Data Structures and Object Oriented Programming using C++

Ahsan Ijaz

Protected Member

Without Inheritance:

- Same as Private members
- Can only be accessed by member functions of class

With Public Inheritance:

- Private member from base class is inaccessible in derived class.
- Protected member from base class is accessible in derived class.

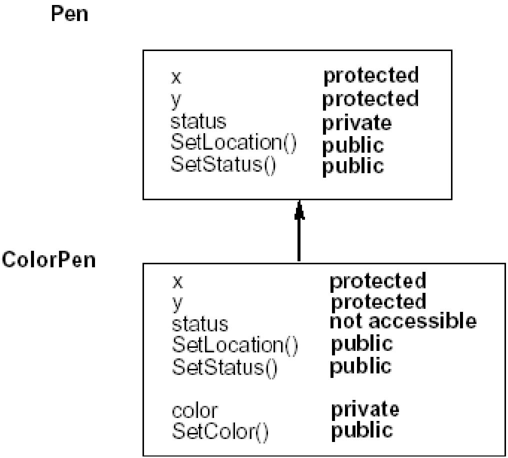
Member access Error

```
class Pen {
public:
void SetLocation(int, int);
void SetStatus(int);
private:
int x, y, status;};
class ColorPen : public Pen {
public:
void SetColor(int);
void setX(int xx){ x = xx; }//Error!
           private:
int color;
};
```

Member Can be accessed now

```cpp
class Pen {
public:
void SetLocation(int, int);
void SetStatus(int);
protected:
int x, y;
private:
int status;};
class ColorPen : public Pen {
public:
void SetColor(int);
void setX(int xx){ x = xx;}//OK!
private:
int color;
};
```

Member Access



Pen

x	protected
y	protected
status	private
SetLocation()	public
SetStatus()	public

ColorPen

x	protected
y	protected
status	not accessible
SetLocation()	public
SetStatus()	public
color	private
SetColor()	public

Member Access

```
void ColorPen::SetColor()
{x = 1; // error or ok?
 y = 1; // error or ok?
 status = 0; // error or ok?
 color = 2; // error or ok?
};
void main()
{ColorPen p;
 p.x = 1; // error or ok?
 p.y = 1; // error or ok?
 p.status = 1; // error or ok?
 p.color = 1; // error or ok?
}
```

Member Access

```
void ColorPen::SetColor()
{x = 1; // ok: x is protected in Pen
 y = 1; // error or ok?
 status = 0; // error or ok?
 color = 2; // error or ok?
};
void main()
{ColorPen p;
 p.x = 1; // error or ok?
 p.y = 1; // error or ok?
 p.status = 1; // error or ok?
 p.color = 1; // error or ok?
}
```

Member Access

```
void ColorPen::SetColor()
{x = 1; // ok: x is protected in Pen
 y = 1; // ok: y is protected in Pen
 status = 0; // error or ok?
 color = 2; // error or ok?
};
void main()
{ColorPen p;
 p.x = 1; // error or ok?
 p.y = 1; // error or ok?
 p.status = 1; // error or ok?
 p.color = 1; // error or ok?
}
```

Member Access

```
void ColorPen::SetColor()
{x = 1; // ok: x is protected in Pen
 y = 1; // ok: y is protected in Pen
 status = 0; // error: status is invisible
 color = 2; // ok: color is private in Colorpen
};
void main()
{ColorPen p;
 p.x = 1; // error or ok?
 p.y = 1; // error or ok?
 p.status = 1; // error or ok?
 p.color = 1; // error or ok?
}
```

Protected Member Summary

- Private members from base class are inaccessible in derived class.
- Protected members from base class are accessible to member functions of derived class.
- Both private and protected members are inaccessible in main() (outside class, both act as private members) and are inaccessible to users.

Object Composition

- An object of the derived class consists of sub-object of its base classes and the derived class portion.
- To construct an object of the derived class, objects of the base classes are constructed first.

