

Inheritance

UNIT III-Inheritance

- “The language mechanism by which one class acquires the properties (data and operations) of another class
- **Base Class (or superclass)**: the class being inherited from
- **Derived Class (or subclass)**: the class that inherits
- Inherit the members of an existing class. This existing class is called the base class, and the new class is referred to as the derived class.
 - It is the technique of organizing the information in hierarchical form.
 - Inheritance allows new classes to be built from older classes instead
 - of being rewritten from scratch.
- -Existing class is known as base class or parent class or super class and
 - new class is known as derived class or child class or sub class.
- -In inheritance base class members are inherited to derived class and
 - also new members can be added to derived class.
- Reuse the code functionality and fast implementation time.

NOTE : All members of a class except Private, are inherited.

Inheritance

Advantages of inheritance

- When a class inherits from another class, there are three benefits:
- (1) You can reuse the methods and data of the existing class
- (2) You can extend the existing class by adding new data and new methods
- (3) You can modify the existing class by overloading its methods with your own implementations

Rules for building a class hierarchy

- Derived classes are special cases of base classes
- A derived class can also serve as a base class for new classes.
- There is no limit on the depth of inheritance allowed in C++ (as far as it is within the limits of your compiler)
- It is possible for a class to be a base class for more than one derived class
- **Polymorphism:**
- Any code you write to manipulate a base class will also work with any class derived from the base class.
- C++ general rule for passing objects to a function:
 - “the actual parameters and their corresponding formal parameters must be of the same type”
- With inheritance, C++ relaxes this rule:
 - “the type of the actual parameter can be a class derived from the class of the formal parameter”

- C ++, not only supports the access specifiers private and public, but also an important access specifiers protected, which is significant in class inheritance.
- A class can use all the three visibility modes as illustrated below.

```
class classname
{
    private: //visible to member functions within its class
            //but not in derived class
    protected: //visible to member functions within its class
            //and its derived class
    public: //visible to member functions within its class,
            //derived classes and through object
};
```

Syntax & EXAMPLE

Syntax : `class Subclass_name : access_mode Superclass_name`

Access Mode is used to specify, the mode in which the properties of superclass will be inherited into subclass, public, private or protected.

EXAMPLE:

```
class Animal // base class
{ public:
  int legs = 4;
};
```

```
class Dog : public Animal // derived class
{ public:
  int tail = 1;
};
```

```
int main()
{
  Dog d;
  cout << d.legs;
  cout << d.tail;
}
```

- Publicly inherited.

1. Private members of the base class cannot be inherited.
2. Public members of the base class become public to derived class.
3. Protected members of the base class become protected to derived class. Therefore they are accessible to the member functions of the derived class. And also further inheritance is possible.

- Privately inherited

1. public members of base class become private to derived class. Therefore the public members of the base class can only be accessed by the members of the derived class.
2. Private members of the base class cannot be inherited to the derived class.
3. Protected members of the base class also become private to the derived class. they can have access to the member functions of the derived class.

- protected mode.

1. Public and protected members of the base class become protected to the derived class.

Visibility of inherited members

Visibility of inherited members

Base class	Derived class visibility		
	Publicly	privately	protected
Private->	Not inherited	Not inherited	Not inherited
Protected->	protected	private	protected
Public->	public	private	protected

Visibility rules

- Any private members in the base class are not accessible to the derived class (because any member that is declared with private visibility is accessible only to methods of the class)
- What if we want the derived class to have access to the base class members?
 - 1) use public access => public access allows access to other classes in addition to derived classes
 - 2) use a friend declaration => this is also poor design and would require friend declaration for each derived class
 - 3) make members protected => allows access only to derived classes

A protected class member is private to every class except a derived class, but declaring data members as protected or public violates the spirit of encapsulation
- 1) write accessor and modifier methods => the best alternative

Note: However, if a protected declaration allows you to avoid convoluted code, then it is not unreasonable to use it.

Class Derivation

Constructors and Destructors

- Constructors and destructors are not inherited
 - Each derived class should define its constructors/destructor
 - If no constructor is written=> hidden constructor is generated and will call the base default constructor for the inherited portion and then apply the default initialization for any additional data members
- When a derived object is instantiated, memory is allocated for
 - Base object
 - Added parts
- Initialization occurs in two stages:
 - the base class constructors are invoked to initialize the base objects
 - the derived class constructor is used to complete the task
- The derived class constructor specifies appropriate base class constructor in the initialization list
 - If there is no constructor in base class, the compiler created default constructor used
- If the base class is derived, the procedure is applied recursively

A derived class inherits all base class methods with the following exceptions:

- **Constructors, destructors and copy constructors of the base class.**
- **Overloaded operators of the base class.**
- **The friend functions of the base class**

Types of inheritance

1. Single Inheritance
2. Multiple Inheritance
3. Hierarchical Inheritance
4. Multilevel Inheritance
5. Hybrid Inheritance

Single inheritance

- In this type of inheritance one derived class inherits from only one base class. It is the most simplest form of Inheritance.

