Chapter 11

Inheritance

Chapter 11 Topics(part 3)

- Multiple Inheritance（多重继承）
  - Inheritance as Combination
  - Multiple Inheritance Declaration
  - Inheritance from Common Ancestors
- Inheritance vs. Composition（组合）
  - Two Approaches to Software Reuse
  - Advantages and Disadvantages

Inheritance as Categorization(分类)

In one sense, the process of inheritance is a form of categorization.

The author of the textbook is

- North American
- Male
- Professor
- Parent

Inheritance as Combination

In real world, objects are combinations of features from different classification(分类) schemes(方案).

- Author is North American, and
- Author is Male, and
- Author is a Professor, and
- Author is a Parent.

An Example

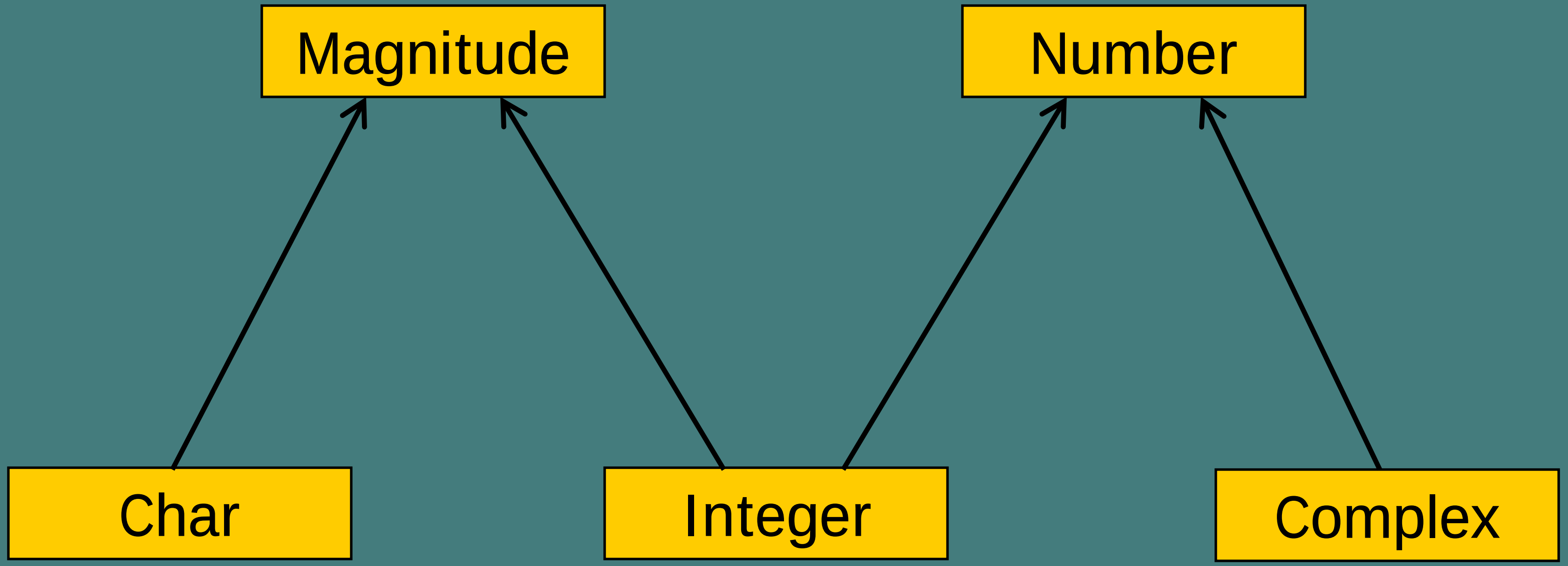
Two abstract classifications

- *Magnitude(数量)*
  - things that can be compared to each other.
- *Number*
  - things that can perform arithmetic operations.

Three specific classes

- **Integer** - comparable and arithmetical
- **Char** - comparable but not arithmetical
- **Complex** - arithmetical but not comparable

Inheritance as a Form of Combination



Multiple Inheritance Declaration

SYNTAX

```
class ChildName:[public|private] ParentName1,…, [public|private] ParentNameN{
…
};
```

Problem with Multiple Inheritance - Name Ambiguity

```
class X{
 public:
    int f( );
};
```

```
class Y{
 public:
    int f( );
    int g( );
};
```

```
class Z:public X, public Y{
 public:
    int g( );
    int h( );
};
```

```
Z obj;
obj . f( );  //error
obj . g( );  //
```

One Solution:Full Qualified Name

One solution is to simply always use fully qualified names.

```
Z obj;
obj . X::f( );  // selects f( ) in X
obj . Y::f( );  // selects f( ) in Y
obj . Y:: g( );  // selects g( ) in Y
obj . g( );  // selects g( ) in Z
```

Constructor Declaration for Multiple Inheritance

SYNTAX

```
ChildName(ParameterList) :ParentName1(ParameterList1), … ,ParentNameN(ParameterListN){
    …
};
```

