

Introduction to Inheritance and OOP in C++

Sebti Foufou

NYU Abu Dhabi
sfoufou@nyu.edu

Inheritance and OOP

- Definition
- Compatibility between objects of base and derived classes
- Using attributes of base class inside derived class
- Overriding member functions
- Constructor, destructor and inheritance
- Copy constructor and inheritance
- Assignment operator and inheritance
- Protected data members
- Different types of inheritance
- Static type of objects
- Virtual functions and dynamic types
- Few remarks regarding virtual functions

Definition

Inheritance is a relation between an existing class, called base class (BC) or super class, and a new class, called derived class (DC) or sub class, which derives from the first one. e.g. below Pointcol is derived class. Point is base class.

```
class Point {  
private :  
    int x, y ;  
public :  
    void initialize(int,int);  
    void displace( int, int);  
    void print() ;  
} ;
```

```
class Pointcol : public Point{  
private :  
    int color ;  
  
public :  
    void initializec(int,int,int);  
    void printc() ;  
} ;
```

Definition

Inheritance is a relation between an existing class, called base class (BC) or super class, and a new class, called derived class (DC) or sub class, which derives from the first one. e.g. below Pointcol is derived class. Point is base class.

```
void main() {  
    Point p ;  
    Pointcol pc ;  
  
    p.initialize(3,5) ;  
    p.displace(1, -1) ;  
    p.print() ;  
  
    pc.initializec(3, 5, 1) ;  
    pc.displace(-1, -2) ;           // inherited from Point  
    pc.print() ;                   // inherited from Point  
    pc.printc() ;  
}
```

Inheritances of a derived class can call functions inherited from the base class.

Definition (2)

In this example:

- Every object instance of `Pointcol` is also a `Point` (but instances of `Point` are NOT instances of `Pointcol`)
- Every operation that can be applied on an instance of `Point` can be applied on instances of `Pointcol`
- Derived classes can change the definition of inherited functions (function overriding).

Compatibility between instances of BC and DC

Conversion of instances of BC and DC

```
Point p(17, 20) ;           // Assume 2 args constr. exists
Pointcol pc(17, 25, 2) ;    // Assume 3 args constr. exists
p = pc ;                    // ok
pc = p ;                    // Error
```

Statement `p = pc` is correct, it leads to the conversion of `pc` to the type `Point`, then the assignment of the result to `p`. However, statement `pc = p`; will generate a compiler error.

In the 1st case, the conversion is doable because every `Pointcol` is also a `Point`. In the 2nd case, the conversion can not be done because a `Point` is not a `Pointcol`.

Conversion of pointers to BC and DC

```
Point    p(17, 20),        *pp = &p ;
Pointcol pc(17, 25, 2),    *ppc = &pc ;
pp = ppc ;                 // ok
ppc = pp ;                 // Error
```

The assignment `pp = ppc` is legal as it leads to the conversion of `Pointcol*` to type `Point*`. The assignment `ppc = pp` is illegal unless we make an explicit cast :

```
ppc = (Pointcol *)pp ;    //Do not do that, unless !!!
```

Using the attributes of BC inside DC

- The DC doesn't have access to the private data members of its BC. In the below code, instruction `cout<<"Located in " <<x<<" "<<y<<endl;` will generate a compiler error because `x` and `y` are private data members of the class `Point`

```
void Pointcol::printc() {  
    cout<< "Located in " << x <<" " << y <<endl;  
    cout << "          Color = : " << color <<endl ;  
}
```

- The DC can access public members of its BC. A correct definition of the function `Pointcol::printc()` could be:

```
void Pointcol::printc() {  
    print() ;      //call Point::print() inherited from Point  
    cout << "          color = : " << color << endl ;  
}
```

- A member function of DC can not access private data members of its BC. However, it can access and use public data members and methods.
- Function `printc()` can not access and use attributes `x` and `y` inherited from `Point`. To display them, it calls function `print()` of `Point`. This function is visible to every other function as it is declared public inside the class `Point`.

