Lecture Notes on
C++ Multi-Paradigm Programming

Bachelor of Software Engineering, Spring 2014

Wan Hai

whwanhai@163.com

13512768378

Software School, Sun Yat-sen University, GZ

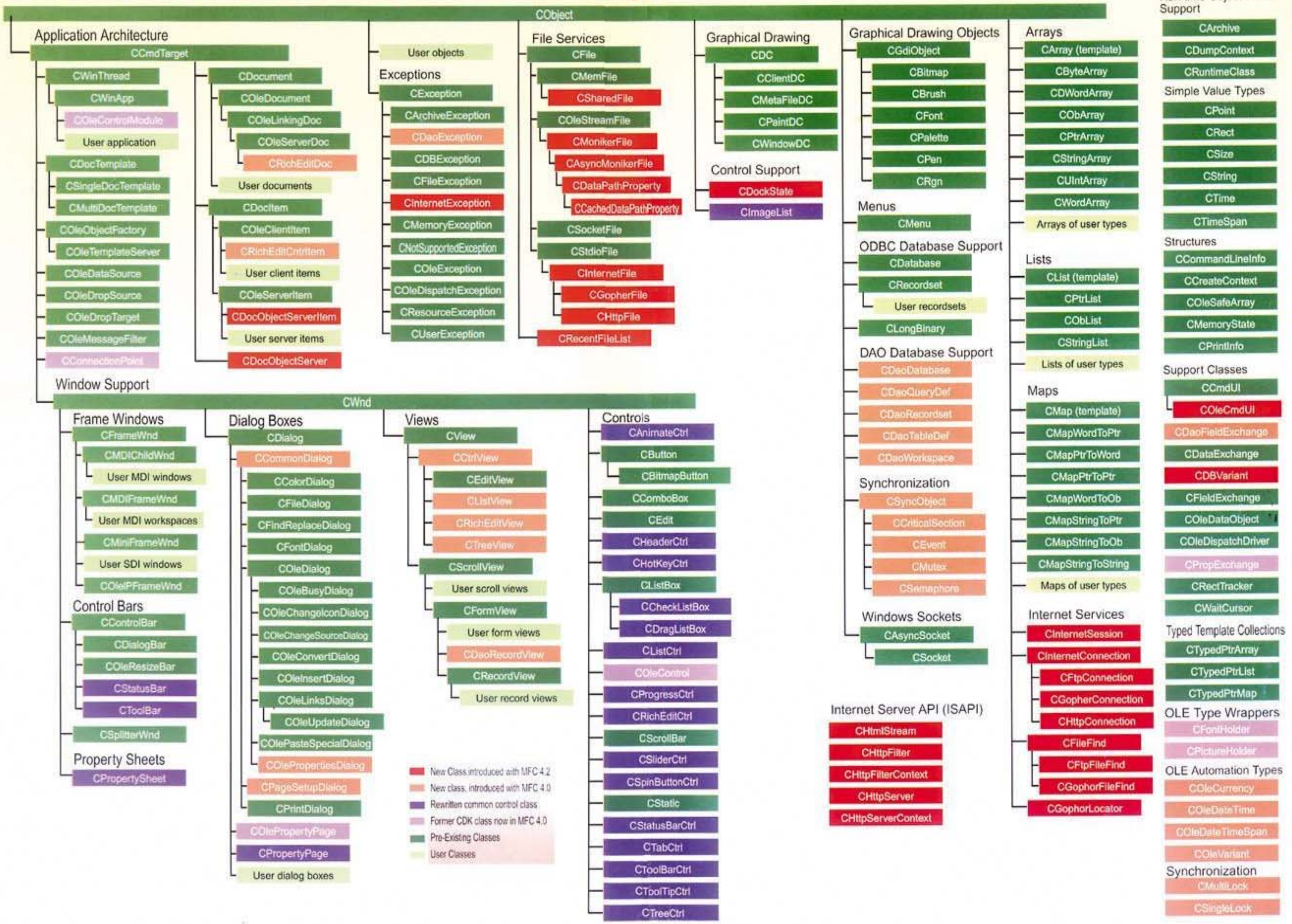
Inheritance

(继承)

Agenda

- Introduction
- Basic Concepts and Syntax
- Conversion between Derived-Class Object and Base-Class Object （派生类与基类的兼容性问题）
- Name Hiding and Overriding（屏蔽和重定义）
- protected Members
- protected and private Inheritance
- Constructors and Destructors Under Inheritance
- Multiple Inheritance
- Relationships between Classes

MFC 4.21 类别组织框架图（Class Hierarchy）



- CObject
  - Application Architecture
    - CCmdTarget
      - CWinThread
        - CWinApp
          - COleControlModule
          - User application
      - CDocTemplate
        - CSingleDocTemplate
        - CMultiDocTemplate
      - COleObjectFactory
        - COleTemplateServer
      - COleDataSource
      - COleDropSource
      - COleDropTarget
      - COleMessageFilter
      - CConnectionPoint
      - CDocument
        - COleDocument
          - COleLinkingDoc
            - COleServerDoc
              - CRichEditDoc
        - User documents
      - CDocItem
        - COleClientItem
          - CRichEditCntrItem
          - User client items
        - COleServerItem
          - CDocObjectServerItem
          - User server items
      - CDocObjectServer
      - Window Support
        - CWnd
          - Frame Windows
            - CFrameWnd
              - CMDIChildWnd
                - User MDI windows
              - CMDIFrameWnd
                - User MDI workspaces
              - CMiniFrameWnd
              - User SDI windows
              - COleIPFrameWnd
          - Control Bars
            - CControlBar
              - CDialogBar
              - COleResizeBar
              - CStatusBar
              - CToolBar
          - CSplitterWnd
          - Property Sheets
            - CPropertySheet
          - Dialog Boxes
            - CDialog
              - CCommonDialog
                - CColorDialog
                - CFileDialog
                - CFindReplaceDialog
                - CFontDialog
                - COleDialog
                  - COleBusyDialog
                  - COleChangeIconDialog
                  - COleChangeSourceDialog
                  - COleConvertDialog
                  - COleInsertDialog
                  - COleLinksDialog
                    - COleUpdateDialog
                  - COlePasteSpecialDialog
                  - COlePropertiesDialog
                - CPageSetupDialog
                - CPrintDialog
              - COlePropertyPage
              - CPropertyPage
              - User dialog boxes
          - Views
            - CView
              - CCtrlView
                - CEditView
                - CListView
                - CRichEditView
                - CTreeView
              - CScrollView
                - User scroll views
                - CFormView
                  - User form views
                  - CDaoRecordView
                  - CRecordView
                    - User record views
          - Controls
            - CAnimateCtrl
            - CButton
              - CBitmapButton
            - CComboBox
            - CEdit
            - CHeaderCtrl
            - CHotKeyCtrl
            - CListBox
              - CCheckListBox
              - CDragListBox
            - CListCtrl
            - COleControl
            - CProgressCtrl
            - CRichEditCtrl
            - CScrollBar
            - CSliderCtrl
            - CSpinButtonCtrl
            - CStatic
            - CStatusBarCtrl
            - CTabCtrl
            - CToolBarCtrl
            - CToolTipCtrl
            - CTreeCtrl
  - User objects
  - Exceptions
    - CException
      - CArchiveException
      - CDaoException
      - CDBException
      - CFileException
      - CInternetException
      - CMemoryException
      - CNotSupportedException
      - COleException
      - COleDispatchException
      - CResourceException
      - CUserException
  - File Services
    - CFile
      - CMemFile
        - CSharedFile
      - COleStreamFile
        - CMonikerFile
          - CAsyncMonikerFile
            - CDataPathProperty
              - CCachedDataPathProperty
      - CSocketFile
      - CStdioFile
        - CInternetFile
          - CGopherFile
          - CHttpFile
    - CRecentFileList
  - Graphical Drawing
    - CDC
      - CClientDC
      - CMetaFileDC
      - CPaintDC
      - CWindowDC
  - Control Support
    - CDockState
    - CImageList
  - Graphical Drawing Objects
    - CGdiObject
      - CBitmap
      - CBrush
      - CFont
      - CPalette
      - CPen
      - CRgn
  - Menus
    - CMenu
  - ODBC Database Support
    - CDatabase
    - CRecordset
      - User recordsets
    - CLongBinary
  - DAO Database Support
    - CDaoDatabase
    - CDaoQueryDef
    - CDaoRecordset
    - CDaoTableDef
    - CDaoWorkspace
  - Synchronization
    - CSyncObject
      - CCriticalSection
      - CEvent
      - CMutex
      - CSemaphore
  - Windows Sockets
    - CAsyncSocket
      - CSocket
  - Arrays
    - CArray (template)
    - CByteArray
    - CDWordArray
    - CObArray
    - CPtrArray
    - CStringArray
    - CUIntArray
    - CWordArray
    - Arrays of user types
  - Lists
    - CList (template)
    - CPtrList
    - CObList
    - CStringList
    - Lists of user types
  - Maps
    - CMap (template)
    - CMapWordToPtr
    - CMapPtrToWord
    - CMapPtrToPtr
    - CMapWordToOb
    - CMapStringToPtr
    - CMapStringToOb
    - CMapStringToString
    - Maps of user types
  - Internet Services
    - CInternetSession
    - CInternetConnection
      - CFtpConnection
      - CGopherConnection
      - CHttpConnection
    - CFileFind
      - CFtpFileFind
      - CGopherFileFind
    - CGophorLocator

Internet Server API (ISAPI)

- CHtmlStream
- CHttpFilter
- CHttpFilterContext
- CHttpServer
- CHttpServerContext

Classes Not Derived From CObject

Run-time Object Model Support

- CArchive
- CDumpContext
- CRuntimeClass

Simple Value Types

- CPoint
- CRect
- CSize
- CString
- CTime
- CTimeSpan

Structures

- CCommandLineInfo
- CCreateContext
- COleSafeArray
- CMemoryState
- CPrintInfo

Support Classes

- CCmdUI
  - COleCmdUI
- CDaoFieldExchange
- CDataExchange
- CDBVariant
- CFieldExchange
- COleDataObject
- COleDispatchDriver
- CPropExchange
- CRectTracker
- CWaitCursor