Example

```
class X{
 public:
    X(int sa)
    {a=sa;}
   int getX( )
    {return a;}
 private:
    int a;
};
```

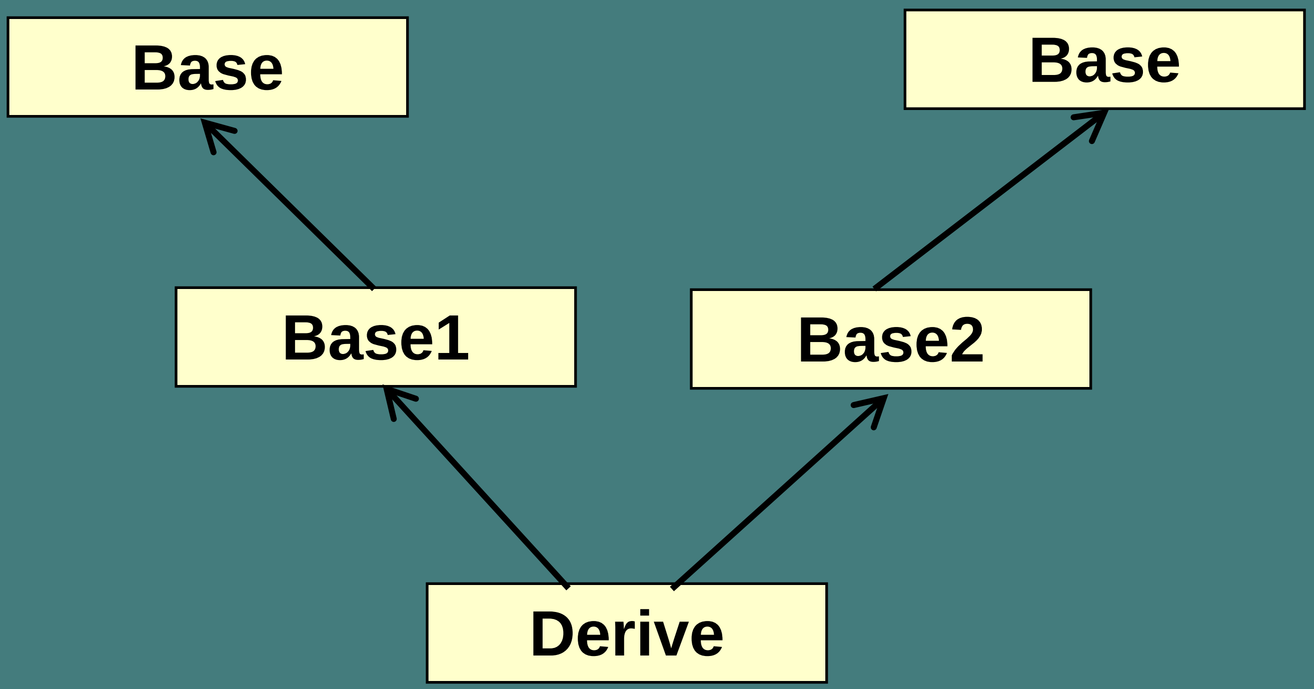
```
class Y{
 public:
    Y(int sb)
    {b=sb;}
   int getY( )
    {return b;}
 private:
    int b;
};
```

```
class Z: public X, private Y{
 public:
   Z(int sa,int sb,int sc):X(sa),Y(sb)
   {c=sc;}
   int getZ( )
    {return c;}
   int getY( )
    {return Y::getY( );}
 private:
    int c;
};
```

```
Z obj(2,4,6);
cout<<"a="<<obj.getX( )<<endl;
cout<<"b="<<obj.getY( )<<endl;
cout<<"c="<<obj.getZ( )<<endl;
```

Inheritance from Common Ancestors

Imagine that the **common ancestor (公共祖先)** declares a data member. Should the child class have one copy of this data field, or two?



Example

```
class Base{
public:
    Base(int sa)
    { a=sa; cout<<“Constructing Base”<<endl;}
private:
    int a;
};
```

```
class Base1: public Base{
public:
    Base1(int sa,int sb):Base(sa)
    {b=sb;
    cout<<“Constructing Base1”<<endl;}
private:
    int b;
};
```

