# UEE1303(1070) S'12 Object-Oriented Programming in C++

## Lecture 07: *Inheritance (I) – Basics & Single Inheritance*

## Learning Objectives

- Concepts of Inheritance
  - as an is-a relationship
- How to publicly derive one class from another
  - constitution of a derived class
  - protected access
- Constructors/Destructors of Derived Class
  - public, private and protected inheritance
- Assignment operator and copy constructors
  - Conversion of Base/Derived Classes

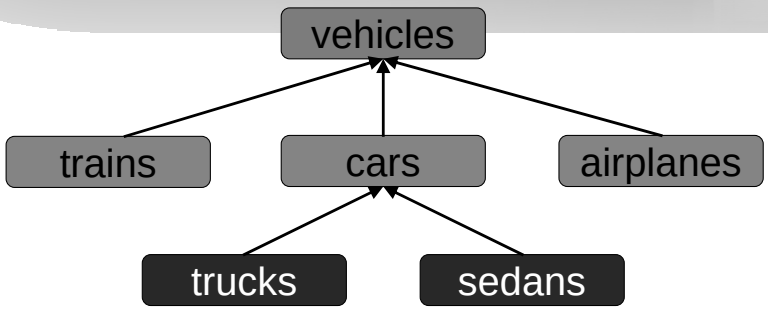


## Fundamentals of Inheritance

- *Code reusability* is the key to enable object-oriented programming (OOP)
  - C++ implements *inheritance*
  - establish parent-child relationship among existing classes
- ***Inheritance*** (a.k.a. ***derivation***) is a relationship between classes
  - one class inherits the properties (attributes and behaviors) of another class
  - is an *is-a* relationship
  - enable a hierarchy of classes (from the most *general* to the most *specific*)



## Example: Hierarchy of Vehicles




- A vehicle contains its *weight*, *seats*, *loading driving*, and etc.
- A car can be defined by either a ***redefinition*** or adding ***more properties*** based on vehicle

## Class Inheritance

- A new class can be derived from an old class
  - **inherit** attributes/methods from the old class
  - **create** new attributes/methods
  - called *inheritance* and *derivation*
- **Base** (super, parent) class v.s. **derived** (sub, child) class
  - base class is the class to be inherited
  - derived class is the new class on top of a base class
  - A base class can have *multiple* derived classes



## Declaration of Inheritance

- Base class defines
  - inheritable properties for derived classes
  - non-inheritable private properties of its own
- General form of a derived class

```
class ⟨Derived Class⟩:
    ⟨Access Specifier 1⟩⟨Base Class 1⟩,
    ⟨Access Specifier 2⟩⟨Base Class 2⟩,
      ...
{
    //specify properties of its own
};
```

  - only one "**:**"
  - if ≥2 base classes ⇨ *multiple inheritance*
  - default as *private* access



## Example of Inheritance (1/4)

- Class `CCircle` and its definition

lec7-1.cpp

```
class CCircle {
protected:
    double radius;
public:
    CCircle(double r=1.0): radius(r) {;}
    void setR(double r=1.0) { radius=r;}
    double calVol() const {
        return (PI*radius*radius);
    }
    void showVol() const {
        cout << "radius=" << radius
             << endl;
        cout << "volume=" << calVol()
             << endl;
    }
};
```



## Example of Inheritance (2/4)

- Class `CCylinder` and its definition

lec7-1.cpp

```
class CCylinder: public CCircle {
protected:
    double length; //add one data member
public:
    CCylinder(double r=1.0, double l=1.0):
        CCircle(r), length(l) { ; }
    void setRL(double r=1.0, double l=1.0){
        radius = r; length = l; }
    double calVol() { //compute volume
        return (CCircle::calVol()*length); }
    //inherit showVol() from CCircle
    void displayVol() const {
        cout<<"d radius="<<radius<<endl;
        cout<<"d volume="<<calVol()<<endl; }
};
```

## Example of Inheritance (3/4)

- `main` program

lec7-1.cpp

```
int main() {
    CCircle cr1, cr2(4);
    CCylinder  cy1, cy2(2,3);

    cr1.setR(2);
    cr1.showVol(); cr2.showVol();
    cy1.setRL(3);
    cy1.showVol(); cy1.displayVol();
    cy2.showVol(); cy2.displayVol();

    cr1 = cy1; //what happen?
    cr1.showVol();

    return 0;
}
```



