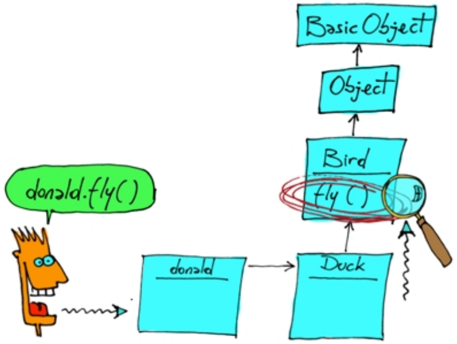
Inheritance

Meng-Hsun Tsai
CSIE, NCKU



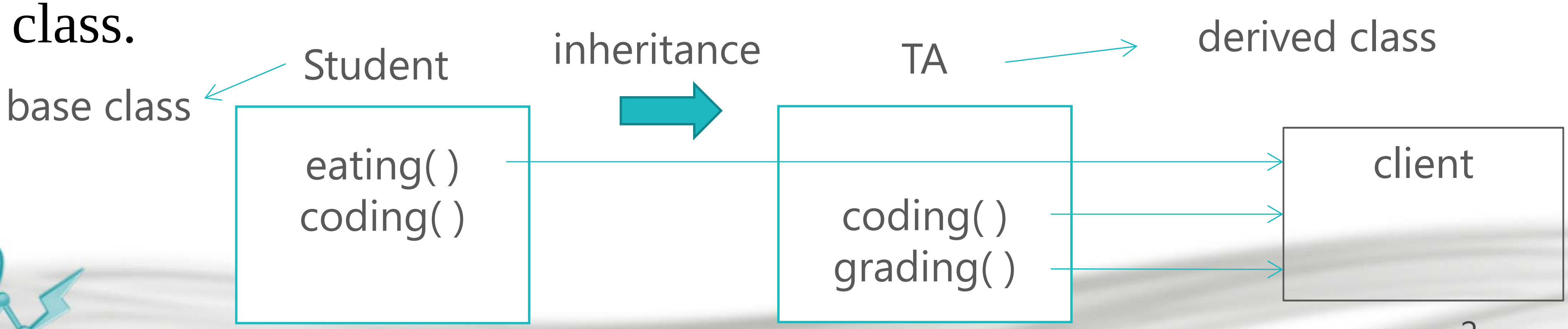
Introduction

- Inheritance is a form of software reuse in which you create a class that absorbs an existing class's data and behaviors and enhances them with new capabilities.

- This existing class is called the base class, and the new class is referred to as the derived class.

- A derived class contains behaviors inherited from its base class and can contain additional behaviors.