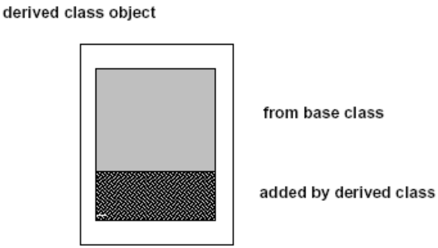


Figure: Composition

Constructors in Multiple inheritance

```
class Base {
// ...
};

class Derived : public Base{
// ...
};

class DoubleDerived : public Derived {
// ...
};
DoubleDerived dd;
```

Composition of Multiple Inheritance

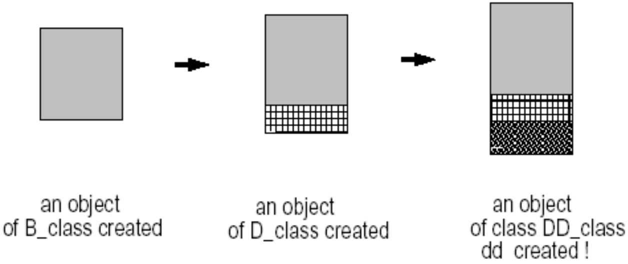


Figure: Composition in case of Multiple inheritance

Constructor Call order

- In an inheritance hierarchy, constructors execute in a **base class to derived class order.**

Derived objects are constructed by the following steps:

- Allocate space for the entire object(base class members and derived class members)
- Invoke the base class constructor to initialize the base class part of the object.
- Invoke the derived class constructor to initialize the derived class part of the object.

Constructor Call order

- In an inheritance hierarchy, constructors execute in a **base class to derived class order.**

Derived objects are constructed by the following steps:

- Allocate space for the entire object(base class members and derived class members)
- Invoke the base class constructor to initialize the base class part of the object.
- Invoke the derived class constructor to initialize the derived class part of the object.

Constructor Call order

- In an inheritance hierarchy, constructors execute in a **base class to derived class order.**

Derived objects are constructed by the following steps:

- Allocate space for the entire object(base class members and derived class members)
- Invoke the base class constructor to initialize the base class part of the object.
- Invoke the derived class constructor to initialize the derived class part of the object.

Constructor Call order

- In an inheritance hierarchy, constructors execute in a **base class to derived class order.**

Derived objects are constructed by the following steps:

- Allocate space for the entire object(base class members and derived class members)
- Invoke the base class constructor to initialize the base class part of the object.
- Invoke the derived class constructor to initialize the derived class part of the object.

Constructor Call order

- In an inheritance hierarchy, constructors execute in a **base class to derived class order.**

Derived objects are constructed by the following steps:

- Allocate space for the entire object(base class members and derived class members)
- Invoke the base class constructor to initialize the base class part of the object.
- Invoke the derived class constructor to initialize the derived class part of the object.

Constructor Call Order

```
class B_class
{public:
B_class(){ cout << "creating␣Bclass\n";};
};
class D_class : public B_class{
public:
D_class()
{ cout << "constructing␣Dclass\n"; }
};
int main(){
D_class d; // output?
}
```

- creating Bclass
- constructing Dclass

Constructor Call Order

```
class B_class
{public:
B_class(){ cout << "creating Bclass\n";};
};
class D_class : public B_class{
public:
D_class()
{ cout << "constructing Dclass\n"; }
};
int main(){
D_class d; // output?
}
```

- creating Bclass
- constructing Dclass

Constructor Calling Example

Consider the following code. What is wrong?

```
class B_class {
public:
B_class(int a, int b) : privateA(a), privateB(b) {}
private:
int privateA, privateB;
};
class D_class : public B_class{
public:
D_class( int zz ) : D_privateA(zz) {}
Private:
int D_privateA;
};
int main(){
D_class d( 10 ); // ....
```

Constructor Calling Example

Solution 1: Provide a default constructor to the base class.

```
class B_class {
public:
B_class(int a, int b) : privateA(a), privateB(b) {}
B_class() {}
private:
int privateA, privateB;
};
class D_class : public B_class{
public:
D_class( int zz ) : D_privateA(zz) {}
private:
int D_privateA;
};
int main(){
D_class d( 10 );
}
```

Constructor Calling Example

Solution 2: Explicitly invokes a B class constructor in D class constructor initializer.

```
class B_class {
public:
        B_class(int a, int b) : privateA(a), privateB(b) {}
private:
    int privateA, privateB;
};
class D_class : public B_class{
public:
        D_class( int zz, int a, int b ) : B_class(a,b), D_privateA
private:
    int D_privateA;
};
int main(){
D_class d(10,5,4);
}
```

