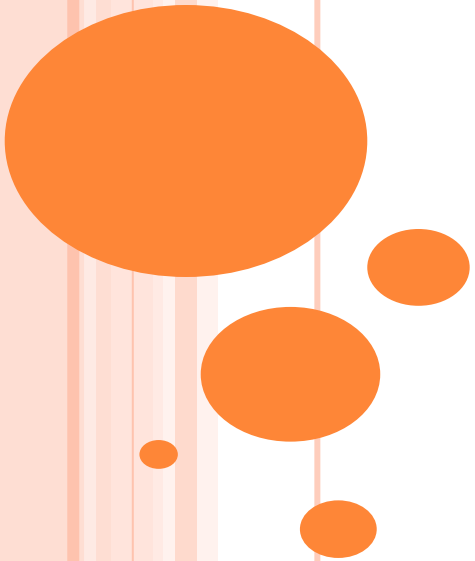


CSE 107: Object Oriented Programming Language

Dr. Tanzima Hashem
Professor
CSE, BUET

Inheritance

- Inheritance allows one object to inherit member variables and/or member functions from another object

- Inherited object is called base object
- Inheriting object is called derived object

- Helps programmer to write reusable code
- Helps programmer to write compact code

Inheritance

- Inheritance starts with defining the base class first

```
class base-class-name {
        .......
};
```

- Derived class is then defined using the base class
- The general form of deriving a class from another class is as follows:

```
class derived-class-name: access base-class-name {
        .........
};
```

- Access can be either private or public or protected
- Default access is private for derived class, public for derived structure

Visibility of base class members in derived class

Member access specifier in base class	Member visibility in derived class		
	Type of Inheritance		
	Private	Protected	Public
Private			
Protected			
Public			

Visibility of base class members in derived class

Member access specifier in base class	Member visibility in derived class		
	Type of Inheritance		
	Private	Protected	Public
Private	Not Inherited	Not Inherited	Not Inherited
Protected	Private	Protected	Protected
Public	Private	Protected	Public

Inheritance: Base Class

```
#include <iostream>
using namespace std;
class Rectangle {
        int width, length;
public:
        void set_width(int w) {width=w;}
        void set_length(int l) {length=l;}
        int area() {return (width*length);}
};
```

Inheritance: Derived Class

```
class Box: public Rectangle {
        int height;
public:
        void set_height (int h){height=h;}
        int volume () {return (area()*height);}
/ *private members of the base class cannot be accessed by the derived class*/
        //int volume () {return (width*length*height);}
};
```

INHERITANCE:

```
int main(){
    Rectangle rect;
    Box box;

    rect.set_width(3);
    rect.set_length(4);

    box.set width(4); //inherited
    box.set_length(4); //inherited
    box.set_height(5);

    cout<<"Rectangle area: "<<rect.area()<<endl;
    cout<< "Box base area: base area: "<<box.area()<<endl; //inherited
    cout<<"Box volume: "<<box.volume()<<endl;
    return 0;
}
```

Inheritance: Derived Class

```
class Box: private Rectangle {
        int height;
public:
        void set_height (int h){height=h;}
        int volume () {return (area()*height);}
};
```

INHERITANCE

```
int main(){
    Rectangle rect;
    Box box;

    rect.set_width(3);
    rect.set_length(4);

    //box.set width(4);
    //box.set_length(4);
    box.set_height(5);

    cout<<"Rectangle area: "<<rect.area()<<endl;
    /* function area() is private to Box object cannot be accessed outside Box object */
    //cout<< "Box base area: base area: "<<box.area()<<endl;
    cout<<"Box volume: "<<box.volume()<<endl;
    return 0;
}
```

Reminder: private modifier means that it can't be accessed from outside the class. But can be from inside(if it was public or protected)

Inheritance: Base Class

```
#include <iostream>
using namespace std;
class Rectangle {
protected:
        int width, length;
public:
        void set_width(int w) {width=w;}
        void set_length(int l) {length=l;}
        int area() {return (width*length);}
};
```

Inheritance: Derived Class

```
class Box: public Rectangle {
        int height;
public:
        void set_height (int h){height=h;}
/*protected members of the base class can be accessed by the derived class */
        int volume () {return (width*length*height);}
};
```

