- *but it can be inherited in the derived class member functions.*

i.e. "They are now visible to the Derived class"
(Can be directly used in the derived class without an object)

```
int main( )
{
MulNo A;
A.getdata(); // function of base class

A.add();  // function of base class

A.print(); // function of base class

A.mul(); // function of Derived class

A.print(); // function of base class

}
```

Relationship between access specifiers

Base class member access specifier	Type of Inheritence (Visibility Mode)		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)

Accessing members as protected in derived class

But when a base class is inherited as **protected**,
public and protected members of the base class become protected members of the derived class.

```
class Base {
   protected:
      int a, b;
   public:
       void Setab(int n, int m)
            { a = n; b = m;}
};
```

```
class Derived: protected Base {
     int c;
   public:
     void Setc() { c = a + b;}
     void Showc() {
        cout << c << endl;
      }
};
```

```
int main() {
   Derived ob;

   ob.Setab(1,2); ERROR
   ob.a = 5;    //Not allowed

   ob.Setc(3);
   ob.Showabc();

}
```

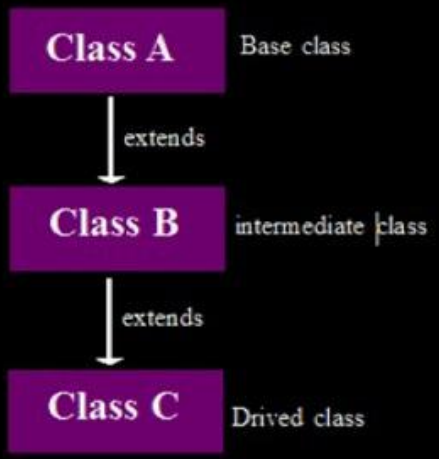
Accessing members as protected in derived class

Protected Access Specifier

- Private members of the base class are inaccessible to the derived class.
- Public members of the base class become protected members of the derived class.
- Protected members of the base class become protected members of the derived class.

i.e. only the public members of the derived class are accessible by the user application.

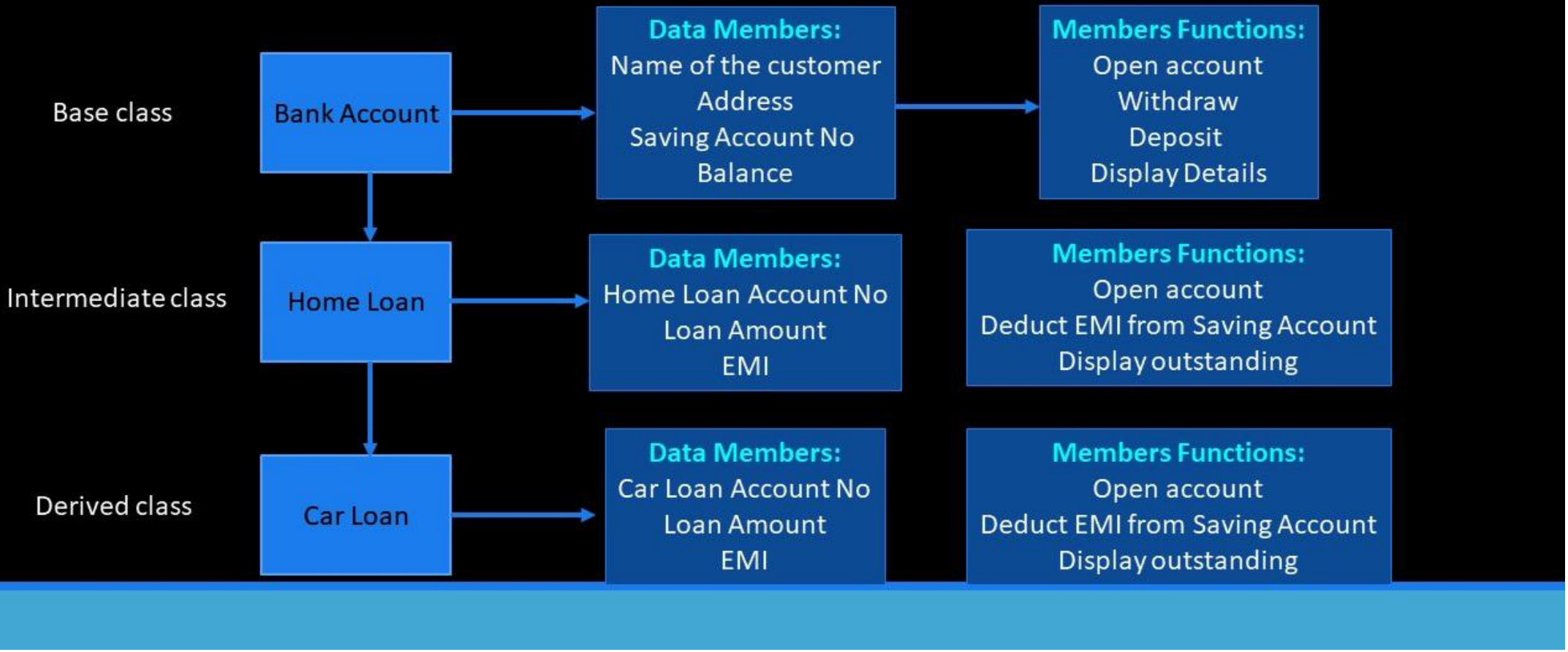
Multilevel Inheritance:



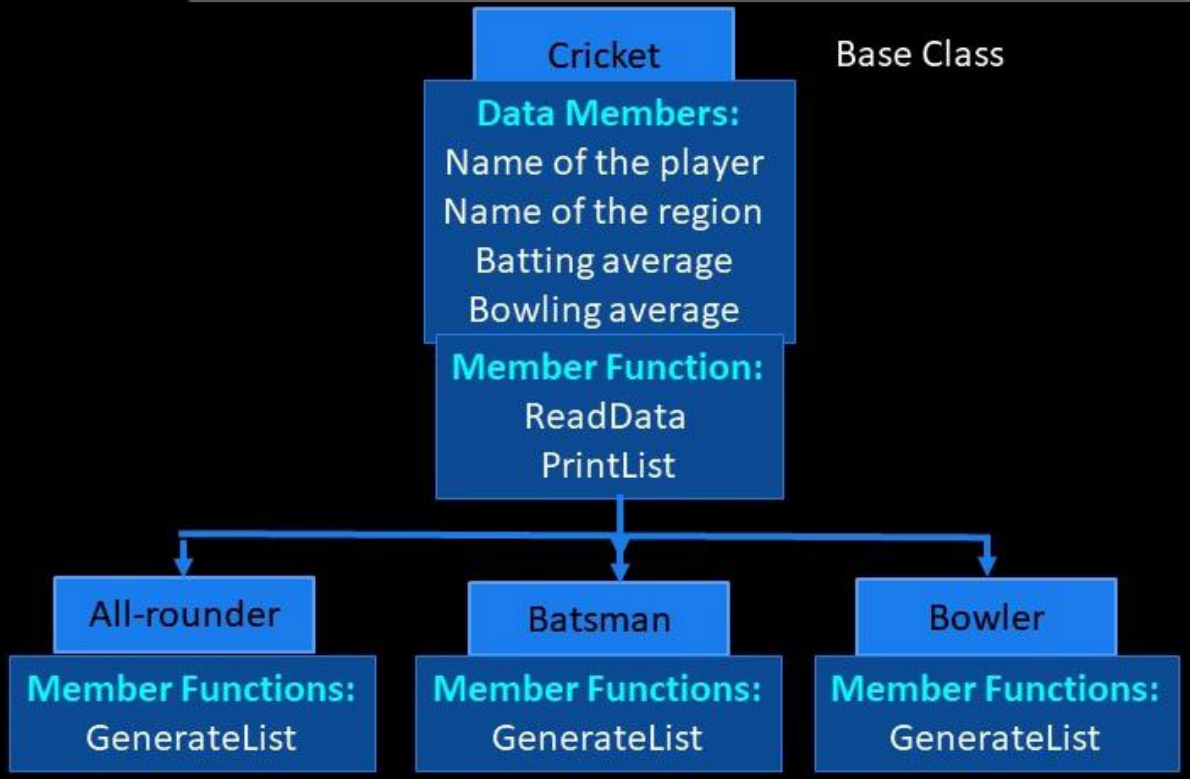

```
class ABC
{
int a, b;
public:
void getab() { cout<<"a and b :";
                cin>>a>>b;}
void putab() {cout<<a<<b;}
};
class XYZ : public ABC
{
int x, y;
public:
void getxy() { cout<<"x and y :";
                cin>>x>>y;}
void putxy() {cout<<x<<y;}
};
```

```
class ABC1 : public XYZ
{
int x1, y1;
public:
void getx1y1() { cout<<"x1 and y1 :";
                  cin>>x1>>y1;}
void putx1y1() { cout<<x1<<y1;}
};
int main()
{
ABC1 A;
A.getab();
A.putab();
A.getxy();
A.putxy();
A.getx1y1();
A.putx1y1();
}
```