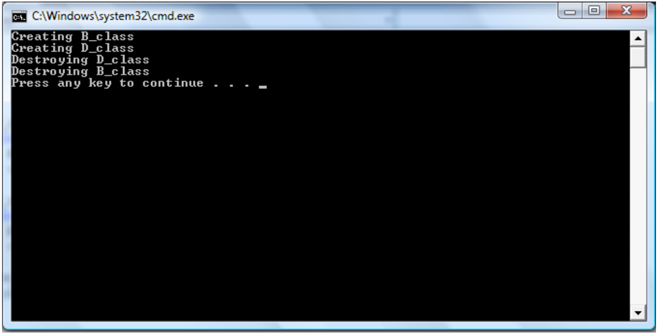
Destructor

- When a derived class object is destroyed, the derived class portion is destroyed first.
- In an inheritance hierarchy, destructors execute in a derived class to base class order, which is the reverse order of constructors.

Destructor Example

```
class B_class {
public:
B_class(){cout<<"Creating␣B_class\n";}
~B_class(){cout<<"Destroying␣B_class\n";}
};
class D_class : public B_class{
public:
D_class(){cout<<"Creating␣D_class␣\n"; }
~D_class(){cout<<"Destroying␣D_class\n";}
};
int main()
{D_class d; // Output?
}
```

Destructor Example Output



```
Creating B_class
Creating D_class
Destroying D_class
Destroying B_class
Press any key to continue . . . _
```

Figure: Destructor Output

Class Scope

- Base class members and its derived class members belong to different class scopes.
- The scope of the derived class can be viewed as nested within the scope of its base class.

Class Scope

```
class B_class{
protected:
string status;
int protectedofB;
private:
int privateOfB;
};
class D_class : public B_class{
public:
void f();
private:
int status;
};
```

Class Scope

```
void D_class::somefunction(){
status = 1;
status = "hello!";
B_class::status = "hello!";
protectedofB = 1;
privateOfB = 1;
}
```

Class Scope

```
void D_class::somefunction(){
status = 1; //Ok
status = "hello!";
B_class::status = "hello!";
protectedofB = 1;
privateOfB = 1;
}
```

Class Scope

```
void D_class::somefunction(){
status = 1; //Ok
status = ”hello!”;
//Error, status is resolved to D_class::x
B_class::status = ”hello!”;
protectedofB = 1;
privateOfB = 1;
}
```

Class Scope

```
void D_class::somefunction(){
status = 1; //Ok
status = "hello!";
//Error, status is resolved to D_class::x
B_class::status = "hello!";   //Okay
protectedofB = 1;
privateOfB = 1;
}
```

Class Scope

```
void D_class::somefunction(){
status = 1; //Ok
status = "hello!";
//Error, status is resolved to D_class::x
B_class::status = "hello!";  //Okay
protectedofB = 1;  //Okay
privateOfB = 1;
}
```

Class Scope

```
void D_class::somefunction(){
status = 1; //Ok
status = "hello!";
//Error, status is resolved to D_class::x
B_class::status = "hello!";  //Okay
protectedofB = 1;  //Okay
privateOfB = 1;
/*Error, accessing private of Class_B in
Derived Class*/
}
```

Function Overloading

If the derived class adds a member with the same name of a member in the base class, the local member hides the inherited member.

Overloaded Function...Not

```
class B_class{
public:
void overfunction(){ /*...*/ }
};
class D_class : public B_class{
public:
void overfunction(int){ /*...*/ }
};
int main(){
D_class x;
x.overfunction(); //Error
}
```

How to overloaded Function

Qualify the base class member with the class scope operator so it become visible in the derived class scope, and therefore the same named functions can be overloaded.

How to overloaded function

```
class Base{
public:
void f(double);
};
class Derived : public Base{
public:
using Base::f;
void f(int);
};
```

Type of Inheritance

Class derivation has three types:

- Public
- Private
- Protected

Type of Inheritance

```
class Base{
//...
};
// protected inheritance
class Derived1 : protected Base{
//...
};
// private inheritance
class Derived2 : Base{ // Caution:
//...       // inheritance is PRIVATE
};         // by default !!!
```

Type and Implementation inheritance

- Public derivation is called type inheritance.
- Private derivation is called implementation inheritance.

Type of Inheritance

```cpp
class A
{
public:
    int x;
protected:
    int y;
private:
    int z;
};

class B : public A
{
    // x is public
    // y is protected
    // z is not accessible from B
};

class C : protected A
{
    // x is protected
    // y is protected
    // z is not accessible from C
};

class D : private A
{
    // x is private
    // y is private
    // z is not accessible from D
};
```