INHERITANCE

```
int main(){
    Rectangle rect;
    Box box;

    //rect.width=4; error!
    rect.set_width(3);
    rect.set_length(4);

    //box.width=4; error!
    box.set width(4); //inherited
    box.set_length(4); //inherited
    box.set_height(5);

    cout<<"Rectangle area: "<<rect.area()<<endl;
    cout<< "Box base area: "<<box.area()<<endl; //inherited
    cout<<"Box volume: "<<box.volume()<<endl;
    return 0;
}
```

Constructors, Destructors, and Inheritance

- Both base class and derived class can have constructors and destructors

- Constructor functions are executed in the order of derivation
- Destructor functions are executed in the reverse order of derivation

- While working with an object of a derived class, the base class constructor and destructor are always executed no matter how the inheritance was done (private, protected or public)

Constructors, Destructors, and Inheritance

```
class base {
public:
  base() {
    cout << "Constructing base class\n";
  }
  ~base() {
    cout << "Destructing base class\n";
  }
};
class derived : public base {
public:
  derived() /*1*/{
    cout << "Constructing derived class\n"; //2
  }
  ~derived() {
    cout << "Destructing derived class\n";
  }
};
```

```
void main() {
  derived obj;
}
```

Output:

```
Constructing base class //1
Constructing derived class //2
Destructing derived class
Destructing base class
```

Constructors, Destructors, and Inheritance

- If a base class constructor takes parameters then it is the responsibility of the derived class constructor(s) to collect them and pass them to the base class constructor using the following syntax -
  - **derived-constructor(arg-list) : base(arg-list) { … }**
  - Here “base” is the name of the base class

- It is permissible for both the derived class and the base class to use the same argument
- It is also possible for the derived class to ignore all arguments and just pass them along the base class.

Constructors, Destructors, and Inheritance

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

class Rectangle {
        int *width, *length;
public:
        Rectangle();
        Rectangle(int, int);
        ~Rectangle ();
        int area () {return (*width * *length);}
};

Rectangle::Rectangle () {
        width= new int;
        length = new int;
        *width = 0;
        *length = 0;
}
```

Constructors, Destructors, and Inheritance

```
Rectangle::Rectangle (int a, int b) {
        width= new int;
        length = new int;
        *width = a;
        *length = b;
}

Rectangle:: ~Rectangle () {
        delete width;
        delete length;
}
```

Constructors, Destructors, and Inheritance

```
class Box: public Rectangle{
        int *height;
public:
        Box();
        Box(int, int, int);
        ~Box();
        int volume(){ return area()*(*height);}
};

Box::Box ()
{
        height=new int;
        *height=0;
}
```

Constructors, Destructors, and Inheritance

```
Box::Box (int w, int l, int h):Rectangle(w,l) {
        height=new int;
        *height=h;
}

Box::~Box () {
        delete height;
}
```

Constructors, Destructors, and Inheritance

```
int main(){
        Rectangle rect(3,4);
        Box box (3,4,5);

        cout<<“Rectangle area: ”<<rect.area()<<endl;
        cout<< “Box base area: base area:”<<box area()<<endl;
        cout<<“Box volume: ”<<box.volume()<<endl;

        return 0;
}
```

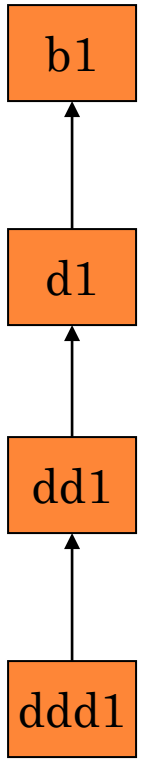
Constructors, Destructors, and Inheritance

```
class MyBase {
public:
  int x;
  MyBase(int m) { x = m; }
};
class MyDerived : public MyBase {
public:
  int y;
  MyDerived() : MyBase(0) { y = 0; }
  MyDerived(int a) : MyBase(a)
  {
    y = 0;
  }
  MyDerived(int a, int b) : MyBase(a)
  {
    y = b;
  }
};
```