Typed Template Collections

- CTypedPtrArray
- CTypedPtrList
- CTypedPtrMap

OLE Type Wrappers

- CFontHolder
- CPictureHolder

OLE Automation Types

- COleCurrency
- COleDateTime
- COleDateTimeSpan
- COleVariant

Synchronization

- CMultiLock
- CSingleLock

🟥 New Class introduced with MFC 4.2

🟧 New class, introduced with MFC 4.0

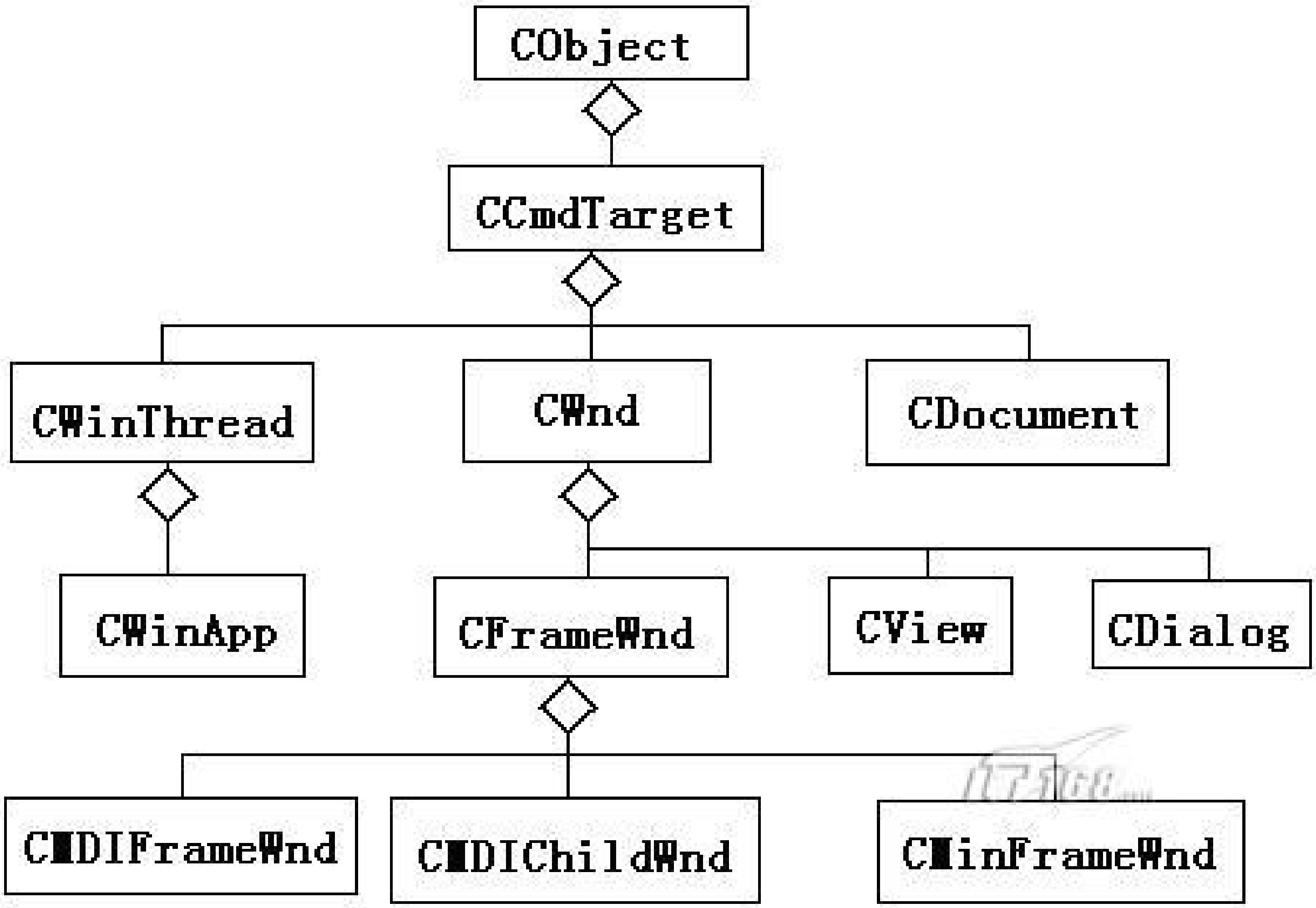
🟪 Rewritten common control class

▨ Former CDK class now in MFC 4.0

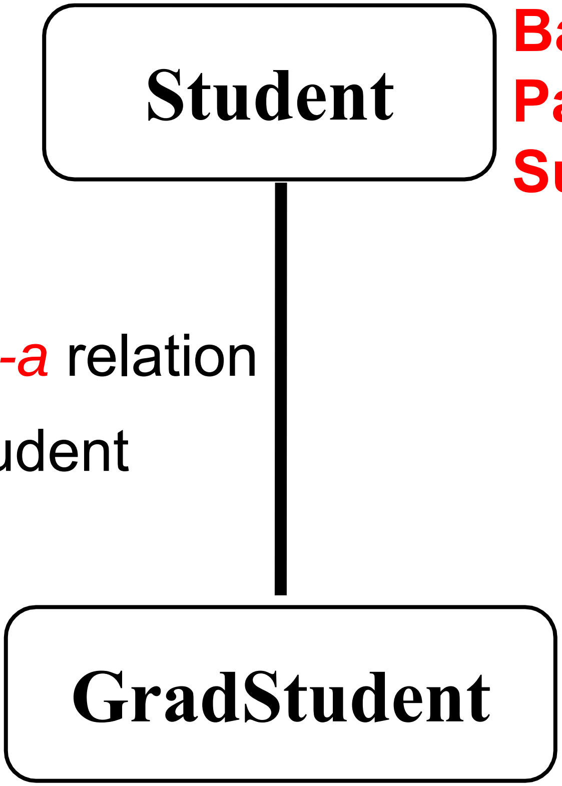
🟩 Pre-Existing Classes

□ User Classes

Introduction



Introduction



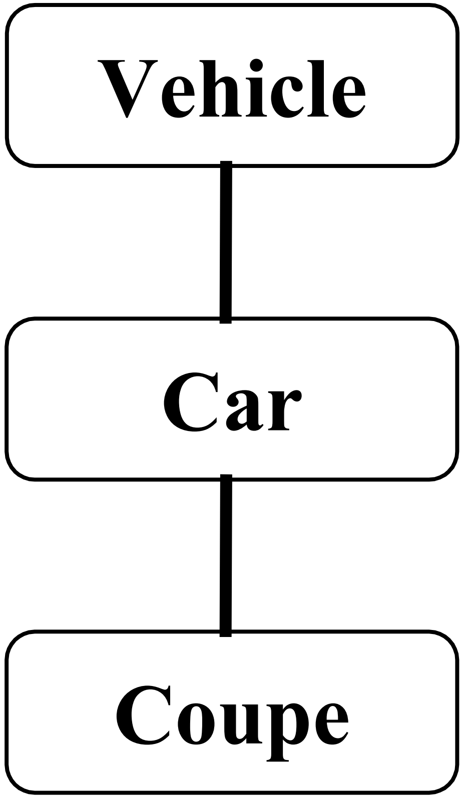
Base class（基类）
Parent class（父类）
Super class（超类）