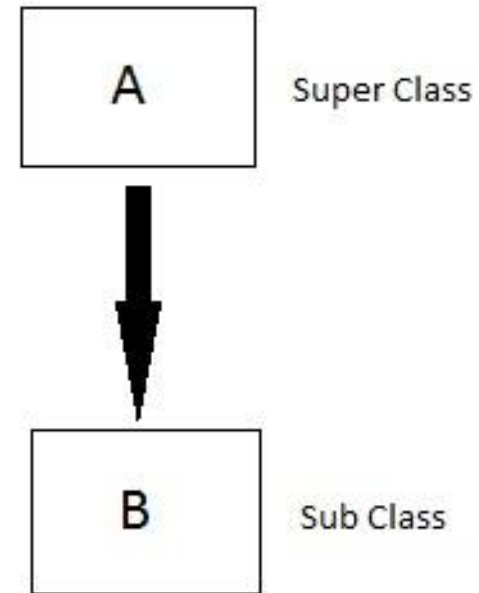
base class:

```
class A  
{  
  
}
```

//members of A

Derived class syntax:

```
class B: public A  
{  
    //members of B  
};
```



Example

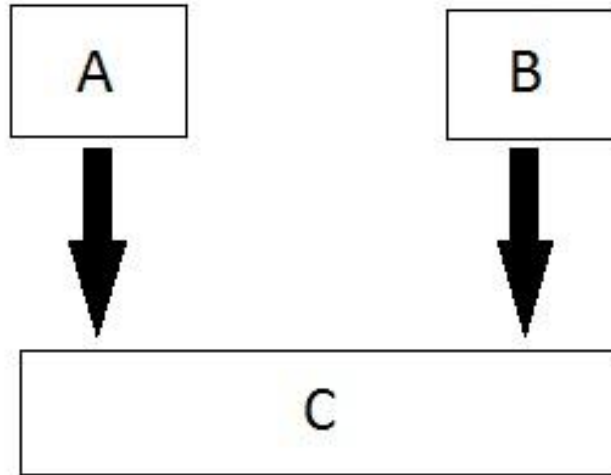
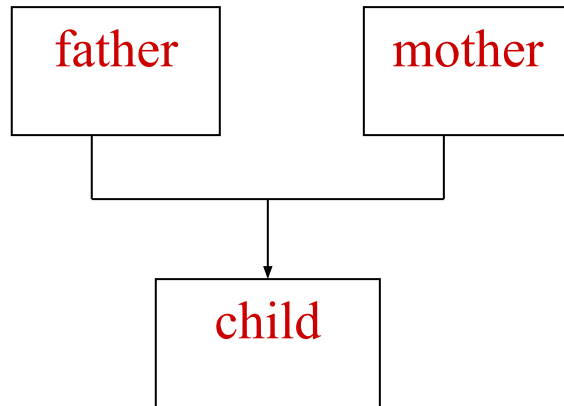
```
class Shape
{
protected:
    float width, height;
public:
    void set_data (float a, float b)
    {
        width = a;
        height = b;
    }
};
```

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

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

Multiple Inheritance

- In this type of inheritance a single derived class may inherit from two or more than two base classes.



Example

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

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

```
class statement:public student,public sports
{
    int tot,avg;
    public:
        void display()
        {
            tot=(m1+m2+sm);
            avg=tot/3;
            cout<<"\n\n\tRoll No  : "<<rno<<"\n\tTotal  : "<<tot;
            cout<<"\n\tAverage  : "<<avg;
        }
};

void main()
{
    clrscr();
    statement obj;
    obj.get();
    obj.getsm();
    obj.display();
    getch();
}
```


Note:- ambiguity resolution in multiple inheritance same function name is used in more than one base class ambiguity occurs which is to be executed

- Use scope resolution operator.

```
class X
{
public:
    void display()
    {
        cout<<"\n class X";
    }
};
```

```
class Y
{
public:
    void display()
    {
        cout<<"\n class Y";
    }
};
```

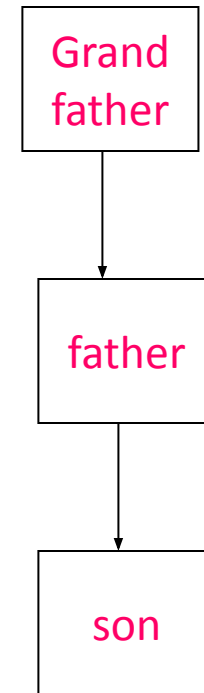
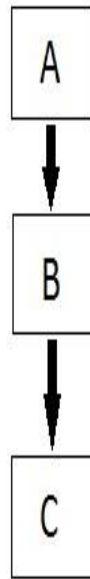
```
class Z: public X,public Y
{
public:
    void display()
    {
        X:: display();
    }
};
```

```
Int main()
{
    Z z;
    z.display();
    or z.X::display();
    z Y:: display();
    return 0;
}
```

Output:
class X
Class Y

Multilevel Inheritance

- In this type of inheritance the derived class inherits from a class, which in turn inherits from some other class. The Super class for one, is sub class for the other.



```
class A  
{  
    public:  
        void display()  
        {  
            cout<<"Base class content.";  
        }  
};
```

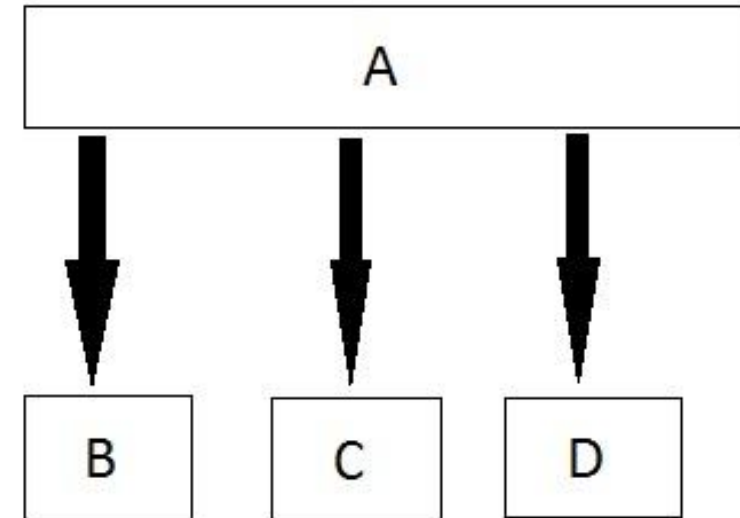
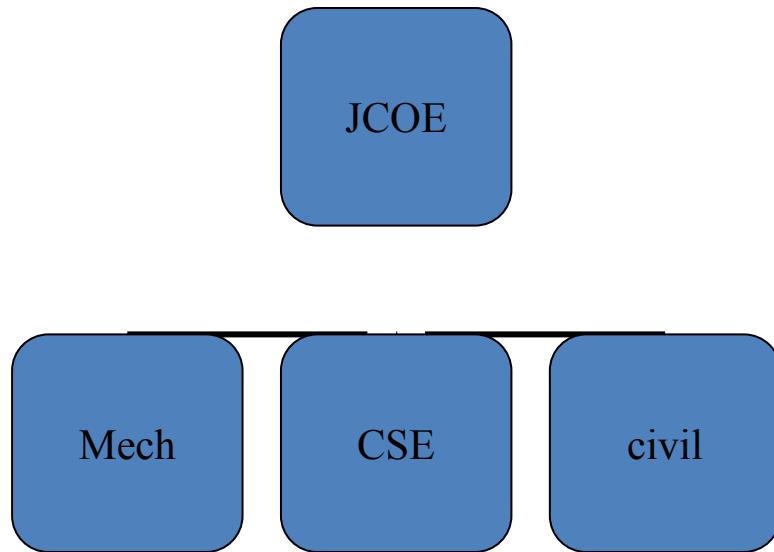
```
class B : public A  
{  
  
};
```

```
class C : public B  
{  
  
};
```

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

Hierarchical Inheritance

- In this type of inheritance, multiple derived classes inherit from a single base class.