## Example of Inheritance (4/4)

- Execution result

```
>./lec7-1
radius=2 volume=12.56
radius=4 volume=50.24
radius=3 volume=28.26
d radius=3 d volume=28.26
radius=2 volume=12.56
d radius=2 d volume=37.68
radius=3 volume=28.26
>
```



## Derived Classes

- A derived class can change
  - add ***new*** members (either data or function)
  - *overload* or *overide* (*redefine*) member functions in the base class
  - change the accessibility of members of the base class in the derived class
- A derived class cannot inherit
  - *constructors*/*destructors* of the base class
  - *friends* of the base class
  - *static* data members and *static* member functions



## Choose Access Specifier

- Different ways of inheritance affects how the members in the base class can be accessed from the derived class
- The access specifier defines the way of inheritance
  - public inheritance (public derivation)
  - private inheritance (private derivation)
  - protected inheritance (protected derivation)

# public Inheritance (1/2)

- Public inheritance is the most common one
  - maintain the accessibility of the base class
- Public and protected members of the base class remain the same accessibility in the derived class
  - member functions of the derived class *can access public and protected* members of the base class directly
  - *cannot access* the *private* members of the base class
- An object of the derived class can only access the *public members* of the base class



# public Inheritance (2/2)

- *Public inheritance* defines
  - the inheriting class as *public* derived class
  - the inherited class as *public* base class
- Accessibility of members in the base class

| members in public base class | accessibility in public derived class |
|---|---|
| **public** | **public** |
| **private** | **inaccessible** |
| **protected** | **protected** |

  - why cannot access private members?
    - ⇨ maintain *encapsulation* of the base class



# Example of public Inheritance (1/3)

lec7-2.cpp

```
class CPoint   //base class
{
    double x, y;
public:
    CPoint(double a=0, double b=0):
           x(a), y(b) {}
    void SetPoint(double a=0, double b=0) {
        x=a; y=b;                          }
    void MovePoint(double dx, double dy) {
        x +=dx; y += dy;                   }
    double GetX() const { return x; }
    double GetY() const { return y; }
};
```



# Example of public Inheritance (2/3)

```
class CRect: public CPoint { //derived class
//private:
    double h, w; //new private data
public:
    //inherit member functions from CPoint
    double GetH() const { return h; }
    double GetW() const { return w; }
    CRect(double a, double b,
          double c, double d):
         h(c), w(d) { SetPoint(a,b); }
    void SetRect(double a, double b,
                 double c, double d) {
        SetPoint(a,b);
        h = c; w = d;                  }
 };
```

lec7-2.cpp

## Example of public Inheritance (3/3)

lec7-2.cpp

```
int main()
{
    CRect cr1;
    cr1.SetRect(2,3,20,10);
    //visit base public members through
    //a derived object
    cr1.MovePoint(3,2);
    cout<<cr1.GetX()<<','<<cr1.GetY()<<','
        <<cr1.GetH()<<','<<cr1.GetW()<<endl;
    return 0;
}
```

```
5 5 20 10
```



## private Inheritance (1/2)

- *Public* and *protected* members of the base class become *private* in the derived class
  - member functions of the derived class still can access public and protected members of the base class directly
  - cannot access *private members* of the base class
- An object of the derived class cannot access any member (including public, protected and private) of the base class



## private Inheritance (2/2)

- *Private inheritance* defines
  - the inheriting class as private derived class
  - the inherited class as private base class
- Accessibility of members in the base class

| members in private base class | accessibility in private derived class |
|---|---|
| **public** | **private** |
| **private** | **inaccessible** |
| **protected** | **private** |

- Need not memorize the rules
  - inherit public/protected members as private



## Example of private Inheritance (1/3)

lec7-3.cpp

```
class CPoint //base class
{
    double x, y;
public:
    CPoint(double a=0, double b=0):
           x(a), y(b) {}
    void SetPoint(double a=0, double b=0) {
        x=a; y=b;                          }
    void MovePoint(double dx, double dy) {
        x +=dx; y += dy;                   }
    double GetX() const { return x; }
    double GetY() const { return y; }
};
```