Inheritance represents Is-a relation

GradStudent *is a* student

Derived class（派生类）
Sub class（子类）

Introduction



Vehicle

Car

Coupe

Direct base class for Car

Indirect base class for Coupe

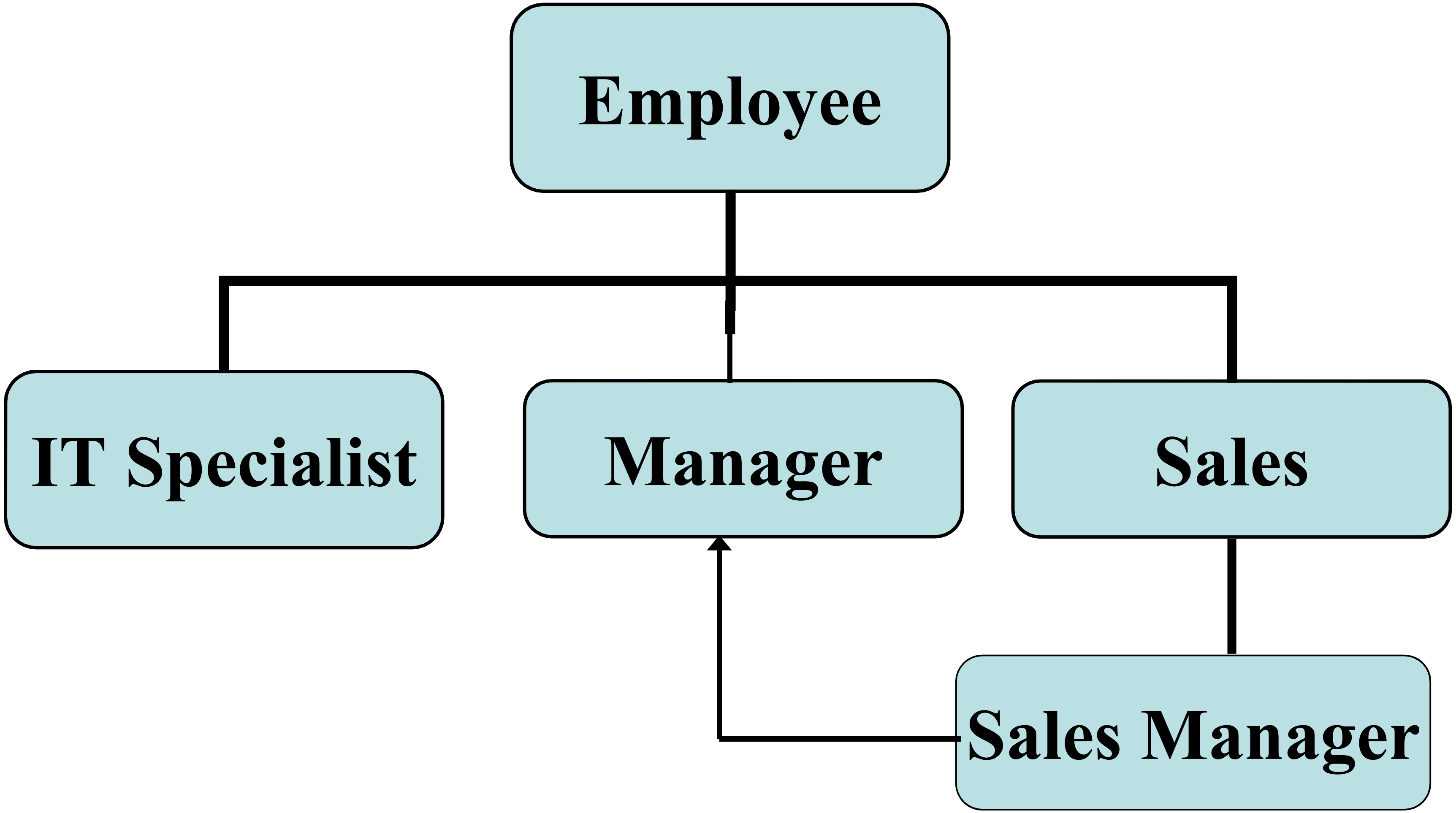
Direct derived class from Vehicle

Direct base class for Coupe

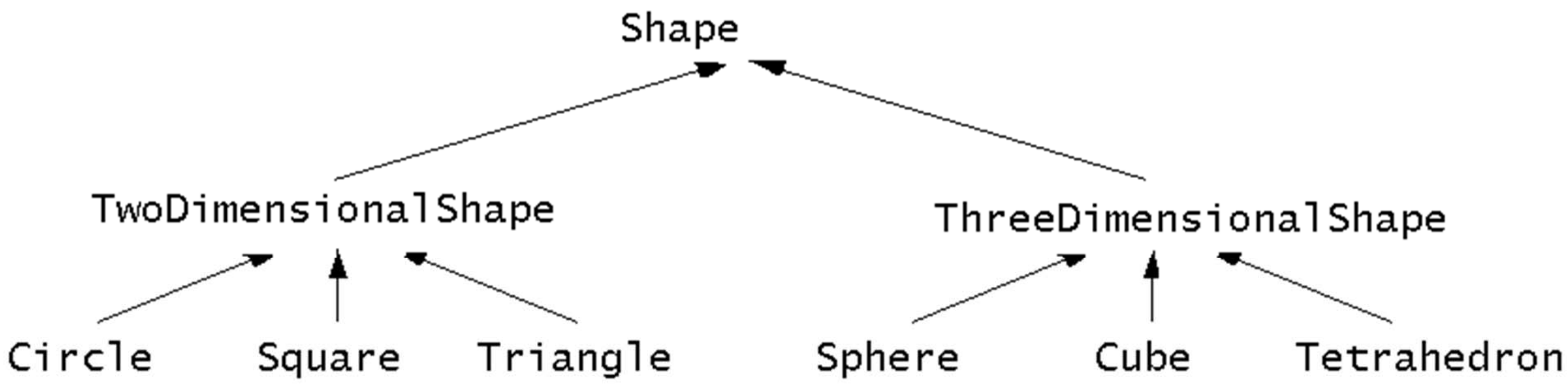
Direct derived class from Car

Indirect derived class from Vehicle

Introduction

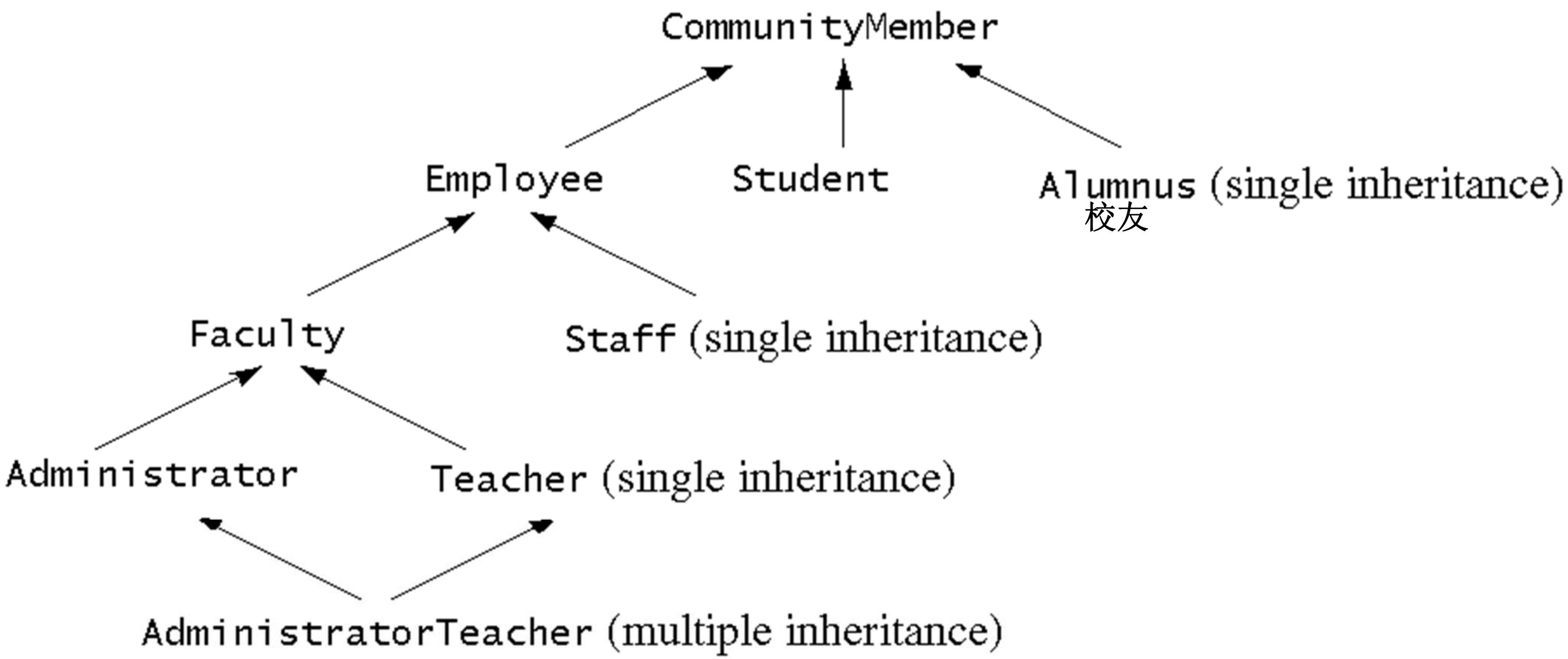


Introduction



A portion of a Shape class hierarchy.

Introduction


CommunityMember
Employee
Student
Alumnus (single inheritance)
校友
Faculty
Staff (single inheritance)
Administrator
Teacher (single inheritance)
AdministratorTeacher (multiple inheritance)

An inheritance hierarchy for university community members.

Introduction

- The notion of a derived class and its associated language mechanisms are provided to express hierarchical relationships, that is, to express commonality between classes.
- It is also the basis for what is commonly called object-oriented programming.

Introduction

- Only with a class concept, there is severe inflexibility in software reusability, evolution, and related concept representation.
- The inheritance mechanism provides a solution to a software reusability, IS-A concept representation, and easy modification.
- Inheritance provides a way to construct a new class via modification (evolution) on one or more existing classes.

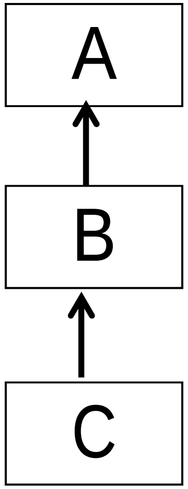
小结

- 使得程序可刻划现实世界的**IS-A**关系。
- 提高程序的可重用性
  - 派生类重用基类类的代码可提高程序开发效率。派生类的定义通常基于设计完善、并经严格测试的基类，从而使程序设计工作建立在一个可靠的基础上，有助于高效地开发出可靠性较高的软件
  - 这种重用是一种灵活的重用方式：子类在继承父类代码的基础上，可根据自己的特性进行调整。

小结

- 定义：“类B继承类A”或“类A派生类B”

  在类B中除了自己定义的成员之外，还自动包括了类A中定义的数据成员与成员函数，这些自动继承下来的成员称为类B的继承成员

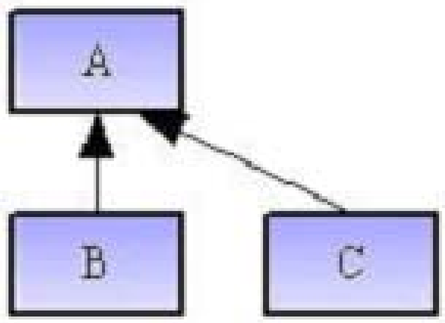


基类 (BASE class)

派生类 (derived class)

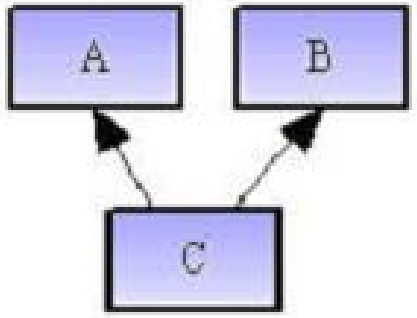
C++所支持的继承形式

单重继承



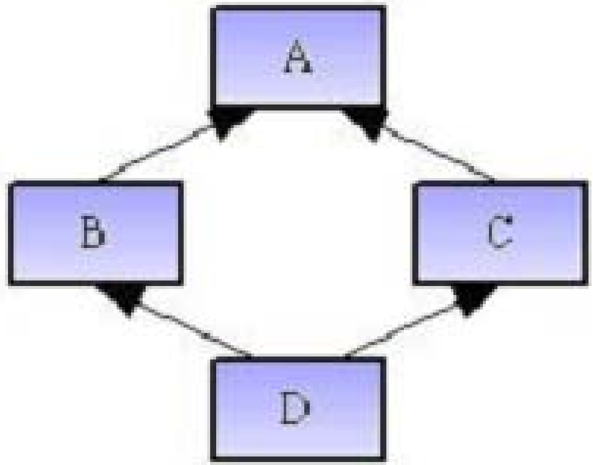
一个派生类只
有一个直接基类

多重继承



一个派生类具有
两个或两个以上
直接基类

重复继承



多重继承的一种特殊形式
派生类2次或2次以上重复
继承某个祖先类

class Time Specification

```
//  SPECIFICATION FILE ( time.h )

class  Time
{
public :
        void  Set ( int  hours , int  minutes , int  seconds ) ;
        void  Increment ( ) ;
        void  Write ( )  const ;
        Time ( int  initHrs, int  initMins,  int  initSecs ) ;  // constructor
        Time ( ) ;     //  default constructor

private :
        int hrs ;
        int mins ;
        int secs ;
} ;
```

class Time Implementation

```
//  IMPLEMENTATION FILE ( time.cpp )

Time::Time(  int initHrs,  int initMins, int initSecs )
{
   hrs = initHrs;
   mins = initMins;
   secs = initSecs;
}

Time::Time()
{
   hrs = 0;
   mins = 0;
   secs = 0;
}
```

class Time Implementation

```
//  IMPLEMENTATION FILE ( time.cpp )

void Time::Set( int hours, int minutes, int seconds )
{
   hrs = hours;
   mins = minutes;
   secs = seconds;
}
```