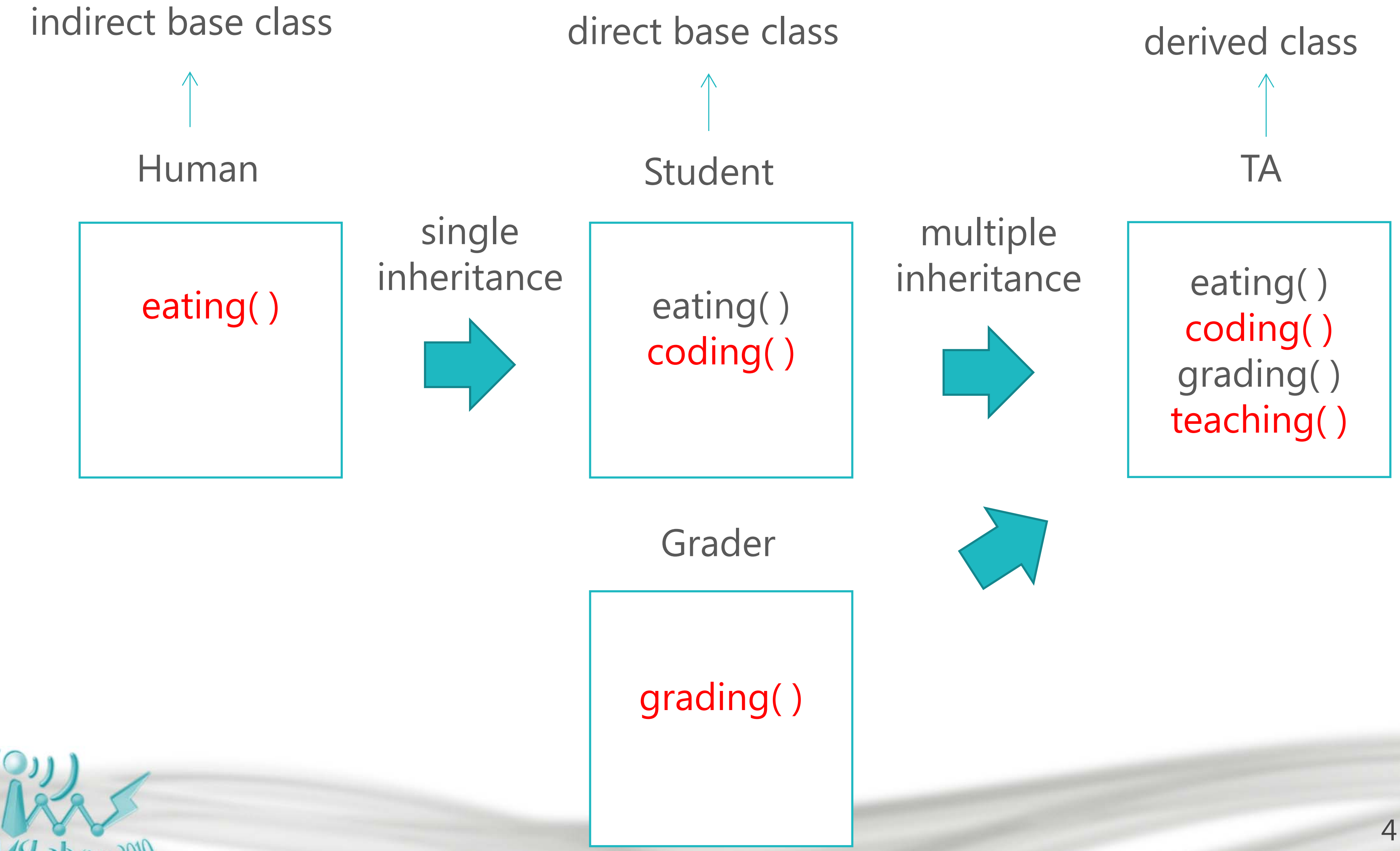
- A derived class can also customize behaviors inherited from the base class.



Direct vs. Indirect Base Class and Single vs. Multiple Inheritance

- A direct base class is the base class from which a derived class explicitly inherits.
- An indirect base class is inherited from two or more levels up in the class hierarchy.
- In the case of single inheritance, a class is derived from one base class.
- C++ also supports multiple inheritance, in which a derived class inherits from multiple (possibly unrelated) base classes.

Direct vs. Indirect Base Class and Single vs. Multiple Inheritance (cont.)



indirect base class

direct base class

derived class

Human

eating( )

single inheritance

Student

eating( )
coding( )

multiple inheritance

TA

eating( )
coding( )
grading( )
teaching( )

Grader

grading( )

public, *private* and *protected* Inheritance

- C++ offers `public`, `private` and `protected` inheritance.
- In this lecture, we concentrate on `public` inheritance and briefly explain the other two.
- The third form, `protected` inheritance, is rarely used.
- Every object of a derived class is also an object of that derived class's base class.
- However, base-class objects are not objects of their derived classes.

is-a vs. *has-a* Relationship

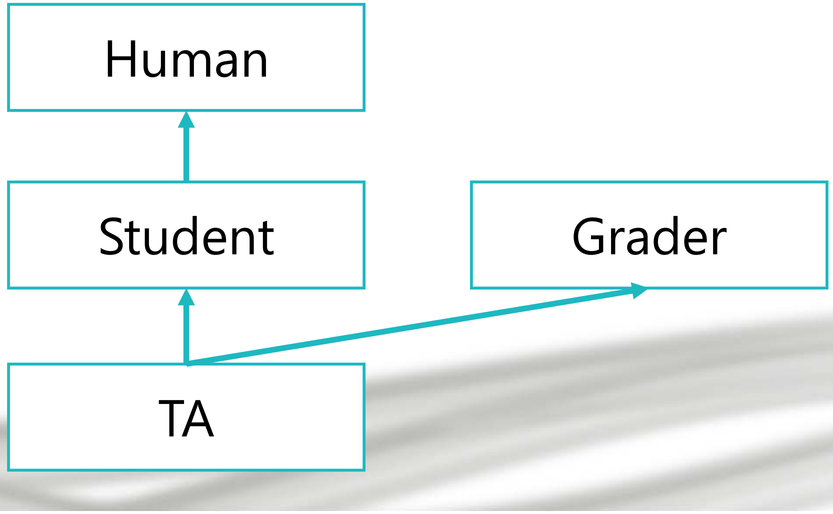
- We distinguish between the is-a relationship and the *has-a* relationship.
- The *is-a* relationship represents inheritance.
- In an *is-a* relationship, an object of a derived class also can be treated as an object of its base class.
- By contrast, the *has-a* relationship represents composition.