```
void main() {
  MyDerived o1; // x = 0, y = 0
  MyDerived o2(5); // x = 5, y = 0
  MyDerived o3(6, 7); // x = 6, y = 7
}
```

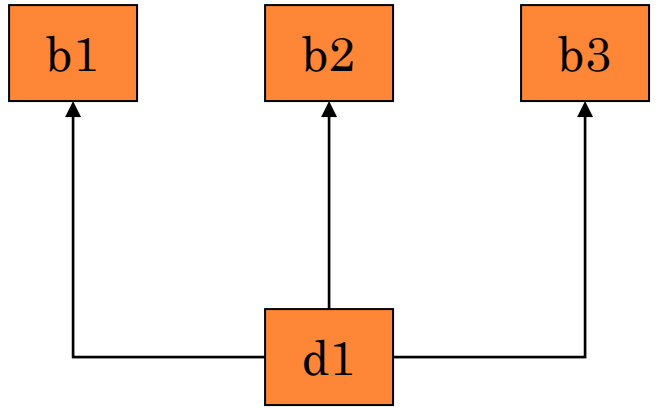
Multiple Inheritance

- A derived class can inherit more than one base class in two ways
  - By a chain of inheritance
    - b1 -> d1 -> dd1 -> ddd1 -> …
    - Here b1 is an indirect base class of both dd1 and ddd1
    - Constructors are executed in the order of inheritance
    - Destructors are executed in the reverse order



Multiple Inheritance

- A derived class can inherit more than one base class in two ways
  - By directly inheriting more than one base class
    - class d1 : *access* b1, *access* b2, …, *access* bN { … }
    - Constructors are executed in the order, left to right, that the base classes are specified
    - Destructors are executed in the reverse order



Multilevel Class Hierarchy

```
class Point{
        double x;
        double y;
public:
        Point(double x, double  y){
                this->x=x;
                this->y=y;
        }
        void get_xy(double &x, double &y){
                x=this->x;
                y=this->y;
        }
}
```

Multilevel Class Hierarchy

```
class Circle: public Point{
protected:
        double rad;
public:
        Circle(double x, double y, double r);
        double area(){return 3.14*rad*rad;}
}

Circle::Circle(double x, double y, double r):Point(x,y){
        rad=r;
}
```

Multilevel Class Hierarchy

```
class Cylinder: public Circle{
        double height;

public:
        Cylinder(double x, double y, double r, double h);
        double volume(){return 3.14*rad*rad*height;}
}

Cylinder::Cylinder(double x, double y, double r, double h):Circle(x,y,r){
        height=h;
}
```

Multiple Direct Inheritance

```
class Point{
        double x;
        double y;
public:
        Point(double x, double y){this->x=x; this->y->y;}
        void get_xy(double &x, double &y){x=this->x, y=this->y;}
}
```

Multiple Direct Inheritance

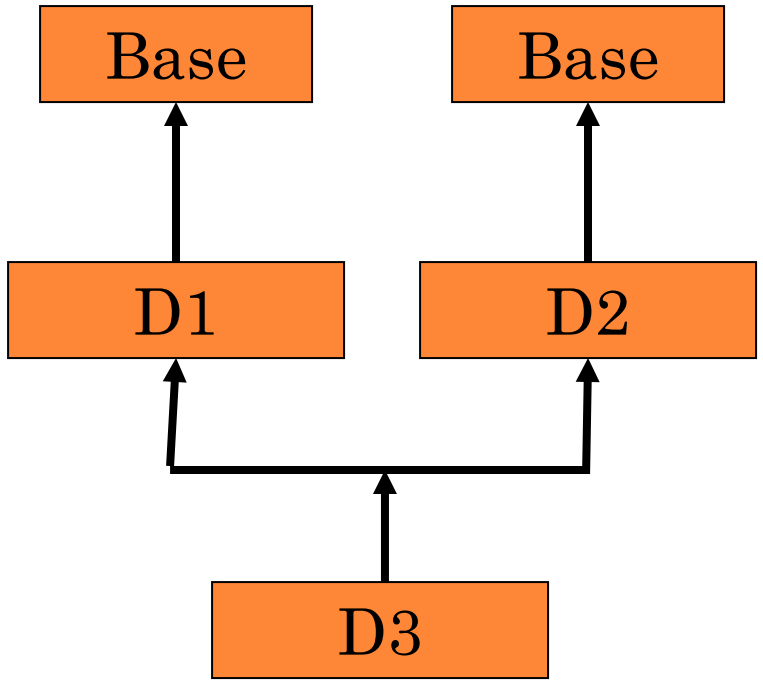
```
class Circle{
protected:
        double rad;
public:
        Circle(double r) {rad=r;}
        double area(){return 3.14*rad*rad;}
}
```

Multiple Direct Inheritance