Overriding inherited member functions

- In the class `Pointcol` we have 2 print methods:
 - `print()` inherited from `Point`
 - `printc()` newly defined within `Pointcol`
- Giving the same name to these 2 functions, means redefining (overriding) function `Point::print()` inside `Pointcol` to add the printing of the color.

```
void Pointcol::print() {  
    print() ;           // Line #1  
    cout << "          color = : " << color <<endl ;  
}
```

- Problem : We have recursive call to `Pointcol::print()`. The system doesn't understand that in Line #1 we want to call function `print()` of the `Point` class and not function `print()` of `Pointcol`!

- **Solution :** Use the class membership operator `::`.

```
void Pointcol::print() {  
    Point::print() ;  
    cout << "          color = " << color <<endl ;  
}
```


Overriding inherited member functions (2)

- The same reasoning applies for the function `initialize()`, which should also be replaced by overridden version of function `Point::initialize()`
- The redefinition of the function `displace()` is not useful as the displacement doesn't affect the color.

- Example :

```
void main() {  
    Pointcol pc ;  
  
    pc.initialize(10, 20, 2) ;  
    pc.print() ;           // call Pointcol::print  
    pc.Point::print() ;    // call Point::print  
    pc.displace() ;        // call Point::displace  
    pc.print() ;           // call Pointcol::print  
}
```

Constructor, destructor and inheritance

- Constructor:

In the previous example (`Point`, `Pointcol`), we know that to create an object of type `Pointcol`, we first create an object of type `Point`, hence call the constructor of the class `Point`, then add what is specific to `Pointcol` (the color).

- Destructor :

When an instance of `Pointcol` is destroyed the call to the destructor of `Point` is done implicitly to destroy `Point` object.

- Parameter passing between constructor of DC and BC:

```
class Point {  
public :  
    Point(int a, int b) ;  
    ...  
} ;  
Point::Point(int a , int b) {  
    x = a ;   y = b ;  
}
```

Constructor, destructor and inheritance (2)

```
class Pointcol : public Point {  
public :  
    Pointcol(int a, int b, int c) ;  
    ...  
} ;  
Pointcol::Pointcol(int a, int b, int c): Point(a, b) {  
    color = c ;  
}
```

The constructor of `Pointcol` passes-on its two arguments to the constructor of class `Point` to create the part `Point` of the object `Pointcol`.

Copy constructor and inheritance

Let B be a base class and D its derived class.

- If D doesn't define a copy constructor:

There will be a call to the default copy constructor of D. For the members inherited from B, there will be a call to the copy constructor of class B:

- if B has a copy constructor, it will be called,
- if B doesn't have a copy constructor, the default CC of B will be called.

- If D has defined a copy constructor of the form:

```
D::D(const D &arg) { ... }
```

There will be a call to the default constructor of B to initialize part B (see example 1 in the next slide)

- If D has defined a copy constructor of the form:

```
D::D(const D &arg) : B(arg) { ... }
```

There will be a call to the matching constructor of B to initialize part B (See example 2 in the next slide).

Copy constructor and inheritance (2)

```
class Point {
    int x, y ;
public :
    Point(int a=0, int b=0) {
        x = a ; y = b ;
        cout << " ++ Point " << x << " " << y << endl;
    }
    Point(const Point & pt) {
        x = pt.x ; y = pt.y ;
        cout << " CC Point " << x << " " << y << endl;
    }
} ;

class Pointcol : public Point {
    int color ;
public :
    Pointcol(int a=0, int b=0, int c=1):Point(a, b){
        color = c ;
        cout << " ++ Pointcol " << color << endl ;
    }
    Pointcol(const Pointcol & pt) {
        color = pt.color ;
        cout << " CC Pointcol " << color << endl ;
        Point(pt)
    }
} ;
```

```
void main() {
    Pointcol a(2,3,4);
    Pointcol b= a;
}
```

```
++ Point 2 3
++ Pointcol 4
++ Point 0 0
CC Pointcol 4
CC Point 2 3
```