Multilevel inheritance



Hierarchical Inheritance

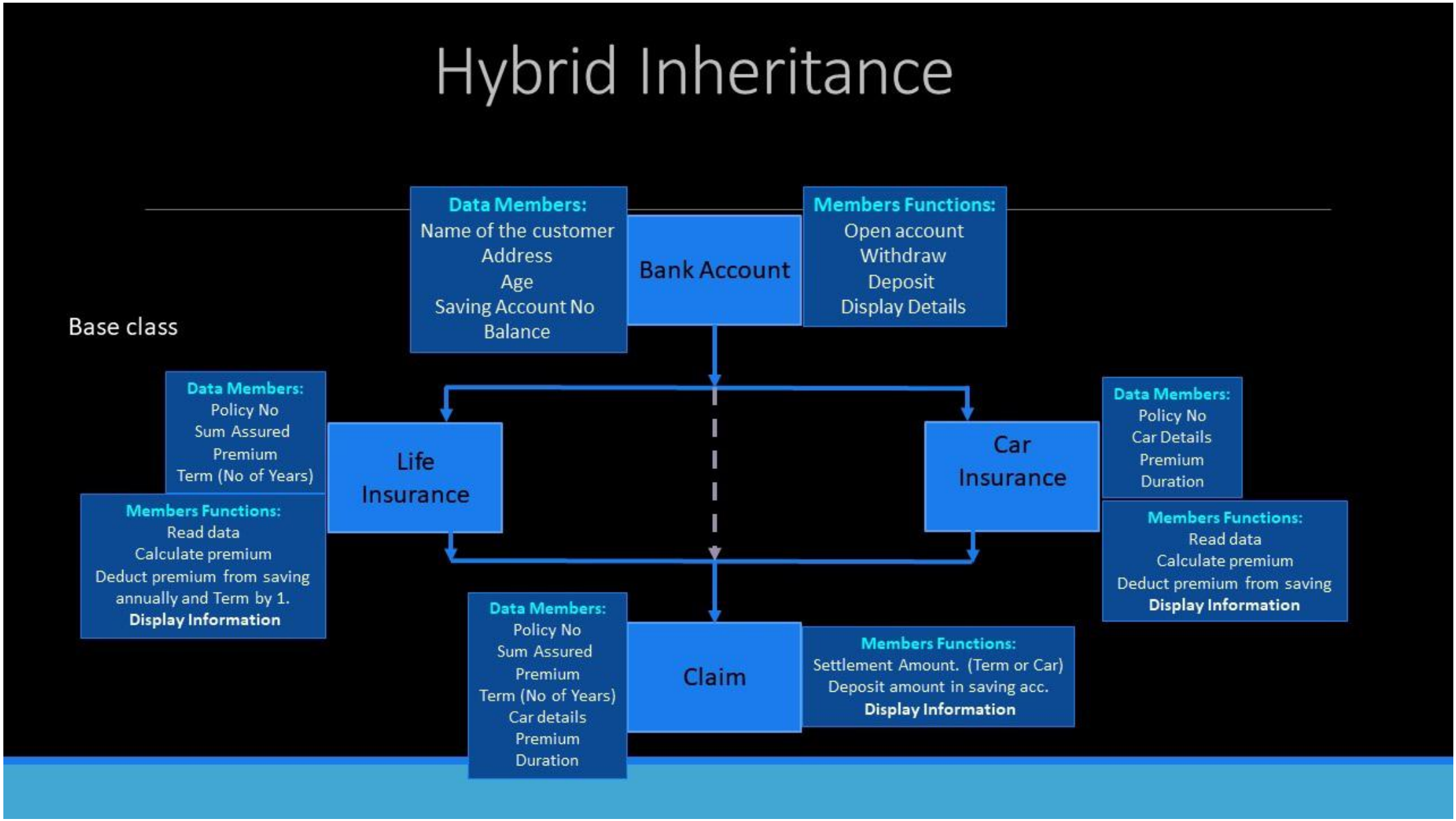


- It is like a tree structure
- Multiple classes can be derived from one base class

Syntax:

```
class Base {          };

class D1 : public Base {        };
class D2 : public Base {        };

class Dn : public Base {        };

int main()

{

D1 d1; D2 d2; D3 d3;

}
```