```
class Cylinder: public Point, public Circle{
        double height;
public:
        Cylinder(double x, double y, double r, double h);
        double volume(){ return 3.14*rad*rad*height;}
}
Cylinder::Cylinder(double x, double y, double r, double h):Point(x,y),Circle(r)
{
        height=h;
}
```

Multiple Inheritance: Virtual Base Class

- Virtual Base Class prevents a derived class to inherit more than one copy of the base class
- This may happen when a derived class directly inherits two base classes and these base classes are also derived from another common base class



Multiple Inheritance: Virtual Base Class

```
class Base {
public:
   int i;
};
class D1 : public Base {
public:
   int j;
};
class D2 : public Base {
public:
   int k;
};
```

```
class D3 : public D1, public D2 {
  // contains two copies of 'i'
};

void main() {
   D3 obj;
   obj.i = 10; // ambiguous, compiler error
   obj.j = 20; // no problem
   obj.k = 30; // no problem
   obj.D1::i = 100; // no problem
   obj.D2::i = 200; // no problem
}
```

Multiple Inheritance: Virtual Base Class

```
class Base {
public:
   int i;
};
class D1 : virtual public Base {
public:
   int j;
}; // activity of D1 not affected
class D2 : virtual public Base {
public:
   int k;
}; // activity of D2 not affected
```

```
class D3 : public D1, public D2 {
  // contains only one copy of 'i'
}; // no change in this class definition

void main() {
   D3 obj;
   obj.i = 10; // no problem
   obj.j = 20; // no problem
   obj.k = 30; // no problem
   obj.D1::i = 100; // no problem, overwrites '10'
   obj.D2::i = 200; // no problem, overwrites '100'
}
```

Function Overriding

- Giving new implementation of base class method into derived class is called function overriding
- Signature of base class method and derived class must be same Signature involves:
  - Number of arguments
  - Type of arguments
  - Sequence of arguments

Function Overriding

```
#include <iostream>
using namespace std;
class Point{
protected:
        double x;
        double y;
public:
        Point() {x=0.0; y=0.0;}
        Point(double x, double y);
        double area(){return 0;}
};

Point::Point(double x, double y){
        this->x=x; this->y=y;
}
```

Function Overriding

```
class Circle: public Point{
protected:
        double rad;
public:
        Circle(){rad=0.0;}
        Circle(double x double y double r);
        double area();
};

Circle::Circle(double x, double y, double r) : Point(x,y) {
        rad=r;
}
double Circle::area(){
        return 3.14*rad*rad;
}
```

Function Overriding

```
class Cylinder:public Circle{
        double height;
public:
        Cylinder(){height=0.0;}
        Cylinder(double x, double y, double r, double h);
        double area();
};
Cylinder::Cylinder(double x, double y, double r, double h): Circle(x,y,r){
        height=h;
}

double Cylinder::area(){
        return 3.14*rad*rad*height;
}
```

Function Overriding

```
int main(){

Point p(1.0, 1.0);
Circle c(1.0, 1.0, 3.0);
Cylinder cl(1.0, 1.0, 3.0, 2.0);

cout<<"The area of the point is: "<<p.area()<<endl;
cout<<"The area of the circle is: "<<c.area()<<endl;
cout<<"The area of the cylinder is: "<<cl.area()<<endl;

return 0;
}
```

Polymorphism in C++

- Compile time polymorphism
  - Uses static or early binding
  - Example: function and operator overloading
- Run time polymorphism
  - Uses dynamic or late binding
  - Example: virtual functions

Early Binding vs. Late Binding

- Early binding
  - Normal functions, overloaded functions, nonvirtual member and friend functions
  - Resolved at compile time
  - Very efficient
  - But lacks flexibility
- Late binding
  - Virtual functions accessed via a base class pointer
  - Resolved at run-time
  - Quite flexible during run-time
  - But has run-time overhead; slows down program execution

Pointers to Derived Classes

- C++ allows base class pointers to point to derived class objects
- Let we have –
  - **class base { … };**
  - **class derived : public base { … };**
- Then we can write –
  - **base *p1; derived d_obj; p1 = &d_obj;**
  - **base *p2 = new derived;**

Pointers to Derived Classes

- Using a base class pointer (pointing to a derived class object) we can access only those members of the derived object that were inherited from the base
- This is because the base pointer has knowledge only of the base class
- It knows nothing about the members added by the derived class

Pointers to Derived Classes