example

```
class A //Base Class
```

```
{  
    public:  
    int a,b;  
    void getnumber()  
    {  
        cout<<"\n\nEnter Number :::\t";  
        cin>>a;  
    }  
};
```

```
class B : public A //Derived Class 1
```

```
{  
    public:  
    void square()  
    {  
        getnumber(); //Call Base class property  
        cout<<"\n\n\tSquare of the number :::\t"<<(a*a);  
    }  
};
```

```
class C :public A //Derived Class 2
```

```
{  
    public:  
    void cube()  
    {  
        getnumber(); //Call Base class property  
        cout<<"\n\n\tCube of the number :::\t"<<(a*a*a);  
    }  
};
```

```
int main()
```

```
{  
    clrscr();
```

```
    B b1;    //b1 is object of Derived class 1
```

```
    b1.square(); //call member function of class B
```

```
    C c1;    //c1 is object of Derived class 2
```

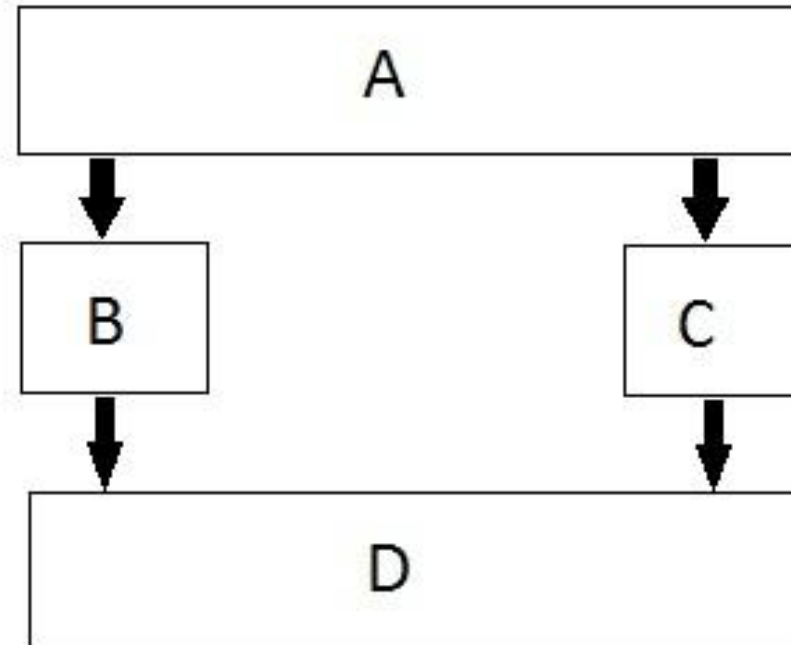
```
    c1.cube(); //call member function of class C
```

```
    getch();
```

```
}
```

Hybrid Inheritance

- Hybrid Inheritance is combination of Hierarchical and Multilevel Inheritance.



example

```
class arithmetic
{
protected:
int num1, num2;
public:
void getdata()
{
cout<<"For Addition:";
cout<<"\nEnter the first number: ";
cin>>num1;
cout<<"\nEnter the second number: ";
cin>>num2;
}
};

class plus:public arithmetic
{
protected:
int sum;
public:
void add()
{
sum=num1+num2;
}
};
```

```
class minus
{
protected:
int n1,n2,diff;
public:
void sub()
{
cout<<"\nFor Subtraction:";
cout<<"\nEnter the first number: ";
cin>>n1;
cout<<"\nEnter the second number: ";
cin>>n2;
diff=n1-n2;
}
};
```

```
class result:public plus, public minus
{
public:
void display()
{
cout<<sum;
cout<<diff;
}
};

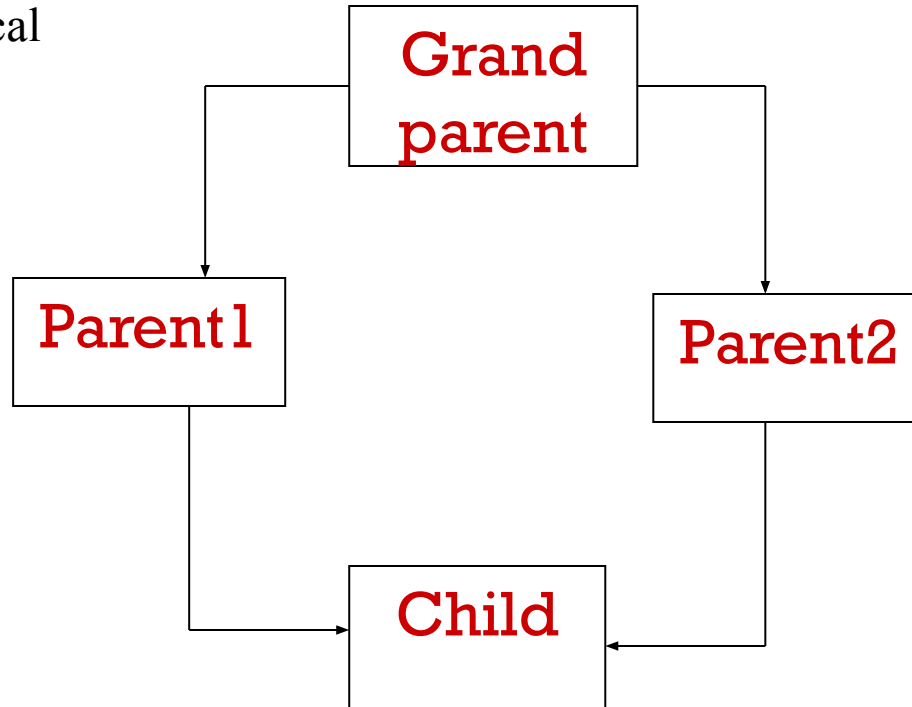
void main()
{
clrscr();
result z;
z.getdata();
z.add();
z.sub();
z.display();
getch();
}
```

Multipath Inheritance or Virtual base classes

Multipath inheritance may lead to duplication of inherited members from a grandparent base class. This may be avoided by making the common base class a virtual base class. When a class is made a virtual base class, C++ takes necessary care to see that only one copy of that class is inherited

Derivation of a class from other derived classes, which are derived from the same base class is called multipath inheritance.