Base Classes and Derived Classes

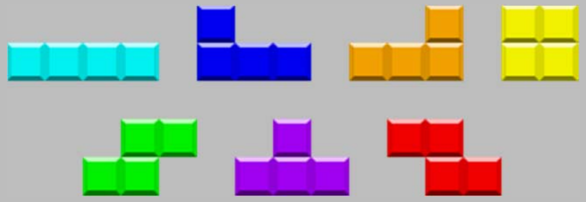
- Often, an object of one class *is an* object of another class, as well.
  - For example, in geometry, a rectangle *is a* quadrilateral (as are squares, parallelograms and trapezoids).
  - Thus, in C++, class `Rectangle` can be said to inherit from class `Quadrilateral`.
  - A rectangle *is a* specific type of quadrilateral, but it’s incorrect to claim that a quadrilateral is a rectangle—the quadrilateral could be a parallelogram or some other shape.

Base Classes and Derived Classes (cont.)

- A base class exists in a hierarchical relationship with its derived classes.
- A class becomes either a base class—supplying members to other classes, a derived class—inheriting its members from other classes, or both.
- In the UML diagram, each arrow in the hierarchy represents an *is-a relationship*. For example, A TA is a student. A student is a human.



Tetrominos in Tetris Game



I : four blocks in a straight line

J : a row of three blocks with one added below the right side.

L : a reflection of J but cannot be rotated into J in two dimensions

S : two stacked horizontal dominoes with the top one offset to the right

Z : a reflection of S but cannot be rotated into S in two dimensions

O : four blocks in a 2×2 square.

T : a row of three blocks with one added below the center

Source: http://en.wikipedia.org/wiki/Tetris

http://en.wikipedia.org/wiki/Tetromino

IBlock.h

```
#ifndef I_BLOCK_H
#define I_BLOCK_H
#include <iostream>
using namespace std;
char I_arr [2][4][4] = {{{'0','0','1','0'},
                          {'0','0','1','0'},
                          {'0','0','1','0'},
                          {'0','0','1','0'}},
                         {{'0','0','0','0'},
                          {'0','0','0','0'},
                          {'1','1','1','1'},
                          {'0','0','0','0'}} };
class I_Block{
    public:
        I_Block():x(0),y(0),rotate_index(0) {}
        I_Block& rotate(){
            rotate_index=(rotate_index>0)?
                          0:rotate_index+1;
            return *this;
        }
        I_Block& left() {x=(x>0)?(x-1):10;
                              return *this;}
        I_Block& right() {x=(x>10)?0:x+1;
                              return *this;}
        void paint() {
            for(int i=0;i<4;++i)
            {
                for(int j=0;j<x;++j) cout << ' ';
                for(int j=0;j<4;++j)
                    cout << I_arr[rotate_index][i][j];
                cout << endl;
            }
            cout << endl;
        }
    public:
        int x, y;
        int rotate_index;
};
#endif
```

tetris.cpp

```cpp
#include <iostream>
#include "IBlock.h"
using namespace std;
int main()
{
  I_Block i;
  i.paint();
  i.rotate().paint();
  i.right().paint();
  i.right().rotate().paint();
  return 0;
}
```

```
0010
0010
0010
0010
0000
0000
1111
0000
 0000
 0000
 1111
 0000
  0010
  0010
  0010
  0010
```