```
class base {
public:
   void show() {
      cout << "base\n";
   }
};
class derived : public base {
public:
   void show() {
      cout << "derived\n";
   }
};
```

```
void main() {
   base b1;
   b1.show(); // base
   derived d1;
   d1.show(); // derived
   base *pb = &b1;
   pb->show(); // base
   pb = &d1;
   pb->show(); // base
}
```

- All the function calls here are statically/early bound

Pointers to Derived Classes

- While it is permissible for a base class pointer to point to a derived object, the reverse is not true

```
base b1;
derived *pd = &b1; // compiler error
```

- We can perform a downcast with the help of type-casting, but should use it with caution

- Pointer arithmetic is relative to the data type the pointer is declared as pointing to
- If we point a base pointer to a derived object and then increment the pointer, it will not be pointing to the next derived object
- It will be pointing to (what it thinks is) the next base object !!!

Virtual Function

- A virtual function is a member function that is declared within a base class and redefined (called ***overriding***) by a derived class
- It implements the "one interface, multiple methods" philosophy that underlies polymorphism
- The keyword **virtual** is used to designate a member function as virtual
- Supports run-time polymorphism with the help of base class pointers

Virtual Function

- While redefining a virtual function in a derived class, the function signature must match the original function present in the base class
  - So, we call it overriding, not overloading
- When a virtual function is redefined by a derived class, the keyword virtual is not needed (but can be specified if the programmer wants)
- The “virtual”-ity of the member function continues along the inheritance chain
- A class that contains a virtual function is referred to as a polymorphic class

Virtual Function

```
class base {
public:
   virtual void show() {
      cout << "base\n";
   }
};
class derived : public base {
public:
   void show() {
      cout << "derived\n";
   }
};
```

```
void main() {
   base b1;
   b1.show(); // base - (s.b.)
   derived d1;
   d1.show(); // derived – (s.b.)
   base *pb = &b1;
   pb->show(); // base - (d.b.)
   pb = &d1;
   pb->show(); // derived (d.b.)
}
```

- Here,
  - s.b. = static binding
  - d.b. = dynamic binding

Virtual Function

```
class base {
public:
   virtual void show() {
      cout << "base\n";
   }
};
class d1 : public base {
public:
   void show() {
      cout << "derived-1\n";
   }
};
```

```
class d2 : public base {
public:
   void show() {
      cout << "derived-2\n";
   }
};
void main() {
   base *pb; d1 od1; d2 od2;
   int n;
   cin >> n;
   if (n % 2) pb = &od1;
   else pb = &od2;
   pb->show(); // guess what ??
}
```

Virtual Destructors

- Constructors cannot be virtual, but destructors can be virtual
- It ensures that the derived class destructor is called when a base class pointer is used while deleting a dynamically created derived class object

Virtual Destructors

```
class base {
public:
  ~base() {
     cout <<  "destructing base\n";
  }
};
class derived : public base {
public:
  ~derived() {
     cout << "destructing derived\n";
  }
};
```

```
void main() {
   base *p = new derived;
   delete p;
}
```

- Output:
  - destructing base

Virtual Destructors

```
class base {
public:
  virtual ~base() {
    cout <<  "destructing base\n";
  }
};
class derived : public base {
public:
  ~derived() {
    cout << "destructing derived\n";
  }
};
```

```
void main() {
  base *p = new derived;
  delete p;
}
```

- Output:
  - destructing derived
  - destructing base

Pure Virtual Functions

- If we want to omit the body of a virtual function in a base class, we can use pure virtual functions
  - **virtual ret-type func-name(param-list) = 0;**
- It makes a class an ***abstract class***
  - We cannot create any objects of such classes
- It forces derived classes to override it
  - Otherwise they become abstract too
- We can still create a pointer to an abstract class
  - Because it is at the heart of run-time polymorphism

Acknowledgement
http://faizulbari.buet.ac.bd/Courses.html
http://mhkabir.buet.ac.bd/cse201/index.html

The End

Topic Covered: Chapter 7 + Chapter 10