`class Time` Implementation

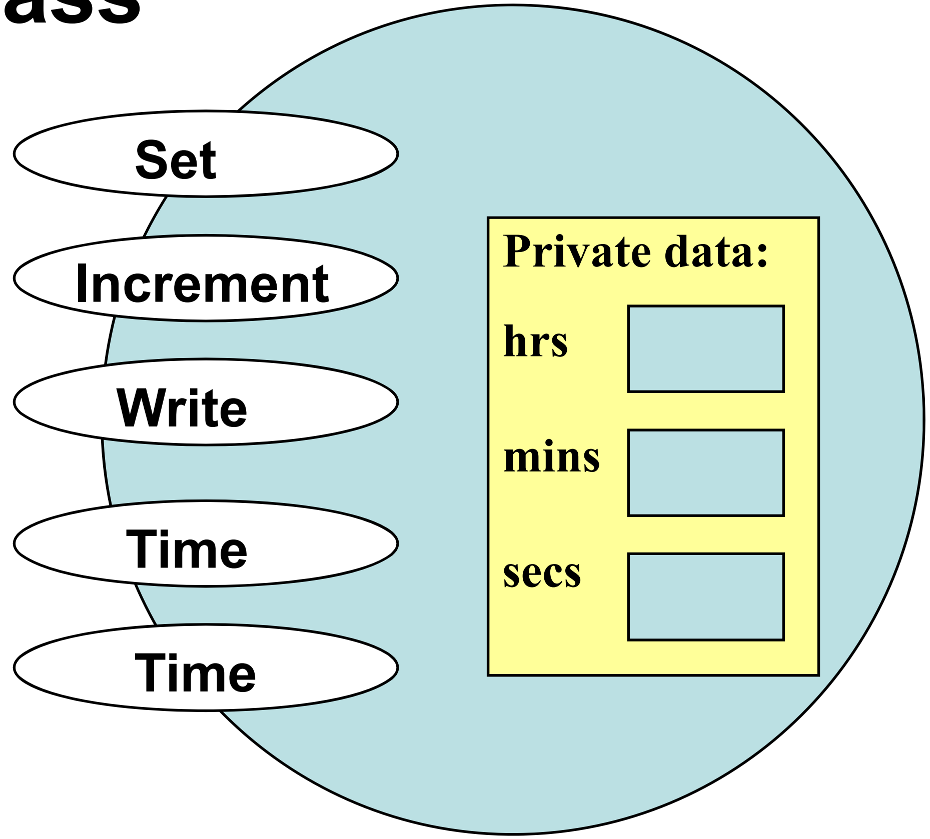
```
void Time::Increment()    //  IMPLEMENTATION FILE ( time.cpp )
{
   secs++;
   if (secs > 59)
   {
      secs = 0;
      mins++;
      if (mins > 59)
      {
         mins = 0;
         hrs++;
         if (hrs > 23)
            hrs = 0;
      }
   }
}
```

class Time Implementation

```
//  IMPLEMENTATION FILE ( time.cpp )

void Time::Write() const
{
   if (hrs < 10)
      cout << '0';
   cout << hrs << ':';
   if (mins < 10)
      cout << '0';
   cout << mins << ':';
   if (secs < 10)
      cout << '0';
   cout << secs;
}
```

Class Interface Diagram

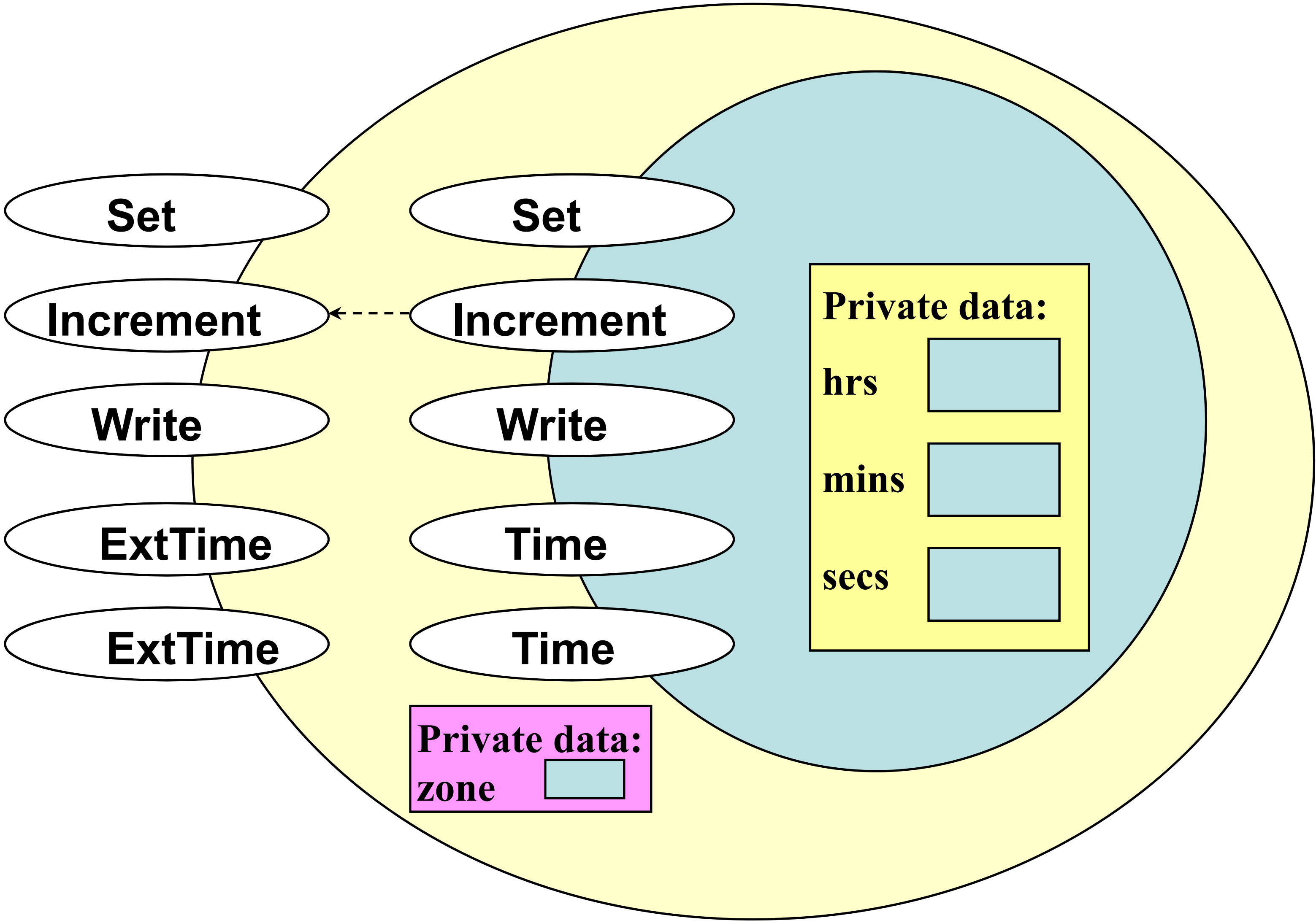
Time class



Construct a derived class by Inheritance

```
// SPECIFICATION   FILE                              ( exttime.h)
#include   “time.h”
enum  ZoneType {EST, CST, MST, PST, EDT, CDT, MDT, PDT } ;

class  ExtTime  :  public  Time        // Time is the base class
{
public :
      ExtTime ( int  initHrs ,  int  initMins ,  int  initSecs ,
                    ZoneType    initZone ) ;      // constructor
      ExtTime ( ) ;                                // default constructor
      void Set ( int  hours,  int  minutes,  int   seconds ,
                    ZoneType   timeZone ) ;
      void Write ( )  const ;

private :
         ZoneType  zone ;      //  added data member
} ;
```