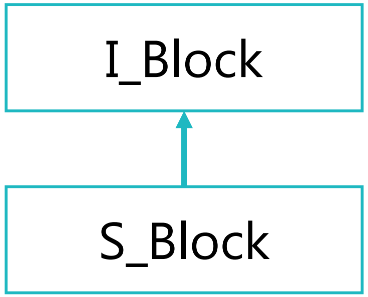
Adding SBlock.h Without Inheritance

```
#ifndef S_BLOCK_H
#define S_BLOCK_H
#include <iostream>
using namespace std;
char S_arr [2][4][4] = {{{'0','0','0','0'},
                         {'0','0','0','0'},
                         {'0','0','1','1'},
                         {'0','1','1','0'}},
                        {{'0','0','0','0'},
                         {'0','1','0','0'},
                         {'0','1','1','0'},
                         {'0','0','1','0'}} };
class S_Block{
  public:
     S_Block():x(0),y(0),rotate_index(0) {};
     S_Block& rotate(){
        rotate_index=(rotate_index>0)?
                           0:rotate_index+1;
        return *this;
     }
     S_Block& left() {x=(x>0)?(x-1):10;
                        return *this;}
     S_Block& right() {x=(x>10)?0:x+1;
                        return *this;}
     void paint() {
        for(int i=0;i<4;++i)
        {
           for(int j=0;j<x;++j) cout << ' ';
           for(int j=0;j<4;++j)
              cout << S_arr[rotate_index][i][j];
           cout << endl;
        }
        cout << endl;
     }
  private:
     int x, y;
     int rotate_index;
};
#endif
```

Most codes are the same as IBlock.h!

Adding SBlock_inh.h With Inheritance

```
#ifndef S_BLOCK_INH_H
#define S_BLOCK_INH_H
#include <iostream>
#include "IBlock.h"
using namespace std;
char S_arr [2][4][4] = {{{'0','0','0','0'},
                         {'0','0','0','0'},
                         {'0','0','1','1'},
                         {'0','1','1','0'}},
                        {{'0','0','0','0'},
                         {'0','1','0','0'},
                         {'0','1','1','0'},
                         {'0','0','1','0'}} };
class S_Block: public I_Block{
    public:
        void paint() {
            for(int i=0;i<4;++i)
            {
                for(int j=0;j<x;++j) cout << ' ';
                for(int j=0;j<4;++j)
                    cout << S_arr[rotate_index][i][j];
                cout << endl;
            }
            cout << endl;
        }
};
#endif
```



Only paint() needs to be re-written.

tetris.cpp

```
#include <iostream>
#include "SBlock_inh.h"
using namespace std;
int main()
{
   S_Block s;
   s.paint();
   s.rotate().paint();
   s.right().paint();
   s.right().rotate().paint();
   return 0;
}
```

Something Wrong!

```
0000
0000
0011
0110
0000
0000
1111
0000
 0000
 0000
 1111
 0000
  0010
  0010
  0010
  0010
```

Member function calls can not be cascaded since rotate() returns I_Block&, causing I_Block's paint() to be executed.

tetris2.cpp

```cpp
#include <iostream>
#include "SBlock_inh.h"
using namespace std;
int main()
{
   S_Block s;
   s.paint();
   s.rotate();
   s.paint();
   s.right();
   s.paint();
   s.right();
   s.rotate();
   s.paint();
   return 0;
}
```

```
0000
0000
0011
0110
0000
0100
0110
0010
0000
0100
0110
0010
0000
0000
0011
0110
```

PASSED
PASSED
PASSED

Passing Arguments to Constructors

IBlock2.h

```
...
class I_Block{
    public:
        I_Block(int xx=0,int yy=0,int ri=0):
            x(xx),y(yy),rotate_index(ri) {}    ;
...
```

SBlock_inh2.h

```
...
class S_Block: public I_Block{
    public:
        S_Block(int sx=0, int sy=0, int si=0):
            I_Block(sx,sy,si) {}
...
```

tetris3.cpp

```
...
int main()
{
    S_Block s(2,0,1);
...
```

```
0000
0100
0110
0010
0000
0000
0011
0110
 0000
 0000
 0011
 0110
  0000
  0100
  0110
  0010
```

Passing Arguments to Constructors (cont.)

- The colon (`:`) in line 14 of *SBlock_inh2.h* indicates inheritance.
- Keyword `public` indicates the type of inheritance.
- As a derived class (formed with `public` inheritance), `S_Block` inherits all the members of class `I_Block`, except for the constructor—each class provides its own constructors that are specific to the class.
- Destructors, too, are not inherited
- The constructor introduces base-class initializer syntax, which uses a member initializer to pass arguments to the base-class constructor.

```
14  class S_Block: public I_Block{
15    public:
16        S_Block(int sx=0, int sy=0, int si=0):
              I_Block(sx,sy,si) {}
```

Error Accessing *private* Members of Base Class

IBlock3.h

```
…
32    private:
33       int rotate_index;
34       int x, y;
…
```