## Example of private Inheritance (2/3)

lec7-3.cpp

```
class CRect: private CPoint //derived class
{
//private:
    double h, w; //new private data
public:
    double GetH() const { return h; }
    double GetW() const { return w; }
    //derived member functions can access
    //base member functions
    void SetRect(double a, double b,
                 double c, double d) {
        CPoint::SetPoint(a, b);
        h = c; w = d;                  }
    void MoveRect(double dx, double dy) {
        CPoint::MovePoint(dx, dy);      }
};
```



## Example of private Inheritance (3/3)

lec7-3.cpp

```
// in main()
    CRect cr2;
    cr2.SetRect(2,3,20,10);
    //a derived object cannot access any
    //member of the base class
    cr2.MovePoint(3,2); //illegal!!!
    //use its own member function
    cr2.MoveRect(3,2); //legal
    cout<<cr2.GetX()<<','<<cr2.GetY()<<','
        <<cr2.GetH()<<','<<cr2.GetW()<<endl;
    //what's wrong above?
```

```
5 5 20 10
```



## protected Inheritance (1/2)

- *Public* and *protected* members of the base class become *protected* in the derived class
  - member functions of the derived class still can access public and protected members of the base class directly
  - cannot access private members of the base class
- An object of the derived class *cannot access* any member (including public, protected and private) of the base class



## protected Inheritance (2/2)

- Protected inheritance defines
  - the inheriting class as protected derived class
  - the inherited class as protected base class
- Accessibility of members in the base class

| members in protected base class | accessibility in protected derived class |
|---|---|
| **public** | **protected** |
| **private** | **inaccessible** |
| **protected** | **protected** |

- Need not memorize the rules
  - inherit public/protected members as protected

## protected Members

- For the base class where a protected member is defined,
  - the protected member works like a private member outside the base class
- For the derived class who inherited the protected member,
  - the protected member works like other public members
- For an object of the base class, protected members cannot be accessed



## Example of protected Members

lec7-4.cpp

```
class A
{
private: double x;
protected: long y;
public: short z;
};
class B: public A {
public:
    void f() { //what's wrong below?
        x = 3.0; y=4; z=2; }
};
int main() {
    B b;
    b.x = 1.3; //what's wrong?!
    b.y = 2;   //what's wrong?!
    b.z = 5;
}
```



## Example of protected Inheritance (1/2)

lec7-5.cpp

```
class CBase
{
    double x;
protected:
    int GetX() { return x; }
public:
    void SetX(double a) { x = a; }
    void ShowX() { cout << x << endl; }
};
class CDerived: protected CBase {
    double y; //its own private member
public:
    void SetY(double b) { y = b; }
    //visit protected member of CBase
    void SetY() { y = GetX(); }
    void ShowY() { cout << y << endl; }
};
```



## Example of protected Inheritance (2/2)

lec7-5.cpp

```
int main()
{
    CDerived cd;

    cd.SetX(15);  //what happen?
    cd.SetY(20);

    cd.ShowX();   //what happen?
    cd.ShowY();

    return 0;
}
```

- What's wrong with these calls?

## Compare Different Inheritances (1/2)

- Member functions of the derived class in different inheritances

| inheritance method | accessibility of member functions in the derived class |
|---|---|
| public | can access public/protected members of the base class |
| private | can access public/protected members of the base class |
| protected | can access public/protected members of the base class |



## Compare Different Inheritances (2/2)

- Objects of the derived class in different inheritances

| inheritance method | accessibility of objects |
|---|---|
| public | can access public members of both the derived and base classes |
| private | can access no member of the base class |
| protected | can access no member of the base class |



## Constructors of Derived Class

- Data members of a derived class consists of
  - old data members of the base class
  - new data members of the derived class
- **Constructor/destructor** cannot be inherited
  ⇨ need to initialize both old and new data members by constructor of the derived class
- Two steps to define a constructor of the derived class
  - call the constructor of the base class for old data members
  - set up new data members properly



## Constructors of Single Inheritance

- **Single inheritance** refers that the derived class has only one base class
  - data members do not include objects of the derived class
- Format: a base class `CB` and a derived class `CD`

```
CD(⟨CD_para_list⟩):CB(⟨CB_para_list⟩) {
    //initialize new data members in CD
}//this is CD's constructor
```

  - `⟨CD_para_list⟩ = ⟨CB_para_list⟩ + new data members`
  - call `CB`'s constructor