Interface Diagram of class ExtTime



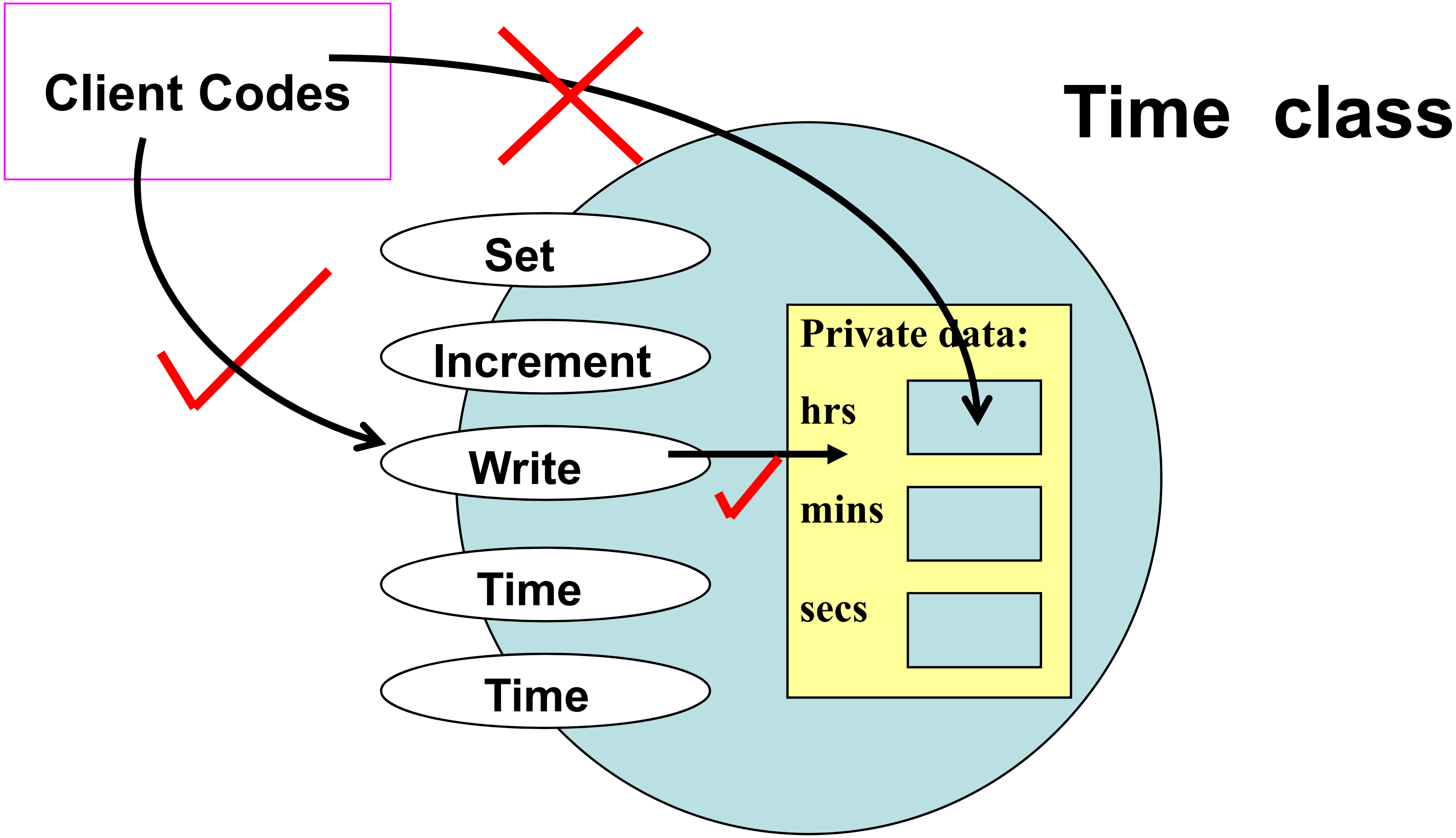
继承的语法

- 单重继承的定义形式

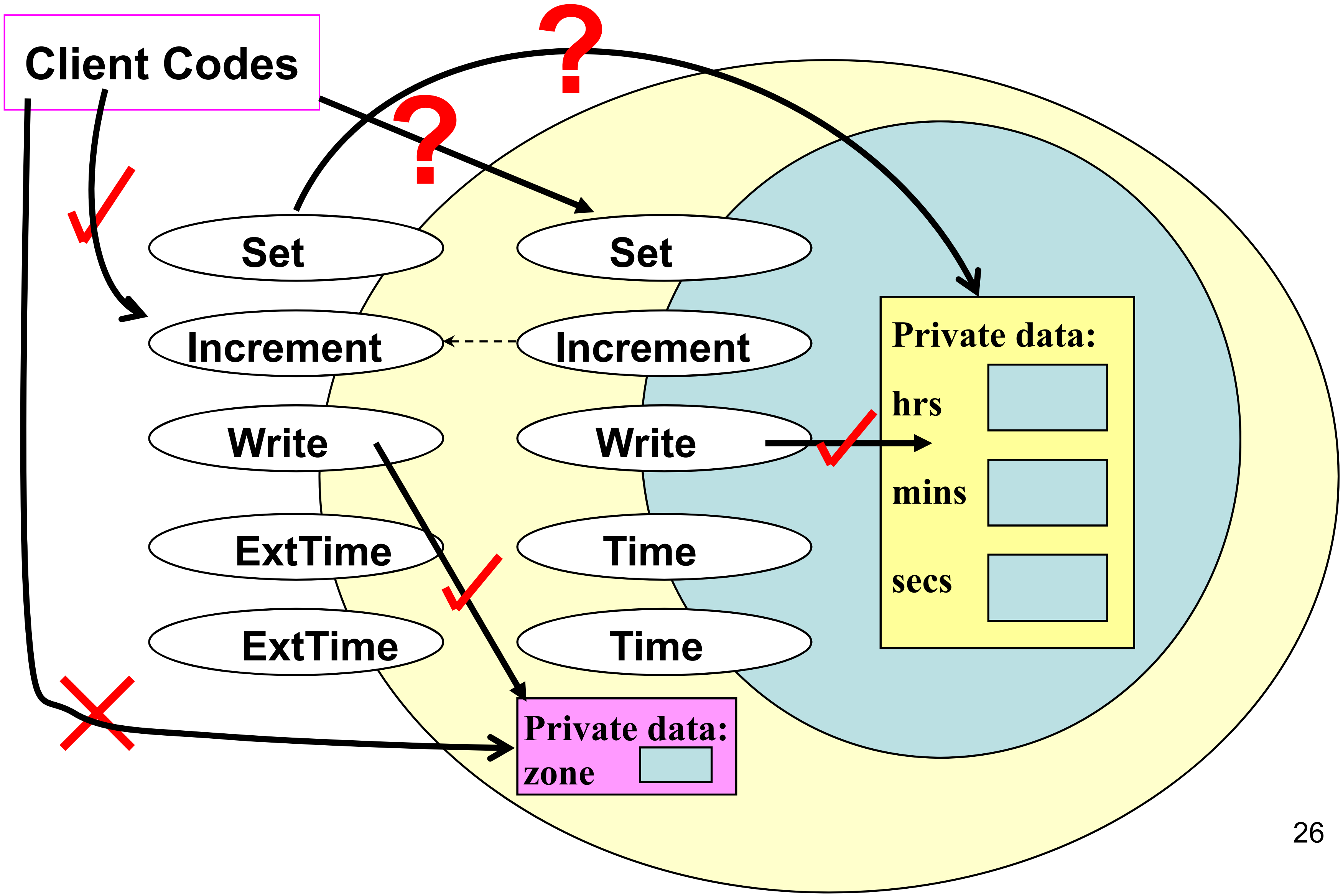
```
class 派生类名：继承访问控制   基类类名{
 成员访问控制：
       成员声明列表；
 }；
```

- 继承访问控制和成员访问控制均由保留**public、protected、private**来定义，缺省均为**private**。

Class Interface Diagram



Interface Diagram of class ExtTime



新成员的访问控制

- **private**（私有的）：

  在**private**后声明的成员称为私有成员，私有成员只能通过本类的成员函数来访问。

- **public**(公有的）：

  在**public**后声明的成员称为公有成员，公有成员用于描述一个类与外部世界的接口，类的外部（程序的其它部分的代码）可以访问公有成员。

- **protected**（受保护的）：

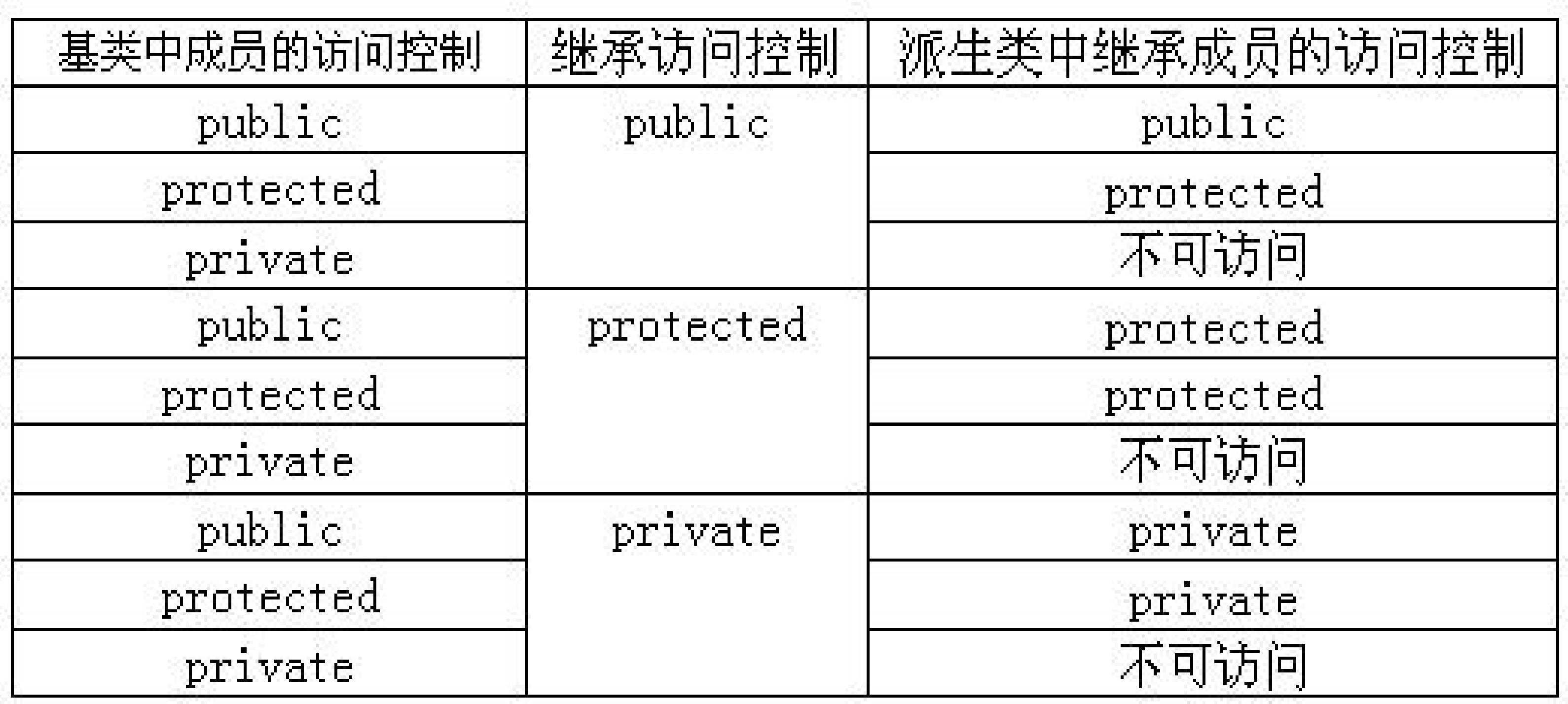
  受保护成员具有**private**与**public**的双重角色：对派生类的成员函数而言，它为**public**，而对类的外部而言，它为**private**。即：**protected**成员只能由本类及其后代类的成员函数访问。

继承成员的访问控制

- 影响继承成员（派生类从基类中继承而来的成员）访问控制方式的两个因素：
  - 定义派生类时指定的继承访问控制
  - 该成员在基类中所具有的成员访问控制

```
  class B：继承访问控制 A {
   成员访问控制：
         成员声明列表；
   } ；
```

继承成员的访问控制规则



基类中成员的访问控制	继承访问控制	派生类中继承成员的访问控制
public	public	public
protected		protected
private		不可访问
public	protected	protected
protected		protected
private		不可访问
public	private	private
protected		private
private		不可访问

- 无论采用什么继承方式，基类的私有成员在派生类中都是不可访问的。
- “私有”和“不可访问”有区别：私有成员可以由派生类本身访问，不可访问成员即使是派生类本身也不能访问。