-Multipath inheritance involves more than one form of inheritance namely multilevel, multiple, and hierarchical



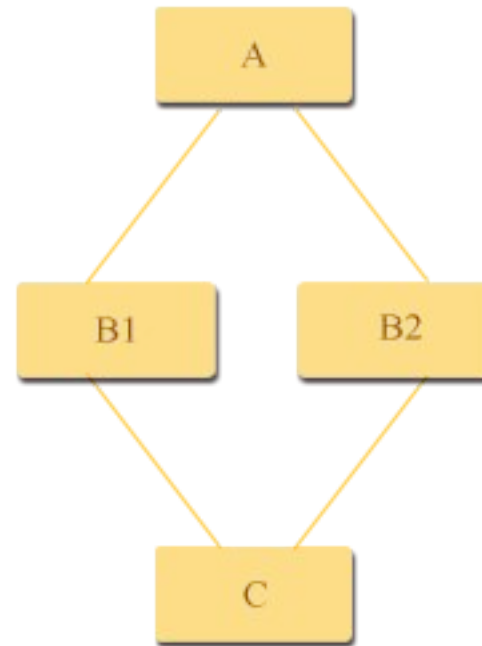
EXAMPLE

```
class A
{
    .....
};
```

```
class B1 : virtual public A
{
    .....
};
```

```
class B2 : virtual public A
{
    .....
};
```

```
class C : public B1, public B2
{
    .....// only one copy of A
    .....// will be inherited
};
```



Public, Protected, Private Inheritance

```
class A {  
public:  
    int i;  
protected:  
    int j;  
private:  
    int k;  
};
```

```
Class B : public A {  
    // ...  
};  
Class C : protected A {  
    // ...  
};  
Class D : private A {  
    // ...  
};
```

- Class A declares 3 variables
 - **i is public** to all users of class A
 - **j is protected**. It can only be used by methods in class A or its derived classes (+ friends)
 - **k is private**. It can only be used by methods in class A (+ friends)
- Class B uses **public inheritance** from A
 - **i remains public** to all users of class B
 - **j remains protected**. It can be used by methods in class B or its derived classes
- Class C uses **protected inheritance** from A
 - **i becomes protected** in C, so the only users of class C that can access i are the methods of class C
 - **j remains protected**. It can be used by methods in class C or its derived classes
- Class D uses **private inheritance** from A
 - **i and j become private** in D, so only methods of class D can access them.

Abstract Classes

- Abstract Class is one that is not used to create Objects.
- It is designed to act as a base class(to be inherited by other classes).
- It is design concept in prog. Development upon which other classes may be built.
-

Abstract Base Classes and Pure Virtual Functions

- Pure virtual function: a virtual member function that must be overridden in a derived class that has objects
- Abstract base class contains at least one pure virtual function:
 virtual void Y() = 0;
- The = 0 indicates a pure virtual function
- Must have no function definition in the base class

Abstract Base Classes and Pure Virtual Functions

- Abstract base class: class that can have no objects. Serves as a basis for derived classes that may/will have objects
- A class becomes an abstract base class when one or more of its member functions is a pure virtual function

Class and Member Construction Order

```
class A {
public:
    A(int i) :m_i(i) {
        cout << "A" << endl;}
    ~A() {cout<<"~A"<<endl;}
private:
    int m_i;
};

class B : public A {
public:
    B(int i, int j)
        : A(i), m_j(j) {
        cout << "B" << endl;}
    ~B() {cout << "~B" << endl;}
private:
    int m_j;
};

int main (int, char *[]) {
    B b(2,3);
    return 0;
};
```

- In the main function, the B constructor is called on object b
 - Passes in integer values 2 and 3
- B constructor calls A constructor
 - passes value 2 to A constructor via base/member initialization list
- A constructor initializes **m_i**
 - with the passed value 2
- Body of A constructor runs
 - Outputs "A"
- B constructor initializes **m_j**
 - with passed value 3
- Body of B constructor runs
 - outputs "B"

Class and Member Destruction Order

```
class A {
public:
    A(int i) :m_i(i) {
        cout << "A" << endl;}
    ~A() {cout<<"~A"<<endl;}
private:
    int m_i;
};

class B : public A {
public:
    B(int i, int j) :A(i), m_j(j) {
        cout << "B" << endl;}
    ~B() {cout << "~B" << endl;}
private:
    int m_j;
};

int main (int, char *[]) {
    B b(2,3);
    return 0;
};
```

- B destructor called on object b in main
- Body of B destructor runs
 - outputs “~B”
- B destructor calls “destructor” of **m_j**
 - **int** is a built-in type, so it’s a no-op
- B destructor calls A destructor
- Body of A destructor runs
 - outputs “~A”
- A destructor calls “destructor” of **m_i**
 - again a no-op
- Compare orders of construction and destruction of base, members, body
 - at the level of each class, order of steps is reversed in constructor vs. destructor
 - ctor: base class, members, body
 - dtor: body, members, base class

Virtual Functions

```
class A {
public:
    A () {cout<<" A";}
    virtual ~A () {cout<<" ~A";}
    virtual f(int);
};

class B : public A {
public:
    B () :A() {cout<<" B";}
    virtual ~B() {cout<<" ~B";}
    virtual f(int) override; //C++11
};

int main (int, char *[]) {
    // prints "A B"
    A *ap = new B;

    // prints "~B ~A" : would only
    // print "~A" if non-virtual
    delete ap;

    return 0;
};
```

- Used to support polymorphism with pointers and references
- Declared virtual in a base class
- Can override in derived class
 - Overriding only happens when signatures are the same
 - Otherwise it just *overloads* the function or operator name
 - More about overloading next lecture
- Ensures derived class function definition is resolved *dynamically*
 - E.g., that destructors farther down the hierarchy get called
- Use **final** to prevent overriding of a virtual method
- Use **override** (C++11) in derived class to ensure that the signatures match (error if not)

Virtual Functions

```
class A {
public:
    void x() {cout<<"A::x";};
    virtual void y() {cout<<"A::y";};
};
```

```
class B : public A {
public:
    void x() {cout<<"B::x";};
    virtual void y() {cout<<"B::y";};
};
```

```
int main () {
    B b;
    A *ap = &b; B *bp = &b;
    b.x (); // prints "B::x"
    b.y (); // prints "B::y"
    bp->x (); // prints "B::x"
    bp->y (); // prints "B::y"
    ap->x (); // prints "A::x"
    ap->y (); // prints "B::y"
    return 0;
};
```

- Only matter with pointer or reference
 - Calls on object itself resolved statically
 - E.g., **b.y ()** ;
- Look first at pointer/reference type
 - If non-virtual there, resolve statically
 - E.g., **ap->x ()** ;
 - If virtual there, resolve dynamically
 - E.g., **ap->y ()** ;
- Note that virtual keyword need not be repeated in derived classes
 - But it's good style to do so
- Caller can force static resolution of a virtual function via scope operator
 - E.g., **ap->A::y ()** ; prints "A::y"

Potential Problem: Class Slicing

- Catch derived exception types by reference
- Also pass derived types by reference
- Otherwise a temporary variable is created
 - Loses original exception's “dynamic type”
 - Results in “the class slicing problem” where only the base class parts and not derived class parts copy