Copy constructor and inheritance (3)

```
class Point {
    int x, y ;
public :
    Point(int a=0, int b=0) {
        x = a ; y = b ;
        cout << " ++ Point " << x << " " << y << endl;
    }
    Point(const Point & pt) {
        x = pt.x ; y = pt.y ;
        cout << " CC Point " << x << " " << y << endl;
    }
} ;

class Pointcol : public Point {
    int color ;
public :
    Pointcol(int a=0, int b=0, int c=1) : Point(a, b) {
        color = c ;
        cout << " ++ Pointcol " << color << endl ;
    }
    Pointcol(const Pointcol & pt) : Pointcol(pt) {
        color = pt.color ;
        cout << " CC Pointcol " << color << endl ;
    }
} ;
```

```
void main() {
    Pointcol a(2,3,4);
    Pointcol b= a;
}
```

```
++ Point 2 3
++ Pointcol 4
CC Point 2 3
CC Pointcol 4
```

Assignment operator and inheritance

Let D be a derived class and B its base class.

- If D doesn't overload the assignment operator:
The assignment of one instance of D to another will be done by the default assignment operator of D. Part B of D will be handled by the assignment operator of B (the overloaded one if any, otherwise the default one).
- If D has overloaded the assignment operator:
The operator should handle all the assignment steps including the assignment of the part inherited from B.

Assignment operator and inheritance (2)

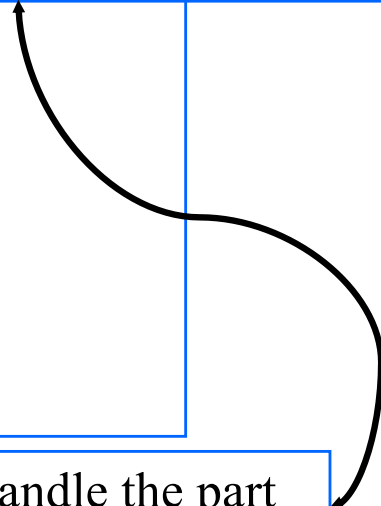
```
class Pointcol : public Point {
    ...
    // Assignment operator overloaded
    Pointcol & operator= (const Pointcol &pt) {
        couleur = pt.color ;
        cout << "Operator = of Pointcol" << endl ;
        return (*this) ;
    }
} ;

void main () {
    Pointcol p(1,3,10), q(4,9,20);

    cout << "p = " ; p.print() ;
    cout << "q before = " ; q.print() ;

    q = p ;
    cout << "q after = " ; q.print() ;
}
```

```
p = Pointcol 1  3  10
q before = Pointcol 4  9  20
Operator = of Pointcol
q after = Pointcol 4  9  10
```



The assignment operator of Pointcol has to be modified to handle the part Point of Pointcol objects. The above definition is **incorrect**.

Assignment operator and inheritance (3)

After correction :

```
class Pointcol : public Point {
    ...
    // redefined assignment operator
    Pointcol & operator = (const Pointcol & pt) {
        x = pt.x ;
        y = pt.y ;
        color = pt.color ;
        cout << "Operator = of Pointcol" << endl ;
        return (*this) ;
    }
} ;

void main () {
    Pointcol p(1,3,10), q(4,9,20);

    cout << "p = " ; p.print( ) ;
    cout << "q before  = " ; q.print( ) ;

    q = p ;
    cout << "q after   = " ; q.print( ) ;
}
```

```
p = Pointcol 1  3  10
q before = Pointcol 4 9 20
Operator = of Pointcol
q after = Pointcol 1 3 10
```

Protected data members

- There is a third way to control the access to the DM of a class: the `protected` access. The benefit of this new status is visible for the classes having derived classes.
- Protected data members remain inaccessible to the users of the class, but they are accessible to its derived classes.

```
class Point {
protected:
    int x, y ;    //x,y will be accessible to Pointcol
    ...
} ;
class Pointcol : public Point {
    int color ;
public:
    ...
    void print() {
        cout<<"I am in " <<x<<" " <<y<<"\n" ;
        cout<<«    my color is: "<<color<<endl ;
    }
};
```

x, y are protected. `Pointcol::print()` can now access to these attributes. Therefore, no need to call `Point::print()` to display these attributes.