继承成员的访问控制

- 在大多数情况下，使用**public**的继承方式；**private**和**protected**是很少使用的。

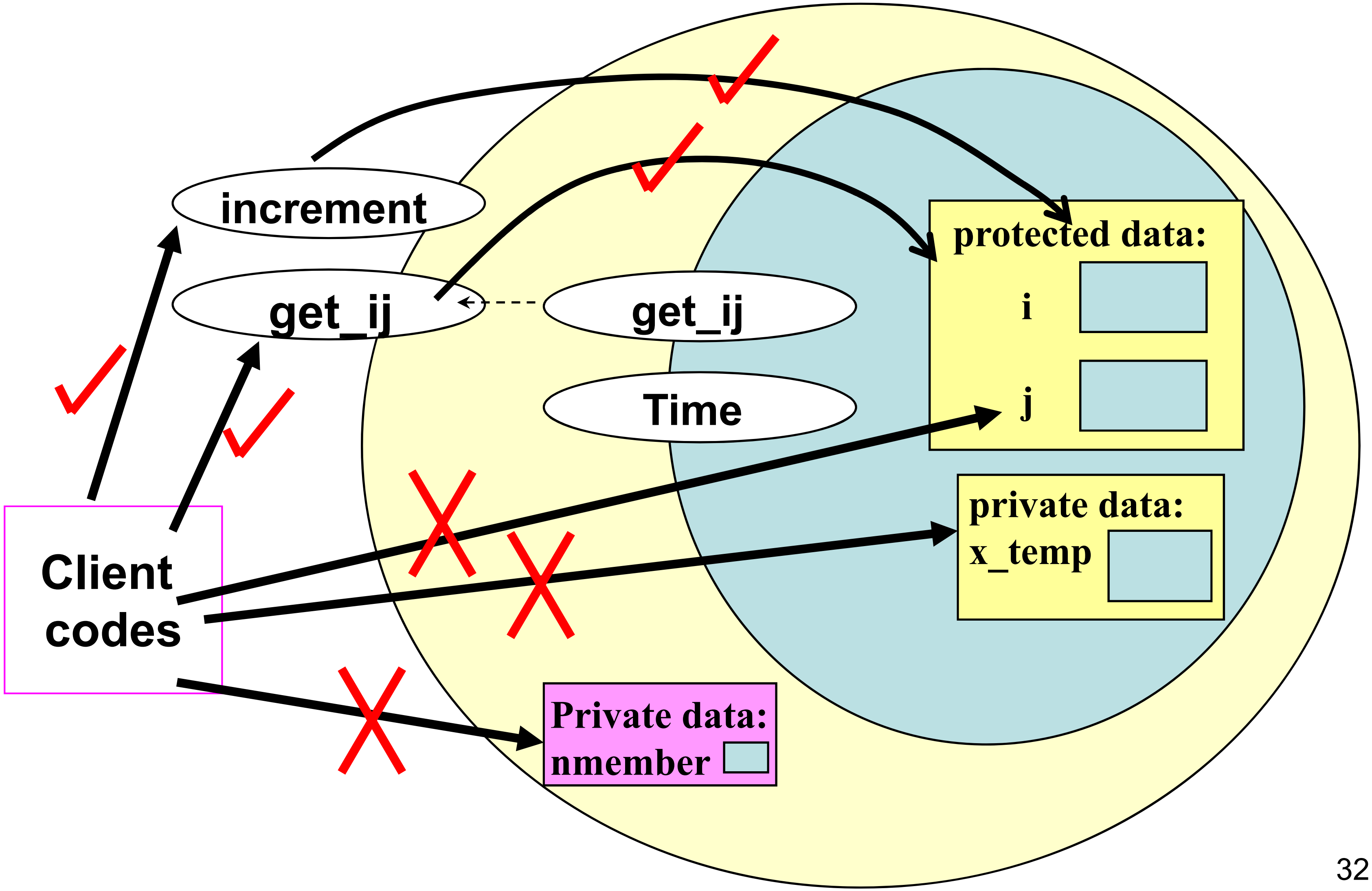
✓ 微软的**MFC**：全部使用**public**的继承方式

✓ **AT&T**的**iostream**库：**95%**以上使用的是**public**的继承方式

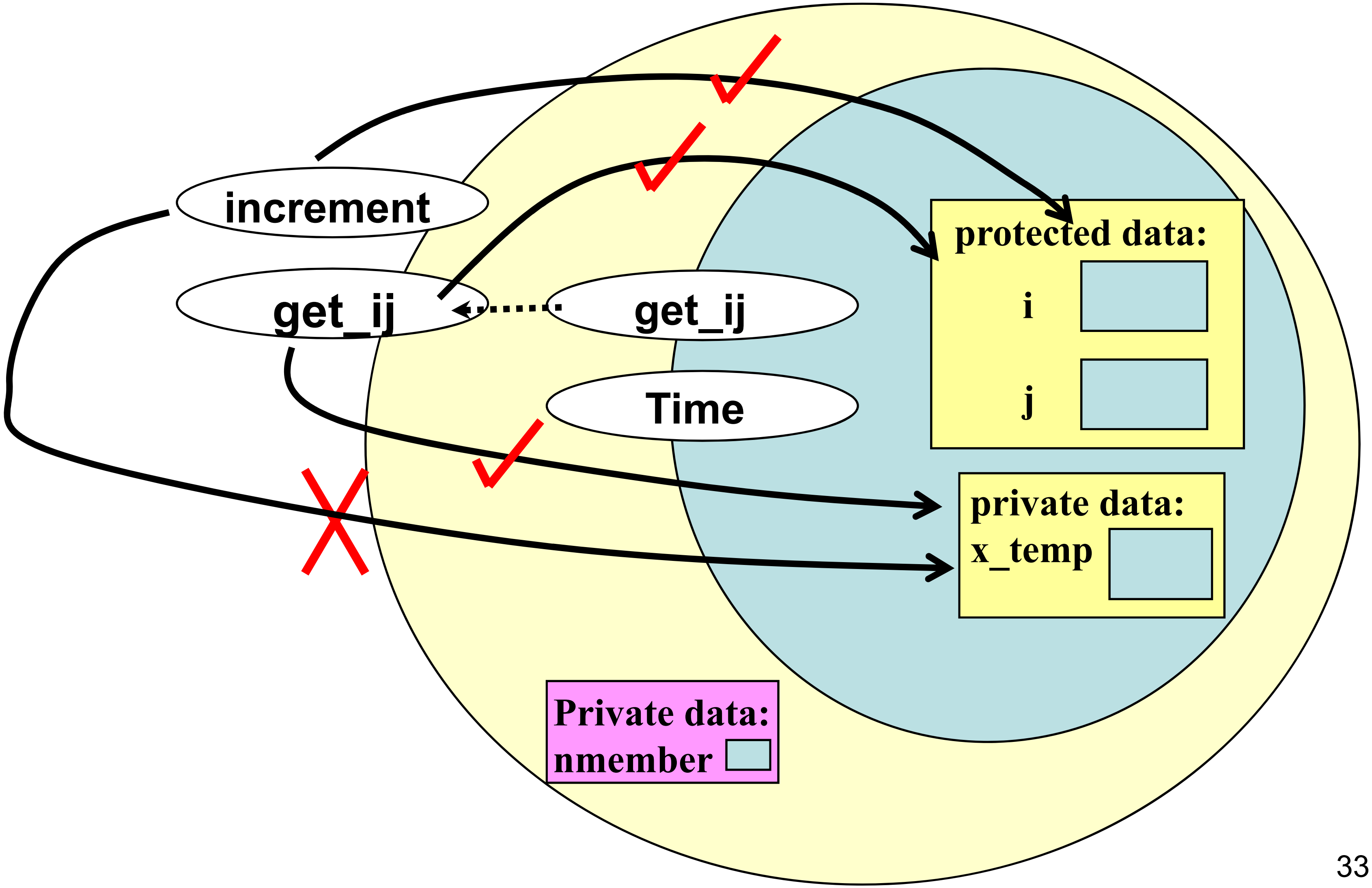
例

```
class BASE
{
      public:        BASE();
                     void get_ij();
      protected:  int i, j;
      private:      int x_temp;
};

//公有派生：在Y1类中，i、j是受保护成员
class Y1:public BASE
{
 public:
     void increment();   //get_ij()是公有成员，x_temp不可访问
 private:
      float  nmember;
};
```

Interface Diagram of class Y1



Interface Diagram of class Y1



访问控制

```
BASE::BASE()
{ i=0; j=0; x_temp=0; }

void BASE:: get_ij()
{
      cout << i << ‘ ‘ << j << endl;
}

void Y1::increment()
{
     i++; j++;
}
```

```
int main()  //程序Access
{
    BASE  obj1;
    Y1       obj2;

    obj2.increment();
    obj2.get_ij();
    obj1.get_ij();
}
```

运行程序 屏幕显示：

```
1 1
0 0
```

例

```
class BASE{
          protected:      int i, j;
          public:         void get_ij();
          private:        int x_temp;
};
```

保护派生：在Y2类中，i、j是受保护成员。get_ij()变成受保护成员，x_temp不可访问

```
class Y2:protected BASE{  … }；
```

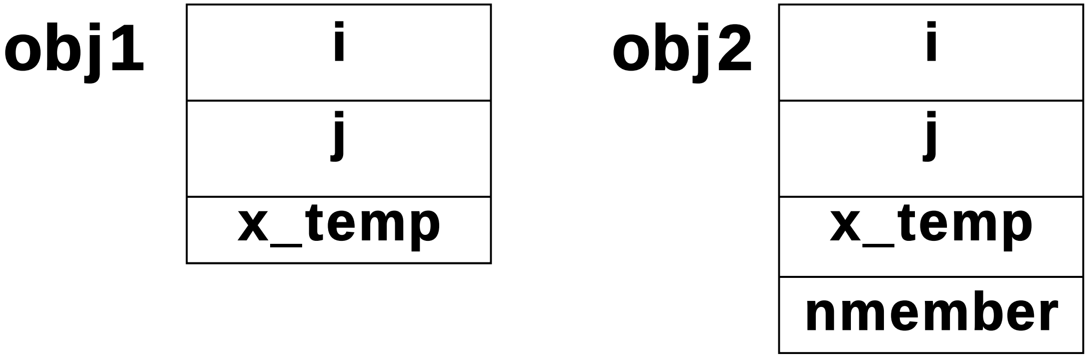
私有派生：在Y3类中，i、j、 get_ij()都变成私有成员，x_temp不可访问

```
class Y3:private BASE{  … };
```

派生类对象的存储

- 派生类的对象不仅存放了在派生类中定义的非静态数据成员，而且也存放了从基类中继承下来的非静态数据成员
- 可以认为派生类对象中包含了基类子对象。

```
BASE  obj1;
Y1      obj2;
```



obj1
i
j
x_temp

obj2
i
j
x_temp
nmember

继承与构造函数、析构函数

- 继承时的构造函数与析构函数
- 构造函数与析构函数调用次序
- 向基类构造函数传递实参

继承时的构造函数

- 基类的构造函数不被继承，派生类中需要声明自己的构造函数。
- 派生类的构造函数中只需要对本类中新增成员进行初始化即可。对继承来的基类成员的初始化是通过自动调用基类构造函数完成的。
- 派生类的构造函数需要给基类的构造函数传递参数。

构造函数的调用次序

- 构造函数的调用次序（创建派生类对象时）
  - 首先调用其基类的构造函数（调用顺序按照基类被继承时的声明顺序（从左向右））。
  - 然后调用本类对象成员的构造函数（调用顺序按照对象成员在类中的声明顺序）。
  - 最后调用本类的构造函数。

析构函数的调用次序

- 撤销派生类对象时析构函数的调用次序与构造函数的调用次序相反
  - 首先调用本类的析构函数
  - 然后调用本类对象成员的析构函数
  - 最后调用其基类的析构函数

[例]

```
//Demo.h

class C {
public:
        C( );   //构造函数
        ~C( ); //析构函数
};

class BASE {
public:
         BASE( );   // 构造函数
         ~BASE( )  // 析构函数
};
```

[例]

```
#include “Demo.h”                                    //Demo.cpp

C::C( )   //构造函数
{   cout << "Constructing C object.\n";      }

C:: ~C( )  //析构函数
{   cout << "Destructing C object.\n";       }

BASE::BASE( )   // 构造函数
{   cout << "Constructing BASE object.\n";  }

BASE:: ~BASE( )  // 析构函数
{   cout << "Destructing BASE object.\n"; }
```

[例]

```
class DERIVED: public BASE {                    // Derived.h
public:
    DERIVED()         // 构造函数
   ~DERIVED()         // 析构函数
private:
C   mOBJ;
};
```

```
#include “Derived.h”                            // Derived.cpp

DERIVED::DERIVED()          // 构造函数
{  cout << "Constructing derived object.\n";  }

DERIVED:: ~DERIVED()        // 析构函数
{  cout << "Destructing derived object.\n";    }
```

[例]

```
#include “Derived.h”  // Client.cpp

int main()
{
    DERIVED obj;  // 声明一个派生类的对象
                  // 什么也不做，仅完成对象obj的构造与析构
    return 0;
}
```