Pure Virtual Functions

```
class A {  
public:  
    virtual void x() = 0;  
    virtual void y() = 0;  
};
```

```
class B : public A {  
public:  
    virtual void x();  
};
```

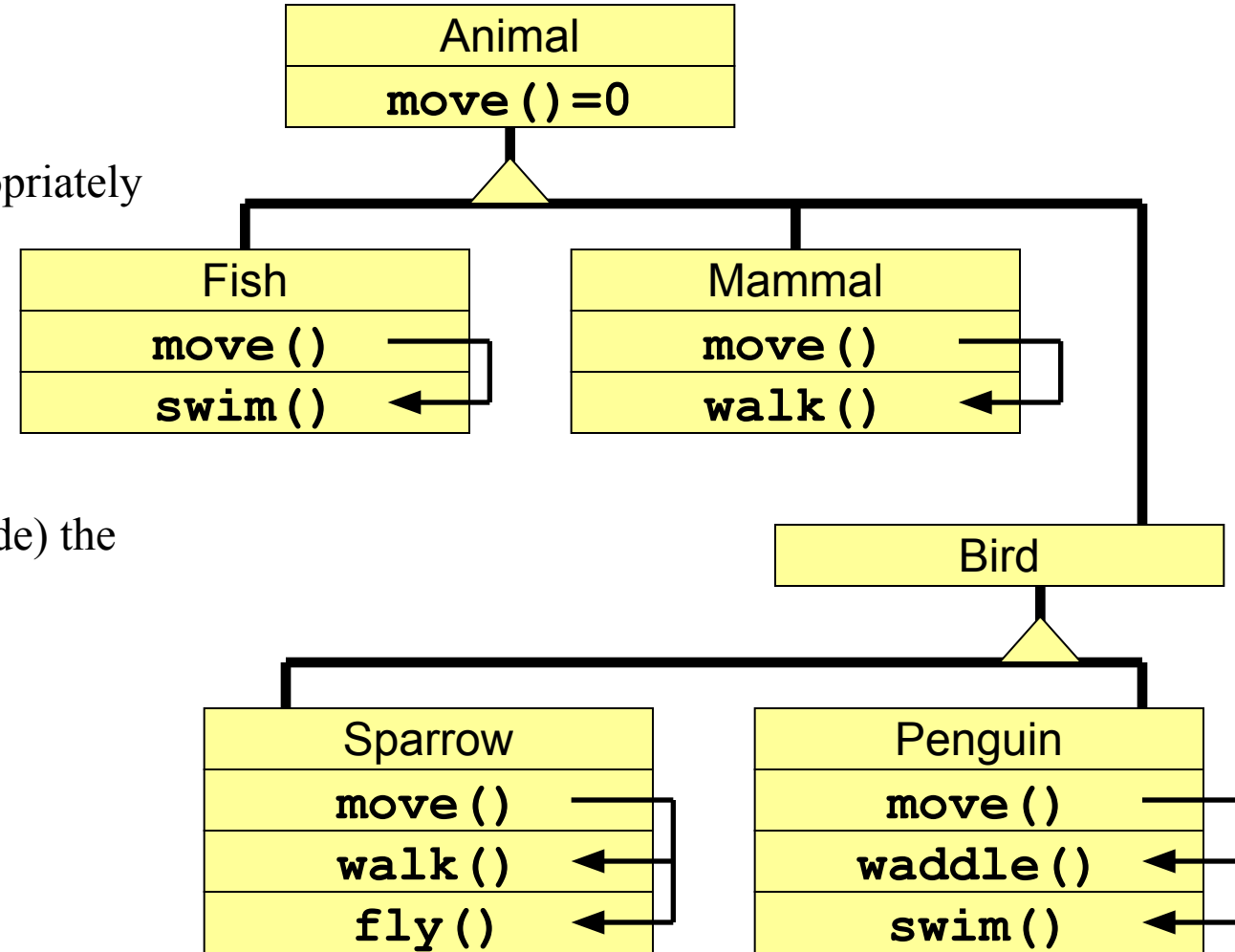
```
class C : public B {  
public:  
    virtual void y();  
};
```

```
int main () {  
    A * ap = new C;  
    ap->x ();  
    ap->y ();  
    delete ap;  
    return 0;  
};
```

- A is an abstract (base) class
 - Similar to an interface in Java
 - Declares pure virtual functions (=0)
 - May also have non-virtual methods, as well as virtual methods that are not pure virtual
- Derived classes override pure virtual methods
 - B overrides x(), C overrides y()
- Can't instantiate an abstract class
 - class that declares pure virtual functions
 - or inherits ones that are not overridden
 - A and B are abstract, can create a C
- Can still have a pointer or reference to an abstract class type
 - Useful for polymorphism

Design with Pure Virtual Functions

- Pure virtual functions let us specify interfaces appropriately
 - But let us defer implementation decisions until later (subclasses)
- As the type hierarchy is extended, pure virtual functions are replaced
 - By virtual functions that fill in (and may override) the implementation details
 - Key idea: *refinement*



Summary: Tips on Inheritance Polymorphism

- A key tension
 - Push common code and variables up into base classes
 - Make base classes as general as possible
- Use abstract base classes to declare interfaces
- Use public inheritance to make sets of polymorphic types
- Use private or protected inheritance only for encapsulation
- Inheritance polymorphism depends on dynamic typing
 - Use a base-class pointer (or reference) if you want inheritance polymorphism of the objects pointed to (or referenced)
 - Use virtual member functions for dynamic overriding
- Even though you don't have to, label each *inherited* virtual (and pure virtual) method “virtual” in derived classes
- Use **final** (C++11) to prevent overriding of a virtual method
- Use **override** (C++11) to make sure signatures match

Inheritance: Constructors & Destructors

- **Constructors & Destructors**

- When a base class and a derived class both have constructor and destructor functions
 - Constructor functions are executed in order of derivation – base class before derived class.
 - Destructor functions are executed in reverse order – the derived class's destructor is executed before the base class's destructor.
- A derived class does not inherit the constructors of its base class.