Hybrid Inheritance



Hybrid Inheritance

Implementation of Virtual Base class concept

```
class Bank {   }
class Life : public virtual Bank
{
}
class Car : public virtual Bank
{
}
class Claim : public Life, public Car
{
}
```

Use of Constructors and Destructors in Inheritance

- Derived-class constructor
  - Calls the constructor for its base class first to initialize its base-class members
  - If the derived-class constructor is omitted, its default constructor calls the base-class' default constructor
- Destructors are called in the reverse order of constructor calls.
  - Derived-class destructor is called before its base-class destructor

Constructors and Destructors in Single Inheritance

```
class ABC
{
int a, b;
public:
ABC(){cout<<"Enter  a and b :";
                cin>>a>>b;}
void putab(){cout<<a<<endl<<b;}
~ABC(){cout<<endl;
   cout<<"Base  class destructor";}
};
class XYZ : public ABC
{
int x, y;
public:
XYZ(){cout<<"Enter  x and y :";
                cin>>x>>y;}
void putxy(){cout<<x<<endl<<y;}
~XYZ(){cout<<endl;
   cout<<"Derived  class destructor";}
};
```

```
int main()
{
XYZ X;
X.putab();
X.putxy();
}
```

Output:

```
Enter a and b :10 20
Enter x and y :30 40
10
20
30
40
Derived class destructor
Base class destructor
```

Constructors and destructors in Multilevel Inheritance

```
class ABC
{
int a, b;
public:
ABC(){cout<<"Enter  a and b :";
cin>>a>>b;}
void putab(){cout<<a<<endl<<b;}
~ABC(){cout<<endl;
   cout<<"Base  class destructor";}
};
class XYZ : public ABC
{
int x, y;
public:
XYZ(){cout<<"Enter  x and y :";
   cin>>x>>y;}
void putxy(){cout<<endl<<x<<endl<<y;}
~XYZ(){cout<<endl;
   cout<<"Intermediate  class destructor";}
};
```

```
class ABC1 : public XYZ
{
int x1, y1;
public:
ABC1(){ cout<<"Enter  x1 and y1 :";
      cin>>x1>>y1;}
void putx1y1(){ cout<<endl<<x1<<endl<<y1;}
~ABC1(){cout<<endl;
      cout<<"Derived  class destructor";}
};
int main()
{
ABC1 X;
X.putab();
X.putxy();
X.putx1y1();
}
```

Output:

```
Enter a and b :10 20
Enter x and y :30 40
Enter x1 and y1 :50 60
10
20
30
40
50
60
Derived class destructor
Intermediate class destructor
Base class destructor
```

Parameterized constructor in Multilevel Inheritance

```
class ABC
{
int a, b;
public:
ABC(int a1, int b1) {a=a1; b=b1;}
void putab(){cout<<a<<endl<<b;}
~ABC(){cout<<endl;
   cout<<"Base  class destructor";}
};
class XYZ : public ABC
{
int x, y;
public:
XYZ(int x1,int y1) : ABC(x1,y1)
{ x=x1; y=y1;}
void putxy(){cout<<endl<<x<<endl<<y;}
~XYZ(){cout<<endl;
   cout<<"Intermediate  class destructor";}
};
```

```
class ABC1 : public XYZ
{
int x1, y1;
public:
ABC1(int x2, int y2) : XYZ(x2,y2)
{ x1=x2; y1=y2;}
void putx1y1(){ cout<<endl<<x1<<endl<<y1;}
~ABC1(){cout<<endl;
      cout<<"Derived  class destructor";}
};
int main()
{
ABC1 X(10,20);
X.putab();
X.putxy();
X.putx1y1();
}
```

Output:

```
10
20
10
20
10
20
Derived class destructor
Intermediate class destructor
Base class destructor
```