运行结果

```
Constructing BASE object.
Constructing C object.
Constructing derived object.
Destructing derived object.
Destructing C object.
Destructing BASE object.
```

class Time Specification

```
//  SPECIFICATION FILE ( time.h )

class  Time
{
public :
     void  Set ( int  hours , int  minutes , int  seconds ) ;
     void  Increment ( ) ;
     void  Write ( )  const ;
     Time ( int  initHrs, int  initMins,  int  initSecs ) ;  // constructor
     Time ( ) ;     //  default constructor

private :
     int hrs ;
     int mins ;
     int secs ;
} ;
```

class Time Implementation

```
//  IMPLEMENTATION FILE ( time.cpp )

Time::Time(  int initHrs,  int initMins, int initSecs )
{
   hrs = initHrs;
   mins = initMins;
   secs = initSecs;
}

Time::Time()
{
   hrs = 0;
   mins = 0;
   secs = 0;
}
```

class Time Implementation

```
//  IMPLEMENTATION FILE ( time.cpp )

void Time::Set( int hours, int minutes, int seconds )
{
   hrs = hours;
   mins = minutes;
   secs = seconds;
}
```

`class Time` Implementation

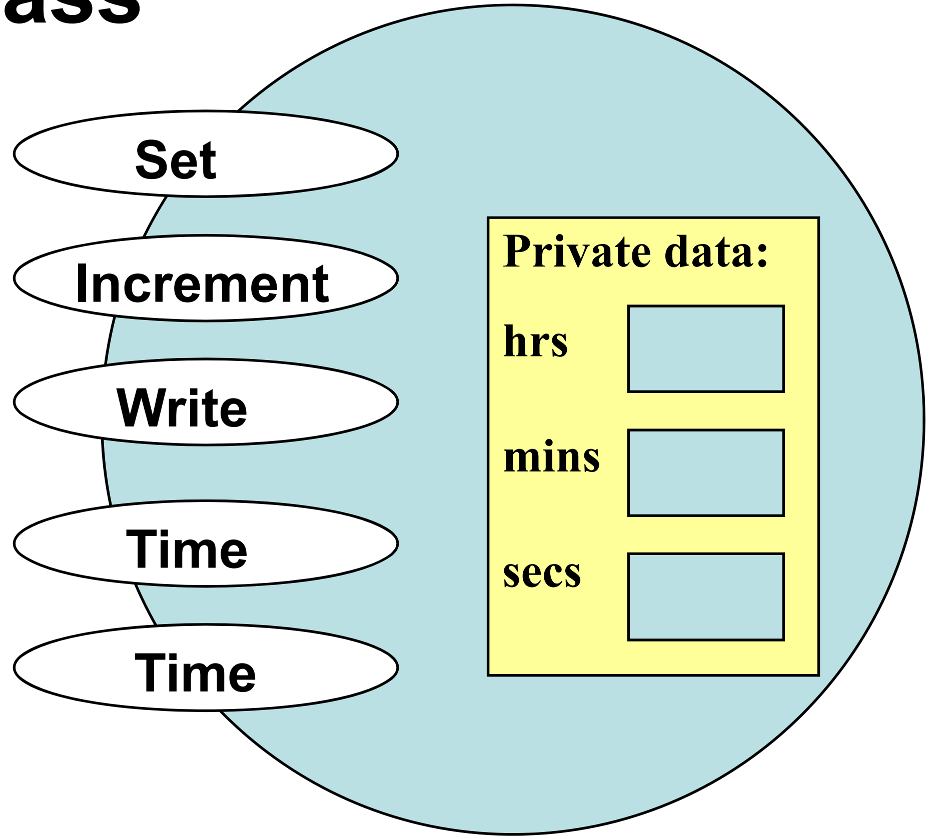
```
void Time::Increment()    //  IMPLEMENTATION FILE ( time.cpp )
{
   secs++;
   if (secs > 59)
   {
      secs = 0;
      mins++;
      if (mins > 59)
      {
         mins = 0;
         hrs++;
         if (hrs > 23)
            hrs = 0;
      }
   }
}
```

class Time Implementation

```
//  IMPLEMENTATION FILE ( time.cpp )

void Time::Write() const
{
   if (hrs < 10)
      cout << '0';
   cout << hrs << ':';
   if (mins < 10)
      cout << '0';
   cout << mins << ':';
   if (secs < 10)
      cout << '0';
   cout << secs;
}
```

Class Interface Diagram

Time class



Construct a derived class by Inheritance

```
// SPECIFICATION   FILE                        ( exttime.h)
#include   “time.h”
enum  ZoneType {EST, CST, MST, PST, EDT, CDT, MDT, PDT } ;

class  ExtTime  :  public  Time       // Time is the base class
{
public :
        ExtTime ( int  initHrs ,  int  initMins ,  int  initSecs ,
                         ZoneType    initZone ) ;      // constructor
        ExtTime ( ) ;                                    // default constructor
        void Set ( int  hours,  int  minutes,  int   seconds ,
                         ZoneType   timeZone ) ;
        void Write ( )  const ;

private :
            ZoneType  zone ;      //  added data member
} ;
```

向基类构造函数传递实参

- 若基类构造函数带参数，则定义派生类构造函数时通过初始化列表显式调用基类构造函数，并向基类构造函数传递实参。
- 带初始化列表的派生类构造函数的一般形式

```
派生类名(形参表) : 基类名(实参表)
{
      派生类新成员初始化赋值语句;
};
```

Construct a derived class by Inheritance

```
// IMPLEMENTATION FILE (exttime.cpp)

ExtTime::ExtTime(  int initHrs,
                   int initMins,
                   int initSecs,
                   ZoneType initZone )
                 : Time(initHrs, initMins, initSecs)
{
    zone = initZone;
}
```

1. Passed to the base class constructor.
2. Base class constructor is called prior to the derived class constructor.

Construct a derived class by Inheritance

```
//base class default constructor is called prior to the derived
//class default constructor.

ExtTime::ExtTime()
{
   zone = EST;
}
```

Construct a derived class by Inheritance

```
void ExtTime::Set( int hours,
                   int minutes,
                   int seconds,
                   ZoneType timeZone )
{
   Time::Set(hours, minutes, seconds);   //调用基类函数。Why?
   zone = timeZone;
}

void ExtTime::Write() const
{
   static string zoneString[8] =
   {  "EST", "CST", "MST", "PST", "EDT", "CDT", "MDT", "PDT“ };

   Time::Write();
   cout << ' ' << zoneString[zone];
}
```

Client code

```
#include "exttime.h"
#include <iostream>
#include "time.h"

using namespace std;

int main()
{
   ExtTime time1(5, 30, 0, CDT);
   ExtTime time2;
   int     count;

   cout << "time1: ";
   time1.Write();
   cout << endl;
           :
```

Client code

```
int main()
{
                      :


   cout << "time2: ";
   time2.Write();
   cout << endl;

   time2.Set(23, 59, 55, PST);
   cout << "New time2: ";
   time2.Write();
   cout << endl;


                      :
```

Client code

```
int main()
{
                :
   cout << "Incrementing time2:" << endl;
   for (count = 1; count <= 10; count++)
   {
      time2.Write();
      cout << endl;
      time2.Increment();
   }

   Time time3(1,2,3);
   cout << "time3: ";
   time3.Write();
   cout << endl << endl;
                :
```

Client code

```
int main()
{
                  :

   //客户代码直接访问派生类继承的基类的public 成员

   time1.Time::Set(3,4,5);
   time1.Time::Write();

   cout << endl;

   return 0;
}
```

Base.h

```
class BASE {
   public:
         BASE(int p1, int p2);

         int inc1();
         int inc2();

         void display();

    private:
         int mem1, mem2;
};
```

BASE.cpp

```
#include “BASE.h”

BASE::BASE(int p1, int p2)
{ mem1 = p1; mem2 = p2;}

int BASE::inc1() { return ++mem1; }

int BASE::inc2() { return ++mem2; }

void BASE::display()
{
    cout << "mem1 = “ << mem1
            << ", mem2 = “ << mem2 << endl;
}
```

Derived.h

```
#include “base.h”

class DERIVED : public BASE{
    public:
            DERIVED(int x1, int x2, int x3, int x4, int x5);
            int inc1() ;
            int inc3( ) ;
            void display( ) ;

    private:
            int mem3;
            BASE mem4;
};
```

Derived.cpp

```
#include “Derived.h”
DERIVED::DERIVED(int x1, int x2, int x3, int x4, int x5):
                BASE(x1,x2), mem4(x3,x4)
{ mem3 = x5 ; }

int DERIVED::inc1() { return  BASE::inc1();}

int DERIVED::inc3( ) { return ++mem3 ; }

void DERIVED::display( )
{          BASE::display();
           mem4.display();
           cout<<"mem3 = "<<mem3<<"\n";     }
};
```

[例]

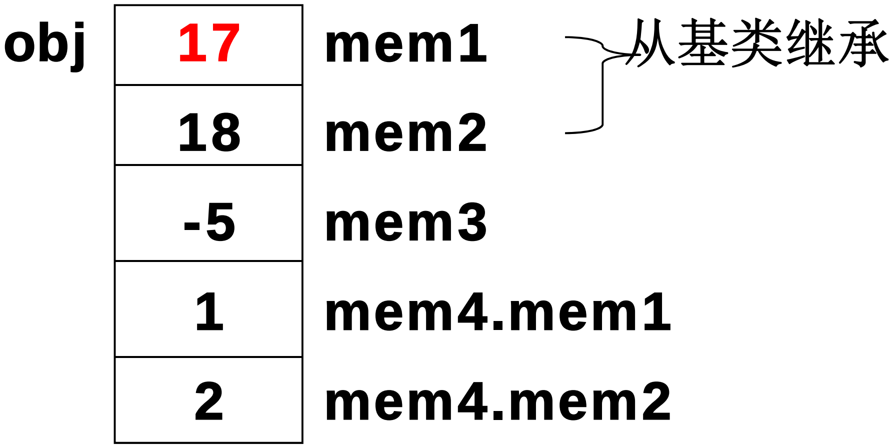
```
#include “Derived.h”
int main()
{
    DERIVED obj( 17, 18, 1, 2, -5);
    obj.inc1();
    obj.display();

    return 0;
}
```

```
mem1 = 18, mem2 = 18
mem1 = 1, mem2 = 2
mem3 = -5
```

运行结果

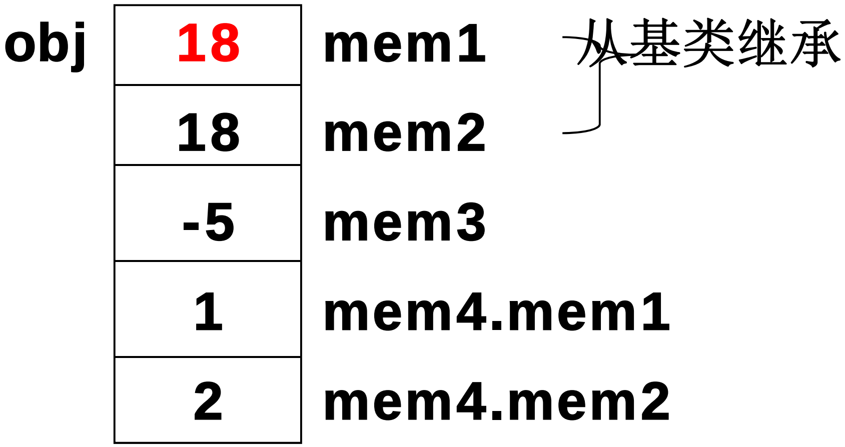
```
mem1 = 18, mem2 = 18
mem1 = 1, mem2 = 2
mem3 = -5
```