Constructors & Destructors

```
class Base {  
    public:  
        Base() { cout << "Constructor Base Class\n";}  
        ~Base() {cout << "Destructing Base Class\n";}  
};  
class Derived : public Base { public:  
    Derived() { cout << Constructor Derived Class\n";}  
    ~Derived(){ cout << Destructing Derived Class\n";}  
};
```

```
int main() {  
    Derived ob;  
    return o;  
}
```

```
---- OUTPUT ----  
Constructor Base Class  Constructor  
Derived Class  Destructing Derived  
Class  Destructing Base Class
```

Constructors & Destructors

- Passing an argument to a derived class's constructor

```
Class Base {  
    public:  
        Base() {cout << "Constructor Base Class\n";}  
        ~Base() {cout << "Destructing Base Class\n";}  
};
```

```
Class Derived : public Base {  
    int J;  
    public:  
        Derived(int X) {  
            cout << "Constructor Derived Class\n";  
            J = X;  
        }  
        ~Derived() { cout << "Destructing Derived Class\n"; }  
        void ShowJ() {  
            cout << "J: " << J << "\n"; }  
};
```

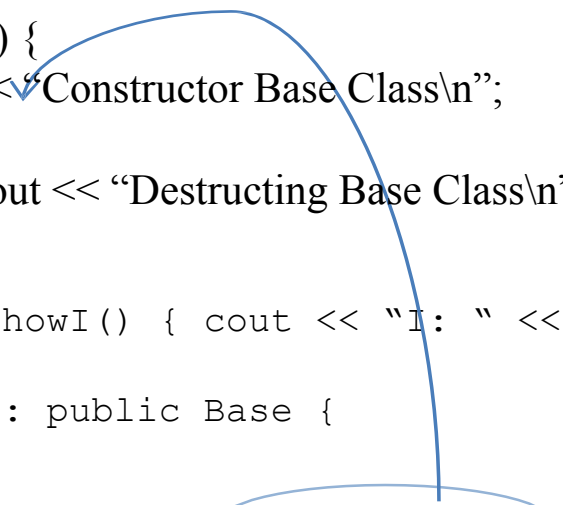
```
int main()  
{  
    Derived Ob(10);  
    Ob.ShowJ();  
    return 0;  
} // end
```

```
} {  
    d Ob(10);  
    wJ(); 0;  
    main
```


Arguments to both Derived and Base Constructors

```
Class Base { int I;
public:
    Base(int Y) {
        cout << "Constructor Base Class\n";
        I = Y;}
    ~Base(){cout << "Destructing Base Class\n";}

    void ShowI() { cout << "I: " << I << endl; }
};
Class Derived : public Base {
    int J;
public:
    Derived(int X) : Base (X) {
        cout << Constructor Derived Class\n";
        J = X;
    }
    ~Derived(){ cout << Destructing Derived Class\n"; }
    void ShowJ() {
        cout << "J:" << J << "\n"; }
};
```

A blue curved arrow originates from the argument 'X' in the 'Base (X)' part of the 'Derived(int X)' constructor and points to the parameter 'Y' in the 'Base(int Y)' constructor. Additionally, the 'Base (X)' text is circled in blue.

```
int main() {    Derived
                Ob(10);

                Ob.ShowI();
                Ob.ShowJ();
                return 0;
            } // end main
```

- **Different arguments to the Base – All arguments to the Derived.**

```
Class Base { int I; public:
```

```
    Base(int Y) {  
        cout << "Constructor Base Class\n"; I = Y;}  
    ~Base(){cout << "Destructing Base Class\n";}  
    void ShowI() { cout << "I: " << I << endl; }
```

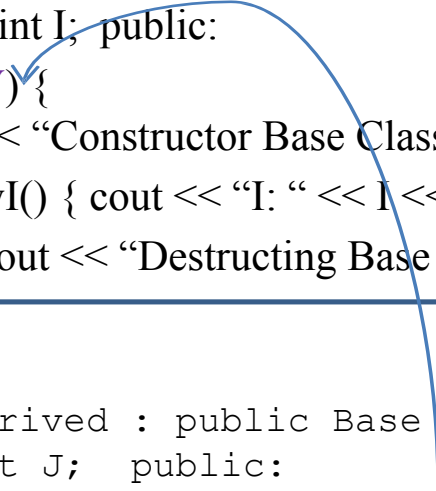
```
};  
Class Derived : public Base {  
    int J; public:  
    Derived(int X, int Y) : Base (Y) {  
        cout << Constructor Derived Class\n";  
        J = X;  
    }
```

```
    ~Derived(){ cout << Destructing Derived Class\n";}  
    void ShowJ() {  
        cout << << "J:" << J << "\n"; }  
};
```

```
int main() {    Derived  
    Ob(5,8);  
  
    Ob.ShowI();  
    Ob.ShowJ();  
    return 0;  
} // end main
```

OK – If Only Base has Argument

```
Class Base { int I; public:  
    Base(int Y){  
        cout << "Constructor Base Class\n"; I = Y;}  
    void ShowI() { cout << "I: " << I << endl; }  
    ~Base(){cout << "Destructing Base Class\n";}
```



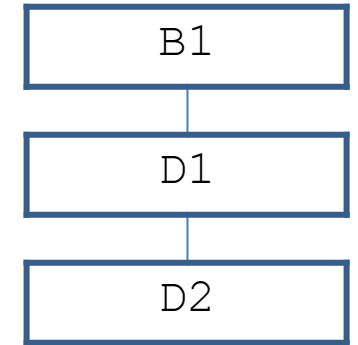
```
};  
Class Derived : public Base {  
    int J; public:  
    Derived(int X) : Base (X) {  
        cout << Constructor Derived Class\n"; J = 0;  
        not used here  
    }  
    ~Derived(){ cout << Destructing Derived Class\n"; }  
    void ShowJ() {  
        cout << "J:" << J << "\n"; }  
};
```

```
int main() { Derived  
    Ob(10);  
  
    Ob.ShowI();  
    Ob.ShowJ();  
    return 0;  
} // end main
```

Multiple Inheritance

1. Multilevel Class Hierarchy

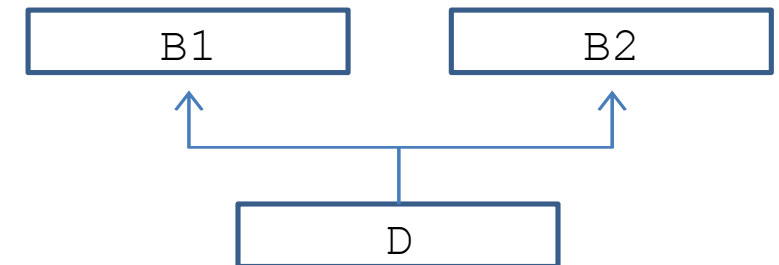
- Constructor functions of all classes are called in order of derivation: B1, D1, D2
- Destructor functions are called in reverse order
-



2. When a derived class directly inherits multiple base classes...

- Access_Specifiers { public, private, protected } can be different
- Constructors are executed in the order left to right, that the base classes are specified.
- Destructors are executed in the opposite order.

```
class Derived_Class_Name: access Base1,  
                        access Base2,... access BaseN  
{  
    //.. body of class  
} end Derived_Class_Name
```