Parameterized Constructors in Multiple Inheritance

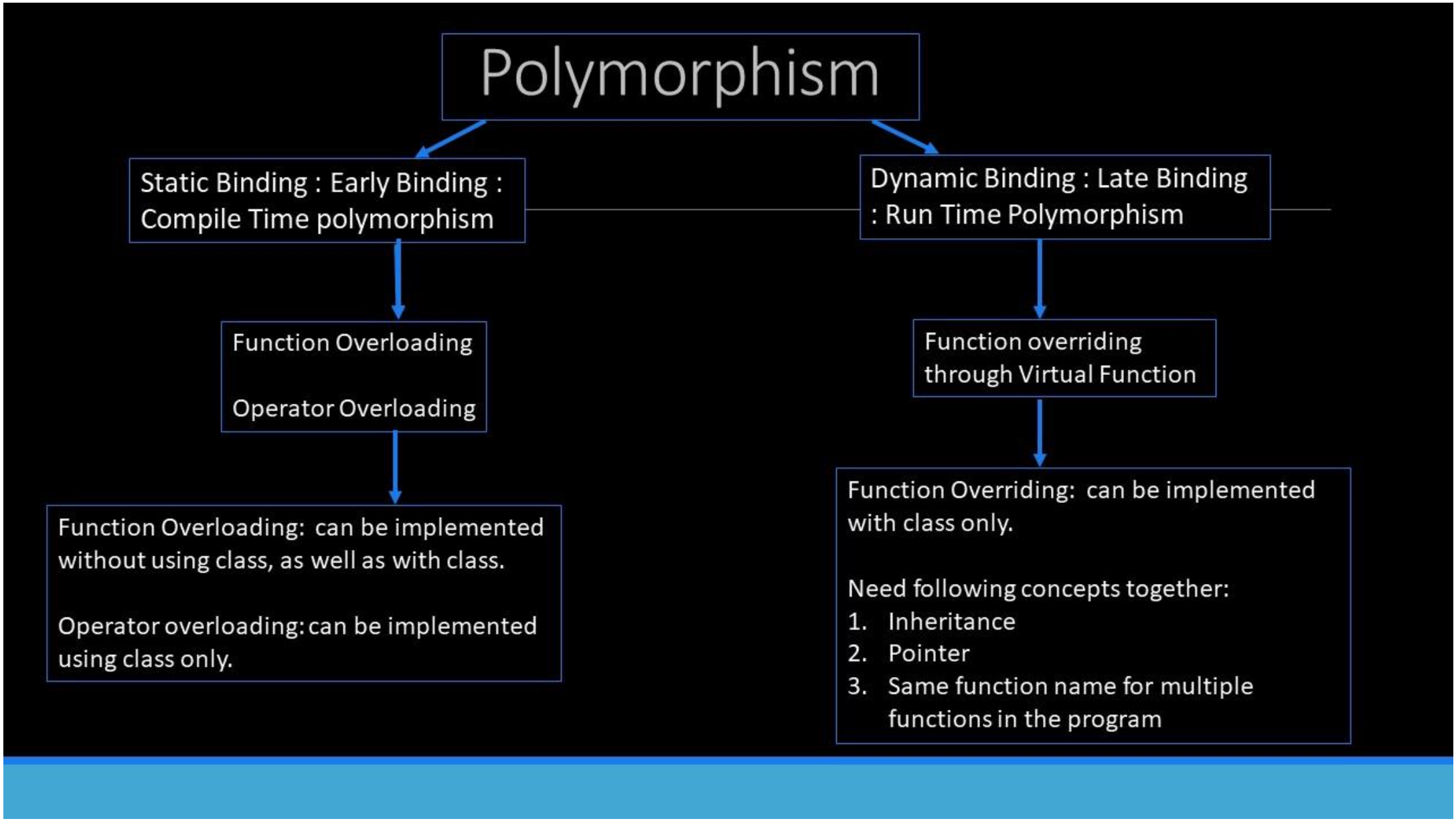
```
class ABC
{
int a, b;
public:
ABC(int a1,int b1){a=a1; b=b1;}
void putab(){cout<<"ABC class"<<endl;
            cout<<a<<endl<<b;}
~ABC(){cout<<endl;
   cout<<"ABC class destructor";}
};
class XYZ
{
int x, y;
public:
XYZ(int x1,int y1){ x=x1; y=y1;}
void putxy(){cout<<endl<<"XYZ class";
            cout<<endl<<x<<endl<<y;}
~XYZ(){cout<<endl;
   cout<<"XYZ class destructor";}
};
```

```
class ABC1 : public ABC, public XYZ
{
int x1, y1;
public:
ABC1(int x2, int y2) : ABC(x2,y2), XYZ(x2,y2)
{ x1=x2; y1=y2;}
void putx1y1(){ cout<<endl<<"ABC1 class";
            cout<<endl<<x1<<endl<<y1;}
~ABC1(){cout<<endl;
     cout<<"Derived class destructor";}
};
int main()
{
ABC1 X(10,20);
X.putab();
X.putxy();
X.putx1y1();
}
```

```
Output:
ABC class
10
20
XYZ class
10
20
ABC1 class
10
20
Derived class destructor
XYZ class destructor
ABC class destructor
```