## Example of Single Inheritance (1/3)

lec7-6.cpp

```
class CPsn //base class
{
    string name; char sex;
public:
    CPsn(string sname, char s) {
        name = sname; sex = s; }
    ~CPsn() {}
    string GetName() { return name; }
    char GetSex() { return sex; }
};
class CStu : public CPsn { //derived class
    int age; string addr;
public:
    CStu(string sname, char s, int a,
         string ad): CPsn(sname, s)  {
        age=a; addr=ad;
    }
}
```



## Example of Single Inheritance (2/3)

lec7-6.cpp

```
    void ShowCStu() {
        cout << "name=" << GetName()
<< endl;
        cout << "sex=" << GetSex() <<
  endl;
        cout << "age=" << age << endl;
        cout << "address=" << addr <<
  endl;
    }
    ~CStu(){}
}
int main() {
    CStu s1("Bob",'M',20,"123 Ave A,
  NY");
    CStu s2("Jan",'F',18,"101 Ave C,
  AZ");
    s1.ShowCStu();
```



## Example of Single Inheritance (3/3)

- s1 assigns 5 arguments

```
CStu s1("Bob",'M',20,"123 Ave A, NY");
```

```
CStu(string sname, char s, int a, string ad):
                                  CPsn(sname, s)
```

- Alternative definition for the constructor

```
CStu(string sname, char s, int a, string ad):
          CPsn(sname, s), age(a), addr(ad) {}
```

- Release an `CStu` object
  - first call `~CStu()`, then call `~CPsn()`



## Destructor of Derived Class

- Destructor of the derived class also cannot be inherited
  - ⇨ need to be defined in the derived class
    - *automatically* (*implicitly*) call the destructor of the base class
- To invoke the destructor of the derived class, *opposite* of how constructors are called
- Example: A derived from B derived from C
  - 1st: class C destructor is executed
  - 2nd: class B destructor is executed
  - 3rd: class A destructor is executed

## Assignment and Copy Constructors

- Overloading assignment operators and copy constructors also cannot be not inherited!
  - but can be used in derived-class definitions
  - typically must be used
  - similar to how derived-class constructor invokes base-class constructor
- Example of assignment operator (=)

```
CDerived& CDerived::operator = (
                const CDerived& rHand) {
    CBase::operator=(rHand);
    //...
}
```



## Example of Copy Constructors

- If not defined, generate a default one
- After : is invocation of base copy constructor
  - set inherited data members of derived-class object being created
  - obj is of type CDerived
  - obj is also of type CBase ⇨ argument is valid ⇨ involve class conversion

```
Cderived::CDerived(const CDerived& obj):
                                CBase(obj) {
    //...
}
```



## Conversion of Base/Derived Classes

- An object of the derived class can be used as an object of the base class, but not in reverse
  - can assign the object of the derived class to the object of the base class
  - can use the object of the derived class to initialize a reference to the base class
  - pointer to the object of the derived class can assign to pointer to the object of the base class
- Through the base class, the pointer can only be used onto the inherited members



## Example of Class Conversion (1/2)

lec7-7.cpp

```
class B0
{
public: void Show() {
    cout << “B0::Show()” << endl; }
};
class B1 : public B0
{
public: void Show() {
    cout << “B1::Show()” << endl; }
};
class D2 : public B1
{
public: void Show() {
    cout << “D2::Show()” << endl; }
};
void fun(B0* ptr) {
    ptr->Show();
}
```

## Example of Class Conversion (2/2)

lec7-7.cpp

```
int main() {
    B0 r0;
    B1 r1;
    D2 r2;

    B0 *p;
    p = &r0; //B0 pointer to B0 object
    fun(p);
    p = &r1; //B0 pointer to B1 object
    fun(p);
    p = &r2; //B0 pointer to D2 object
    fun(p);

    return 0;
}
```

```
B0::Show()
B0::Show()
B0::Show()
```



## Summary (1/2)

- Inheritance provides code reuse
  - allow one class to "derive" from another, adding features
- Derived class objects inherit members of base class
  - and may add members
- Private member variables in base class cannot be accessed "by name" in derived
- Private member functions are not inherited



## Summary (2/2)

- Can redefine inherited member functions
  - to perform differently in derived class
- Protected members in base class:
  - can be accessed "by name" in derived class member functions
- Overloaded assignment operator not inherited
  - but can be invoked from derived class
- Constructors are not inherited
  - are invoked from derived class's constructor