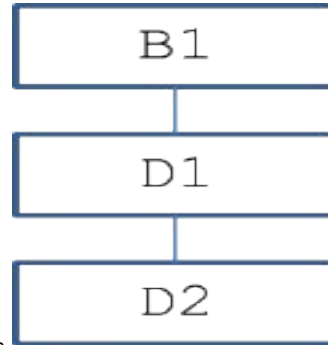
C++ Inheritance

- Derived class inherits a class derived from another class.

```
class B1 {
    int A;
public:
    B1(int Z) { A = Z; }
    int GetA() { return A; }
};

class D1 : public B1 { int B;
public:
    D1(int Y, int Z) : B1 (Z) { B = Y; }
    void GetB() { return B; }
};

Class D2 : public D1 { int C;
public:
    D2 (int X, int Y, int Z) : D1 ( Y, Z) { C = X; }
    void ShowAll () {
        cout << GetA() << " " << GetB() << " " << C << endl; }
};
```



```
int main() {
    D2 Ob(5,7,9);

    Ob.ShowAll();

    // GetA & GetB are still public here
    cout << Ob.GetA() << " "
        << Ob.GetB() << endl;

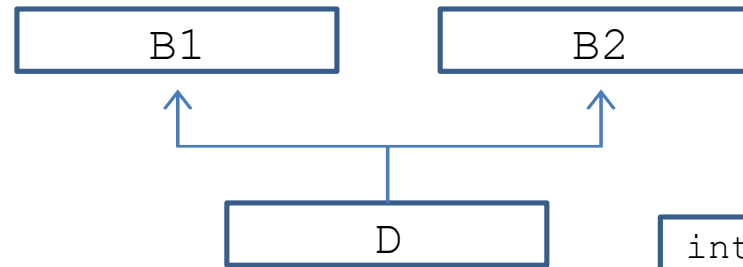
    return 0;
} // end main
```

*Because bases are inherited as public,
D2 has access to public elements of both B1 and D1*

C++ Inheritance

Derived Class Inherits Two Base Classes

```
class B1 {  
    int A;  
public:  
    B1(int Z) { A = Z;}  
    int GetA() { return A; }  
};  
class B2 {  
    int B; public:  
    B2 (int Y) { B = Y; }  
    void GetB() { return B; }  
};  
class D : public B1, public B2 {  
    int C;  
public:  
    D (int X, int Y, int Z) : B1(Z), B2 (Y) { C = X; }  
    void ShowAll () {  
        cout << GetA() << " " << GetB() << " " << C << endl; }  
};
```

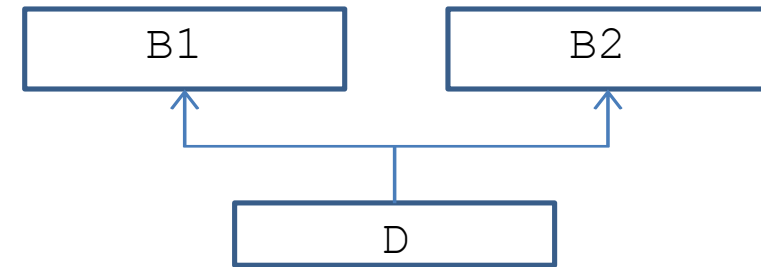


```
int main() {  
    D Ob(5,7,9);  
  
    Ob.ShowAll();  
  
    return 0;  
} // end main
```

C++ Inheritance

- Inheritance Multiple Base Classes (constructor and destructor)

```
class B1 {
public:
    B1() {cout << "Constructing B1\n"; }
    ~B1() {cout << "Destructing B1\n"; }
};
class B2 {
public:
    B2() {cout << "Constructing B2\n"; }
    ~B2() {cout << "Destructing B2\n"; }
};
class D : public B1, public B2 {
public:
    D() {cout << "Constructing D\n"; }
    ~D() {cout << "Destructing D\n"; }
};
```



```
int main () { D
    ob; return 0;
} // end main
```

```
----OUTPUT----
Constructing      B1
Constructing      B2
Constructing D
Destructing D
Destructing B2
Destructing B1
```

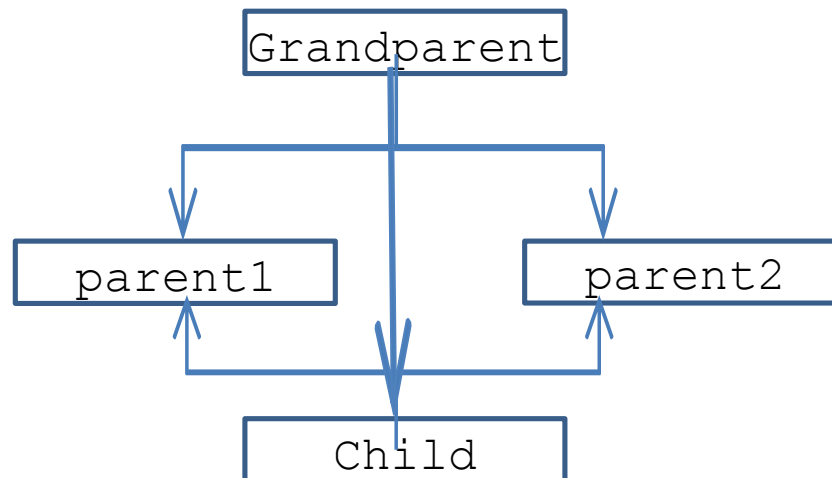
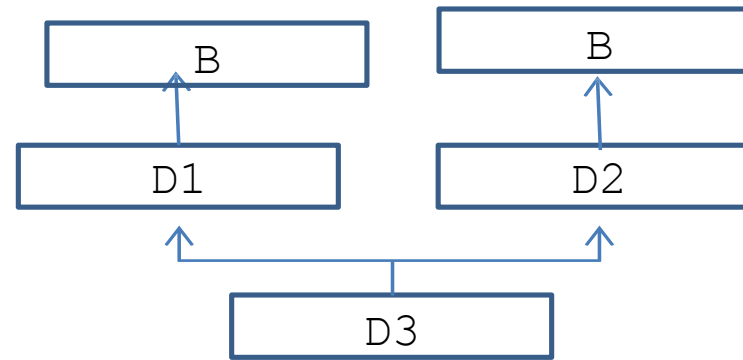
Virtual Base Class

PROBLEM:

The Base B is inherited twice by D3.

– There is ambiguity!

– Solution: mechanism by which only one copy of B will be included in D3.