SBlock_inh2.h

```
17      void paint() {
18         for(int i=0;i<4;++i)
19         {
20            for(int j=0;j<x;++j) cout << ' ';
21            for(int j=0;j<4;++j)
22               cout << S_arr[rotate_index][i][j];
23            cout << endl;
24         }
25         cout << endl;
26      }
```

```
In file included from tetris.cpp:2:
IBlock3.h: In member function 'void S_Block::paint()':
IBlock3.h:34: error: 'int I_Block::x' is private
SBlock_inh2.h:20: error: within this context
IBlock3.h:33: error: 'int I_Block::rotate_index' is private
SBlock_inh2.h:22: error: within this context
*** [tetris.o] Error code 1
```

Error Accessing *private* Members of Base Class

IBlock4.h

```
...
protected:
    int rotate_index;
    int x, y;
...
```

```
0000
0100
0110
0010
0000
0000
0011
0110
 0000
 0000
 0011
 0110
  0000
  0100
  0110
  0010
```

PASSED
★★★
PASSED
★★★
PASSED

Accessing *private* Members of Base Class

- The compiler generates errors because base class `I_Block`'s data members are `private`—derived class `S_Block`'s member functions are not allowed to access base class `I_Block`'s `private` data.
- The errors in derived class could have been prevented by using the *get* member functions inherited from base class.

`private` Members

- A derived class can access the non-`private` members of its base class.
- Base-class members that should not be accessible to the member functions of derived classes should be declared `private` in the base class.
- A derived class can change the values of `private` base-class members, but only through non-`private` member functions provided in the base class and inherited into the derived class.

protected Members

- A base class's public members are accessible anywhere.
- A base class's private members are accessible only within its body and to the friends of that base class.
- Using protected access offers an intermediate level of protection between public and private access.
- Derived-class member functions can refer to public and protected members of the base class simply by using the member names.

`protected` Members (cont.)

- When a derived-class member function redefines a base-class member function, the base-class member can be accessed from the derived class by preceding the base-class member name with the base-class name and the binary scope resolution operator (`::`).
- A base class's `protected` members can be accessed by members and `friend`s of the base class and by members and `friend`s of any classes derived from that base class.
- Objects of a derived class also can access `protected` members in any of that derived class's indirect base classes

Fake main Function Through Inheritance

```
class A {
public:
   A(int aa, int bb): x(aa), y(bb) {  }
private:
   int x,y;
};
class B : public A {
public:
   B(int a, int b): A(a,b) { }
   void fake_main() { cout << x << '\t' << y << endl; }
};
int main()
{
   B b(1,2);
   b.fake_main();
}
```

```
fake_main.cpp:5: error: 'int A::x' is private
fake_main.cpp:10: error: within this context
fake_main.cpp:5: error: 'int A::y' is private
fake_main.cpp:10: error: within this context
```

Constructors and Destructors in Derived Classes

- Instantiating a derived class object begins a chain of constructor calls in which the derived class constructor, before performing its own tasks, invokes its direct base class's constructor either explicitly (via a base class member initializer) or implicitly (calling the base class's default constructor).

- If the base class is derived from another class, the base class constructor is required to invoke the constructor of the next class up in the hierarchy, and so on.

- The last constructor called in this chain is the constructor of the class at the base of the hierarchy, whose body actually finishes executing first.

- The original derived-class constructor's body finishes executing last.

Constructors and Destructors in Derived Classes (cont.)

- When a derived class object is destroyed, the program calls that object's destructor.
- This begins a chain (or cascade) of destructor calls in which the derived-class destructor and the destructors of the direct and indirect base classes and the classes' members execute in reverse of the order in which the constructors executed.
- When a derived class object's destructor is called, the destructor performs its task, then invokes the destructor of the next base class up the hierarchy.
- This process repeats until the destructor of the final base class at the top of the hierarchy is called.
- Then the object is removed from memory.

Constructors and Destructors in Derived Classes (cont.)

- Base-class constructors, destructors and overloaded assignment operators are not inherited by derived classes.
- Derived-class constructors, destructors and overloaded assignment operators **can call** base-class constructors, destructors and overloaded assignment operators.