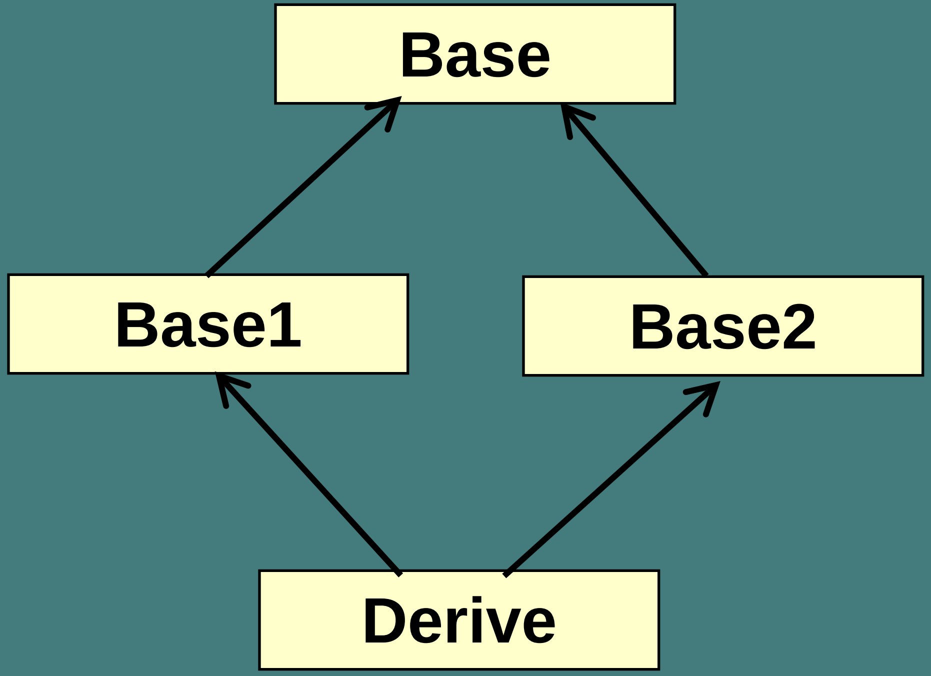
```
class Base2: public Base{
public:
    Base2(int sa,int sc):Base(sa)
    {c=sc;
    cout<<“Constructing Base2”<<endl;}
private:
    int c;
};
```

```
class Derive: public Base1, public Base2{
public:
    Derive(int sa,int sb, int sc,int sd):Base1(sa,sb), Base2(sa,sc)
    {d=sd;cout<<“Constructing Derive”<<endl;}
private:
    int d;
};
```

```
#include<iostream>
using namespace std;
int main( )
{Derive obj(2, 4, 6, 8);  return 0;}
```

One Solution: virtual parent class

C++ gets around this by introducing the idea of a **virtual parent class(虚基类)**. If your parent is virtual, there is one copy.



Example

```
class Base{
public:
    Base(int sa)
    { a=sa; cout<<“Constructing Base”<<endl;}
private:
    int a;
};
```

```
class Base1:virtual public Base{
public:
    Base1(int sa,int sb):Base(sa)
    {b=sb;
    cout<<“Constructing Base1”<<endl;}
private:
    int b;
};
```

```
class Base2:virtual public Base{
public:
    Base2(int sa,int sc):Base(sa)
    {c=sc;
    cout<<“Constructing Base2”<<endl;}
private:
    int c;
};
```

```
class Derive: public Base1, public Base2{
public:
    Derive(int sa,int sb, int sc,int sd):Base(sa), Base1(sa,sb), Base2(sa,sc)
    {d=sd;cout<<“Constructing Derive”<<endl;}
private:
    int d;
};
```

```
#include<iostream>
using namespace std;
int main(  )
{Derive obj(2, 4, 6, 8);  return 0;}
```

Chapter 11 Topics(part 3)

- Multiple Inheritance
  - Inheritance as Combination
  - Multiple Inheritance Declaration
  - Inheritance from Common Ancestors
- Inheritance vs. Composition
  - Two Approaches to Software Reuse
  - Advantages and Disadvantages

Two Approaches to Software Reuse

- **Inheritance -- the *is-a* relationship.**
- **Composition -- the *has-a* relationship.**

Example - Building `Set` from `List`

Suppose we have already a `List` data type with the following behavior:

```
class List {
public:
    //constructor
    List( );
    //methods
    void add (int);
    int firstElement ( );
    int size( );
    bool includes (int);
    void remove (int);
    …
};
```

Want to build the `Set` data type.

Using Composition

```
class Set {
public:
   //constructor
   Set( );
   //operations
   void add(int);
   int size( );
   bool includes(int);
private:
   List theData;
};
```

```
//initialize list
Set::Set( ):theData( )
{  }
```

```
int Set::size( )
{return theData.size( );}
```

```
bool Set::includes(int newValue)
{return theData.includes(newValue);}
```

```
void Set::add(int newValue)
{  //if not already in set
 if(!includes(newValue))
     //then add
     theData.add(newValue);
}
```

Using Inheritance

```
class Set:public List {
public:
    //constructor
    Set( );
    //operation
    void add(int);
};
```

```
Set::Set( ):List( )
{  }
```

```
void Set::add(int newValue)
{ //if not already in set
 if(!includes(newValue))
    //then add
    List::add(newValue);
}
```

Advantages and Disadvantages of Each Mechanism

- Composition is simpler, and clearly indicates what operations are provided.
- Inheritance makes for shorter code, possibly increased functionality, but makes it more difficult to understand what behavior is being provided.
- Inheritance may open to the door for unintended usage, by means of unintended inheritance of behavior.
- Easier to change underlying details using composition (i.e., change the data representation).
- Inheritance may permit polymorphism(多态性)
- Understandability and maintainability are toss-up(难以定夺的事), each has different complexity issues (size versus inheritance tree depth)
- Very small execution time advantage for inheritance