The different types of inheritance

- Until now we only used `public` inheritance. Two other types of inheritance are available, but they are not commonly used. These are `private` and `protected` inheritance.
- The below table summarizes access rights to DM of the BC for the member functions of its derived classes (MFDC) and for the users of the derived class (UDC) in case of `public` inheritance:

Status in BC	Access FMDC	Access UDC	Status in DC
public	yes	yes	public
protected	yes	no	protected
private	no	no	private

static binding of objects

Definition: We refer to static binding (or static linkage) when the list of methods that an object can call is defined by the compiler during the compiling time according to the type of the object.

```
void main() {
    Point    p(3,5) , *pp ;
    Pointcol pc(8,6, 3), *ppc ;

    pp = &p ;
    ppc = &pc ;
    pp->print() ;           // Point::print()
    ppc->print() ;          // Pointcol::print()

    pp = ppc ;              // Line #10
    pp->print() ;            // Line #11, Point::print()
}
```

pp is of type `Point*`, even if this pointer is pointing to a `Pointcol` (Line #10), it can only call the methods of the class `Point` (static binding).
The instruction `pp->print()` in Line #11 calls method `print()` of `Point` and not `print()` of `Pointcol`.

Virtual functions and dynamic binding

To allow an object to call methods according to its current state and not according to its type (dynamic binding), we need to use virtual functions (VF). Keyword `virtual`

In the previous example, with the mechanism of VF the instruction `pp->print()` ; will not systematically call the function `Point::print()`, but the function `print()` of the type of the object currently pointed by `pp`.

For this, we only need to declare `print()` as a virtual function:

```
class Point {  
    ...  
public:  
    virtual void print() ;  
} ;  
  
void Point::print() { ... }
```

This can be useful if you're using
a & or *...

Virtual functions and dynamic binding (2)

```
class Point {
protected :
    int  x, y ;
public :
    Point(int a=0, int b=0) {
        x = a ; y = b ;
    }
    virtual void print() {
        cout<<"Point: "<<x<<" "<<y<<endl;
    }
} ;

class Pointcol: public Point {
    ...
} ;
```

```
void main() {
    Point p(3,5), *pp = &p ;
    Pointcol pc(8, 6, 2) ,
    Pointcol *ppc =&pc ;
    //call Point::print()
    pp->print() ;
    //call Pointcol::print()
    ppc->print() ;
    pp = ppc ;
    //call Pointcol::print()
    pp->print() ;
    //call Pointcol::print()
    ppc->print() ;
}
```

Declaring `Point::print()` as a virtual function allows dynamic binding: pointer `pp` will call `Point::print()` if it is pointing to an instance of `Point`, or `Pointcol::Print()` if it is pointing to an instance of `Pointcol`.

Few remarks about VF

- If f is a VF of class A, it will remain virtual in all (direct or indirect) sub-classes of A.
- The override of a VF is not mandatory, to make the overriding mandatory we should declare the VF as **pure** :

```
class A {  
    ...  
    public:  
    virtual void f() = 0;  
    ...  
} ;
```

Attention : a pure VF has no definition in the class where it is declared. It is mandatory to define it in all the subclasses of this class.

- Any class that contains a pure VF is an abstract class. No object can be instantiated from this class.

```
A myobject ;    //Forbidden declaration, A is an abstract class.
```

- A VF of a class A, can be overridden in its subclasses. However, if the function is declared with arguments that are not exactly the same as those in the parent class, then for the system it is a different function.

End

Next Lecture: Template Functions and Classes

Thank you