Virtual Base Class

```
class B {
    public:
        int I;
};
class D1 : virtual public B {
    public:
        int J;
};
class D2 : virtual public B {
    public:
        int K;
};
class D3 : public D1, public D2
    public:
        int product {return I * J * K; }
};
```

```
int main() {
    D3 ob;

    ob.I = 15; //must be virtual
              // else compile
              // time error
    ob.J = 21;
    ob.K = 26;

    cout << "Product: "
          << ob.product() << endl;

    return 0;
} // end main
```

Virtual Base Class

- A Derived class does not inherit the constructors of its base class.
- Good Advice: You can and should include a call to one of the base class constructors when you define a constructor for a derived class.
- If you do not include a call to a base class constructor, then the default (zero argument) constructor of the base class is called automatically.
- If there is no default constructor for the base class, an error occurs.

- If the programmer does not define a **copy constructor** in a derived class (or any class), C++ will auto-generate a copy constructor for you. (Bit-wise copy)
- Overloaded assignment operators are not inherited, but can be used.
- When the destructor for the derived class is invoked, it auto- invokes the destructor of the base class. No need to explicitly call the base class destructor.

- A derived class inherits all the member functions (and member variables) that belong to the base class – except for the constructor.
- If a derived class requires a different implementation for an inherited member function, the function may be redefined in the derived class. (not the same *overloading*)
 - List its declaration in the definition of the derived class (even though it is the same as the base class).
 - Redefined function will have the same number and types of parameters. I.e. signature is the same.
 - Ok to use both (must use the base class qualifier to distinguish between the 2)

Virtual Functions

- **Virtual Functions**

- Background:

- A pointer declared as a pointer to a base class can also be used to point to any class derived from that base.
 - We can use a base pointer to point to a derived object, but you can access only those members of the derived object that were inherited from the base. The base pointer has knowledge only of the base class; it knows nothing about the members added by the derived class.
 - A pointer of the derived type cannot (should not) be used to access an object of the base class.

Virtual Functions

- Virtual Functions-
Background

```
class Base { int X;
public:
    void SetX(int I) { X = I;} int
    GetX() { return X;}
};
class Derived : public Base { int Y;
public:
    void SetY(int I) { Y = I;}
    int GetY() { return Y;}
};
```

```
int main() {
    Base *ptr;
    Base BaseOb;
    Derived DerivedOb;

    ptr = &BaseOb; ptr->SetX(15);
    cout <<"Base X: "
         << ptr->GetX() << endl;

    ptr = &DerivedOb;
    ptr->SetX(29);

    DerivedOb.SetY(42); // cannot use ptr cout <<
    "Derived Object X: "
        << ptr->GetX() << endl;
    cout << "Derived Object Y: "
         << DerivedOb.GetY() << endl;

    return 0;
} // end main
```

Virtual Functions

- **late binding** or **dynamic binding**
- When the programmer codes “virtual” for a function, the programmer is saying, “I do not know how this function is implemented”.
 - Technique of waiting *until runtime* to determine the implementation of a procedure is called **late binding** or **dynamic binding**.
 - A virtual function is a member function that is declared within a base class and redefined by a derived class.
 - Demonstrates “One interface, multiple methods” philosophy that is polymorphism.
 - “Run-time polymorphism”- when a virtual function is called through a pointer.
 - When a virtual function is redefined by a derived class, the keyword `virtual` is not needed.
 - “A base pointer points to a derived object that contains a virtual function and that virtual function is called through that pointer, C++ determines which version of that function will be executed based upon the type of object being pointed to by the pointer.” Schildt

Virtual Functions

- **Virtual Functions**

- Exact same **prototype** (Override not Overload) Signature + return type
- Can only be class members
- Destructors can be virtual; constructors cannot.
- Done at runtime!
- *Late Binding*: refers to events that must occur at run time.
- *Early Binding*: refers to those events that can be known at compile time.

Virtual Functions

- Virtual Functions

Polymorphic class
contains a virtual
function.

```
class Base {
public:
    int I;
    base(int X) { I = X;}
    virtual void func() {
        cout << "Using Base version of func(): ";
        cout << I << endl;
    }
};

class D1 : public Base { public:
    D1(int X) : base(X) {}
    void func() {
        cout << "Using D1's version of func(): "; cout << I*I <<
        endl;
    }
};

class D2 : public Base { public:
    D2(int X) : base(X) {}
    void func() {
        cout << "Using D2's version of func(): "; cout << I+I
        << endl;
    }
};
```

```
int maint() {
    Base *ptr;
    Base BaseOb(10);
    D1 D1Ob(10);
    D2 D2Ob(10);

    ptr = &BaseOb;
    ptr->func(); // use Base's func()

    ptr = &D1Ob;
    ptr->func(); // use D1's func()

    ptr = &D2Ob;
    ptr->func(); // use D2's func()

    return 0;
}
```

----OUTPUT----

Using Base version of func(): 10 Using D1's
version of func(): 100 Using D2's version
of func(): 20

*If the derived class does not override a virtual function,
the function defined within its base class is used.*

Virtual Functions

```
class Area {
    double dim1, dim2; public :
    void SetArea(double d1, double d2) {    dim1 = d1;
        dim2 = d2;
    }
    void GetDim (double &d1, double &d2) {    d1 = dim1;
        d2 = dim2;
    }
}
```

```
    virtual double GetArea() {
        cout << "DUMMY DUMMY OVERRIDE function";
        return 0.0;
    }
```

```
class Rectangle : public Area { public
:
```

```
    double GetArea() {
        double temp1, temp2  GetDim
        (temp1, temp2); return
        temp1 * temp2;
    }
};
```

```
class Triangle : public Area { public :
    double GetArea() { double temp1,
        temp2  GetDim (temp1, temp2);
        return 0.5 temp1 * temp2;
    }
};
```

```
int main () {
    Area *ptr;
    Rectangle R;
    Triangle T;

    R.SetArea(3.3, 4.5);
    T.SetArea(4.0, 5.0);

    ptr = &R;
    cout << "RECTANGLE_AREA: "
        << ptr->GetArea() << endl;

    ptr = &T;
    cout << "TRIANGLE_AREA: "
        << ptr->GetArea() << endl;

    return 0;
} // end main
```

*When there is no meaningful action for a base class virtual function to perform, the implication is that any derived class **MUST** override this function. C++ supports **pure virtual functions** to do this.*

```
virtual double GetArea() = 0; // pure virtual
```

Virtual Functions

- **Virtual Functions**

- When a class contains at least one *pure* virtual function, it is referred to as an *abstract class*.
- An abstract class contains at least one function for which no body exists, so an abstract class exists mainly to be inherited.
- Abstract classes do not stand alone.
- If Class B has a virtual function called f(), and D1 inherits B and D2 inherits D1, both D1 and D2 can override f() relative to their respective classes.