Constructors and Destructors in Derived Classes with Composition

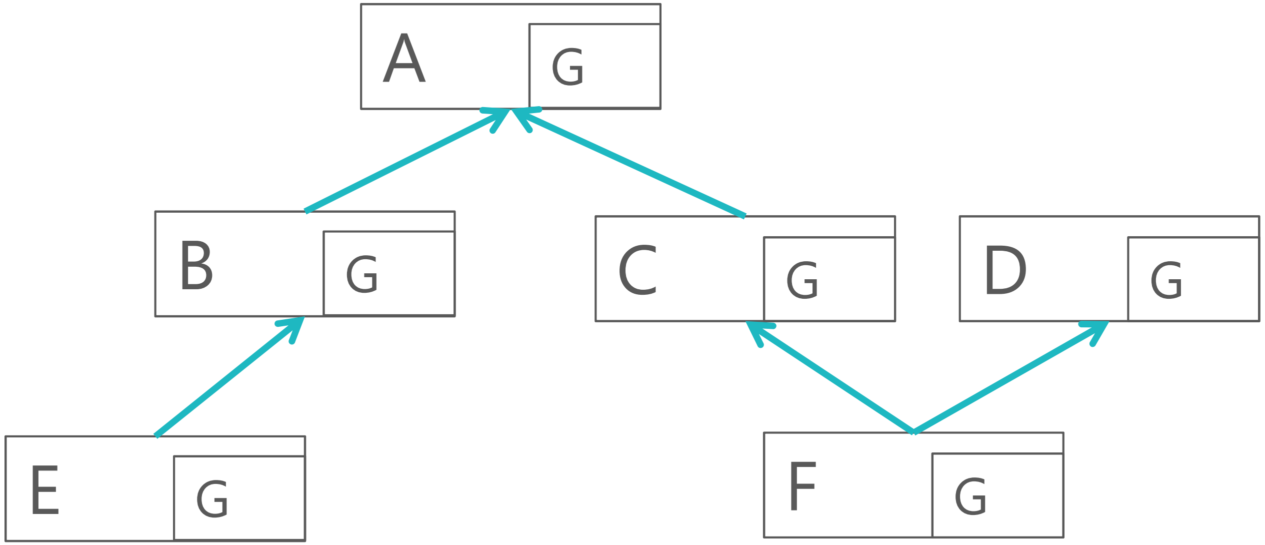
- Suppose that we create an object of a derived class where both the base class and the derived class contain (via composition) objects of other classes. When an object of that derived class is created, first the constructors for the base class's member objects execute, then the base class constructor executes, then the constructors for the derived class's member objects execute, then the derived class's constructor executes.
- Destructors for derived class objects are called in the reverse of the order in which their corresponding constructors are called.

Sequence of Constructors and Destructors

```
#include <iostream>
using namespace std;
class G {
public: G() { cout << "G ctor" << endl;}
        ~G() { cout << "G dtor" << endl;}
};
class A {
public:  A() { cout << "A ctor" << endl;}
         ~A() { cout << "A dtor" << endl;}
         G objG;
};
class B: public A {
public:  B() { cout << "B ctor" << endl;}
        ~B() { cout << "B dtor" << endl;}
        G objG;
};
class C: public A {
public:  C() { cout << "C ctor" << endl;}
         ~C() { cout << "C dtor" << endl;}
         G objG;
};
```

```
class D {
public:  D() { cout << "D ctor" << endl;}
         ~D() { cout << "D dtor" << endl;}
         G objG;
};
class E: public B {
public:  E() { cout << "E ctor" << endl;}
          ~E() { cout << "E dtor" << endl;}
          G objG;
};
class F: public C, public D {
public:  F() { cout << "F ctor" << endl;}
          ~F() { cout << "F dtor" << endl;}
          G objG;
};
int main()
{
   E objE;
   cout << endl;
   F objF;
   cout << endl;
   return 0;
}
```

Sequence of Constructors and Destructors (cont.)



Output:

```
G ctor      F dtor
A ctor      G dtor
G ctor      D dtor
B ctor      G dtor
G ctor      C dtor
E ctor      G dtor
            A dtor
G ctor      G dtor
A ctor      E dtor
G ctor      G dtor
C ctor      B dtor
G ctor      G dtor
D ctor      A dtor
G ctor      G dtor
F ctor
```

public, protected and private Inheritance

- When deriving a class from a base class, the base class may be inherited through `public`, `protected` or `private` inheritance.
- Use of `protected` and `private` inheritance is rare, and each should be used only with great care; we normally use `public` inheritance.
- A base class's `private` members are never accessible directly from a derived class, but can be accessed through calls to the `public` and `protected` members of the base class.

`public`, `protected` and `private` Inheritance (cont.)

Access Specifier in Base Class	**public inheritance**	**protected inheritance**	**private inheritance**
public	**public** in derived class	**protected** in derived class	**private** in derived class
protected	**protected** in derived class	**protected** in derived class	**private** in derived class
private	*Hidden* in derived class	*Hidden* in derived class	*Hidden* in derived class