Pointers and Objects:

```
class ABC
{
int a,b;
public:
void getdata (int a1,int b1){a=a1; b=b1;}
void putab(){cout<<"Data of the class: "<<endl;
                        cout<<a<<endl<<b;}
};
int main()
{
ABC *A;
ABC B;
A=&B;
A->getdata(20,30);
A->putab();
}
```


Polymorphism
Static Binding : Early Binding : Compile Time polymorphism
Dynamic Binding : Late Binding : Run Time Polymorphism
Function Overloading
Operator Overloading
Function overriding through Virtual Function
Function Overloading: can be implemented without using class, as well as with class.
Operator overloading: can be implemented using class only.
Function Overriding: can be implemented with class only.
Need following concepts together:
1. Inheritance
2. Pointer
3. Same function name for multiple functions in the program

Inheritance – Function Overloading (same function name in base and derived class)

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

class XYZ
{
   int x,y;
   public:
   void getdata()
   {
      cout<<"Enter x and Y :";
      cin>>x>>y;
   }
   void putab()
   {
      cout<<"value of x is : "<<x<<endl;
      cout<<"value of y is : "<<y;
   }
};
```

```
class ABC : public XYZ
{
int a,b;
public:
void getdata ()
{
   cout<<"Enter a and b :";
   cin>>a>>b;
}
void putab()
{
   cout<<"Data of the class ABC: "<<endl;
   cout<<"value of a is : "<<a<<endl;
   cout<<"value of b is : "<<b;
}
};
```

```
int main()
{
ABC A;
A.getdata();
A.putab();

//XYZ X;
//X.getdata();
//X.putdata();
}
```

Output:

```
Enter a and b :12 14
Data of the class ABC:
value of a is : 12
value of b is : 14
```

Inheritance – Pointer Object and Function Overloading

```
#include<iostream>
using namespace std;
class XYZ
{
   int x,y;
   public:
   void getdata()
   {
      cout<<"Enter x and Y :";
      cin>>x>>y;
   }
   void putdata()
   {
      cout<<"Data of the class XYZ  :"<<endl;
      cout<<"value of x is : "<<x<<endl;
      cout<<"value of y is : "<<y;
   }
};
```

```
class ABC : public XYZ
{
int a,b;
public:
   void getdata()
   {
      cout<<"Enter a and b :";
      cin>>a>>b;
   }
   void putdata()
   {
      cout<<"Data of the class ABC: "<<endl;
      cout<<"value of a is : "<<a<<endl;
      cout<<"value of b is : "<<b;
   }
};
```

```
int main()
{
ABC *A;
ABC B;
A=&B;
A->getdata();
A->putdata();

XYZ X;
A =&X;
A->getdata();
A->putdata();
}
```

Error:

Invalid Conversion from XYZ * to ABC *

Run time polymorphism – virtual function

```
#include<iostream>
using namespace std;
class XYZ
{
   int x,y;
   public:
   virtual void getdata()
   {
      cout<<"Enter x and Y :";
      cin>>x>>y;
   }
   virtual void putdata()
   {
      cout<<"Data of the class XYZ  :"<<endl;
      cout<<"value of x is : "<<x<<endl;
      cout<<"value of y is : "<<y<<endl;
   }
};
```

```
class ABC : public XYZ
{
int a,b;
public:
void getdata ()
{
   cout<<"Enter a and b :";
   cin>>a>>b;
}
void putdata()
{
   cout<<"Data of the class ABC: "<<endl;
   cout<<"value of a is : "<<a<<endl;
   cout<<"value of b is : "<<b<<endl;
}
};
```

```
int main()
{
XYZ *X;
ABC B;
X=&B;
X->getdata();
X->putdata();

XYZ X1;
X=&X1;
X->getdata();
X->putdata();
}
```

```
Output:
Enter a and b :10 20
Data of the class ABC:
value of a is : 10
value of b is : 20
Enter x and Y :30 40
Data of the class XYZ  :
value of x is : 30
value of y is : 40
```

Virtual Function

- A virtual function is a member function of a base class which is redefined in the derived class.
- It is achieved by using the keyword ‘virtual’ in the base class.
- The function call is decided on the type of referred object not according to the type of pointer.

Rules to use virtual function

- Virtual functions cannot be static and friend to another class
- Virtual functions must be accessed using pointers or references of base class type
- The function prototype should be same in both base and derived classes
- A class must not have a virtual constructor. But it can have a virtual destructor
- They are always defined in the base class and redefined in the derived class