对象的存储



obj		
17	mem1	从基类继承
18	mem2	
-5	mem3	
1	mem4.mem1	
2	mem4.mem2	

执行**DERIVED obj(17, 18, 1, 2, -5);**之后

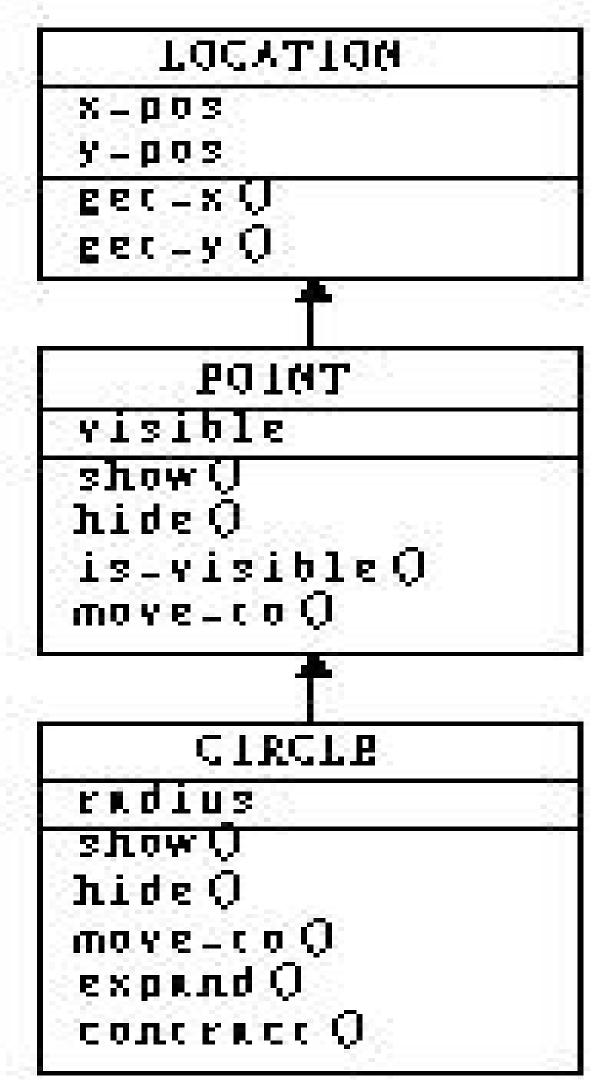


obj		
18	mem1	从基类继承
18	mem2	
-5	mem3	
1	mem4.mem1	
2	mem4.mem2	

执行**obj.inc1();**之后

[例]继承机制的应用

- 图形元素的处理：将圆看作是一种带有半径的点，而将点看作是一种带有显示状态的位置，利用继承机制可将这三个类组织为如图所示的类层次。



LOCATION
x-pos
y-pos
get-x()
get-y()

POINT
visible
show()
hide()
is-visible()
move-to()

CIRCLE
radius
show()
hide()
move-to()
expand()
contract()

BASGRAPH.H

```
//说明：类LOCATION以x和y坐标描述了计算机屏幕上的一个位置。

#include <graphics.h>
class LOCATION {
public:
    LOCATION(int x, int y); // 构造函数，将当前位置设置为(x, y)

    int get_x();  // 返回当前位置的x坐标

    int get_y();  // 返回当前位置的y坐标

protected:
    // 位置的内部状态，在LOCATION的派生类中需要访问
     int x_pos, y_pos;
};
```

BASGRAPH.H

```
// 说明：类POINT描述了某一个位置是隐藏的还是显示的。
// 以public继承表示x_pos和y_pos在POINT中是protected
class POINT: public LOCATION {
public:
        POINT(int x, int y); // 构造函数，初始化位置为(x, y)

       // 判断当前点是否已显示，是则返回TRUE，否则返回FALSE
       BOOLEAN is_visible();
       void show();     // 在当前位置显示点
       void hide();      // 将点隐藏起来
       // 将当前点移动到新位置(x, y)并显示它
       void move_to(int x, int y);
protected:
      // 点的内部状态，在POINT的派生类中需要访问
       BOOLEAN visible;
};
```

BASGRAPH.H

```
// 说明：类CIRCLE描述了一个在屏幕上由POINT派生出来的圆。
// 由POINT类派生，从而也继承了LOCATION类
class CIRCLE: public POINT {
public:
        // 构造函数，初始化圆心为(x, y)，半径为r
        CIRCLE(int x, int y, int r);
        void show(); // 在屏幕上画出圆
        void hide(); // 将圆隐藏起来
        void move_to(int x, int y); // 将当前圆移动到新位置(x, y)
        // 放大圆，使得新的半径为(r + delta)
        void expand(int delta);
        // 缩小圆，使得新的半径为(r - delta)
        void contract(int delta);
protected:
        int radius;   // 圆的半径，在CIRCLE的派生类中可以访问
};
```

LOCATION.CPP

```
#include "basgraph.h"

LOCATION::LOCATION(int x, int y)
{
        x_pos = x;
        y_pos = y;
}

int LOCATION::get_x()
{
        return x_pos;
}

int LOCATION::get_y()
{
        return y_pos;
}
```

LOCATION.CPP

```
#include "basgraph.h"

POINT::POINT(int x, int y): LOCATION(x, y)
{
        visible = FALSE;  // 缺省情况下不显示
}

BOOLEAN POINT::is_visible()
{
        return visible;
}
```

LOCATION.CPP

```
void POINT::show()
{
     if (! is_visible()) {
        visible = TRUE;
         putpixel(x_pos, y_pos, getcolor()); // 使用缺省颜色画点
     }
}

void POINT::hide()
{
    if (is_visible()) {
         visible = FALSE;
         // 使用背景颜色画点，即擦除该点
        putpixel(x_pos, y_pos, getbkcolor());
    }
}
```

LOCATION.CPP

```
void POINT::move_to(int x, int y)
{
        hide();         // 首先使当前点不可见

        x_pos = x;      // 改变当前点的x和y坐标
        y_pos = y;

        show();         // 在新位置显示点

}
```

CIRCLE.CPP

```
#include "basgraph.h"

CIRCLE::CIRCLE(int x, int y, int r): POINT(x, y)
{
            radius = r;
}

void CIRCLE::show()
{
      if (! is_visible())
     {
            visible = TRUE;   // 改变圆的内部状态
            // 画圆，(x_pos, y_pos)为圆心、radius为半径
           circle(x_pos, y_pos, radius);
     }
}
```

CIRCLE.CPP

```
void CIRCLE::hide()
{
    unsigned int temp_color;  // 用于保存当前颜色的临时变量

    if (is_visible())
   {
       temp_color = getcolor();  // 保存当前的缺省颜色
       setcolor(getbkcolor());    // 设置当前颜色为背景颜色
       visible = FALSE;                  // 改变圆的内部状态
       circle(x_pos, y_pos, radius);// 用背景颜色画圆，即擦除圆
       setcolor(temp_color);            // 恢复原来的缺省颜色
    }
}
```

CIRCLE.CPP

```
void CIRCLE::move_to(int x, int y)
{
      hide();          // 擦除旧的圆
      x_pos = x;   // 设置新的位置
      y_pos = y;
      show();         // 在新的位置画圆
}
void CIRCLE::expand(int delta)
{
      hide();         // 擦除旧的圆
      radius = radius + delta;        // 扩大半径
      if (radius < 0)  radius = 0;      // 避免半径为负数
      show();        // 按新的半径画圆
}
void CIRCLE::contract(int delta)
{
     expand(-delta);   // 利用expand()成员函数实现contract()
}
```

GRAFDEMO.CPP

```
#include "basgraph.h"  // 基本图形元素的类界面
#include <conio.h>       // 利用其中的getch()函数暂停
int main()
{
    int graphdriver = DETECT, graphmode ; // 初始化图形系统所需变量
   // 声明一个圆，圆心在(100, 200)，半径为50
   CIRCLE circle(100, 200, 50);
   initgraph(&graphdriver, &graphmode, ""); // 初始化图形系统
   circle.show();  // 声明一个圆并显示它
   circle.move_to(200, 250);  // 移动圆
   circle.expand(50); // 放大圆
   circle.expand(50);
   circle.contract(65); // 缩小圆
   circle.contract(65);
   closegraph();  // 关闭图形系统
   return 0;
}
```

文件的组织

```
// BASGRAPH.H

class LOCATION {
public:  …
protected: …
};

class POINT: public LOCATION {
public: …
protected:  …
};

class CIRCLE: public POINT {
public:
protected:
};
```

```
// LOCATION.CPP
#include <BASGRAPH.H>
LOCATION各成员函数的实现
```

```
// POINT.CPP
#include <BASGRAPH.H>
POINT各成员函数的实现
```

```
// CIRCLE.CPP
#include <BASGRAPH.H>
CIRCLE各成员函数的实现
```

```
// GRAFDEMO.CPP
#include <BASGRAPH.H>
客户代码
```

文件的组织

```
//LOCATION.H
class LOCATION {
public:  …
protected: …
};
```

```
// LOCATION.CPP
#include < LOCATION.H >
LOCATION各成员函数的实现
```

```
#inlcude<LOCATION.H>
class POINT: public LOCATION {
public: …
protected:  …
};                          //POINT.H
```

```
// POINT.CPP
#include <POINT.H>
POINT各成员函数的实现
```

```
#include<POINT.H>
class CIRCLE: public POINT {
public:
protected:
};                         //CIRCLE.H
```

```
// CIRCLE.CPP
#include <CIRCLE.H>
CIRCLE各成员函数的实现
```

文件的组织

- 下面我们将再通过**TIME**和其派生类**EXTTIME**探讨多文件的组织问题。

class Time Specification: time.h

```
#ifndef TIME_H
#define TIME_H

class  Time
{
public :
     void  Set ( int  hours , int  minutes , int  seconds ) ;
     void  Increment ( ) ;
     void  Write ( )  const ;
     Time ( int  initHrs, int  initMins,  int  initSecs ) ;  // constructor
     Time ( ) ;        //  default constructor

private :
     int hrs ;
     int mins ;
     int   secs ;
} ;

#endif
```

Implementation： time.cpp

```
#include "time.h"
#include <iostream>

using namespace std;

Time::Time( int initHrs, int initMins, int initSecs )
{   :   }

Time::Time()
{   :   }
void Time::Set(int hours, int minutes, int seconds )
{  :   }
void Time::Increment()
{  :   }

void Time::Write() const
{  :   }
```

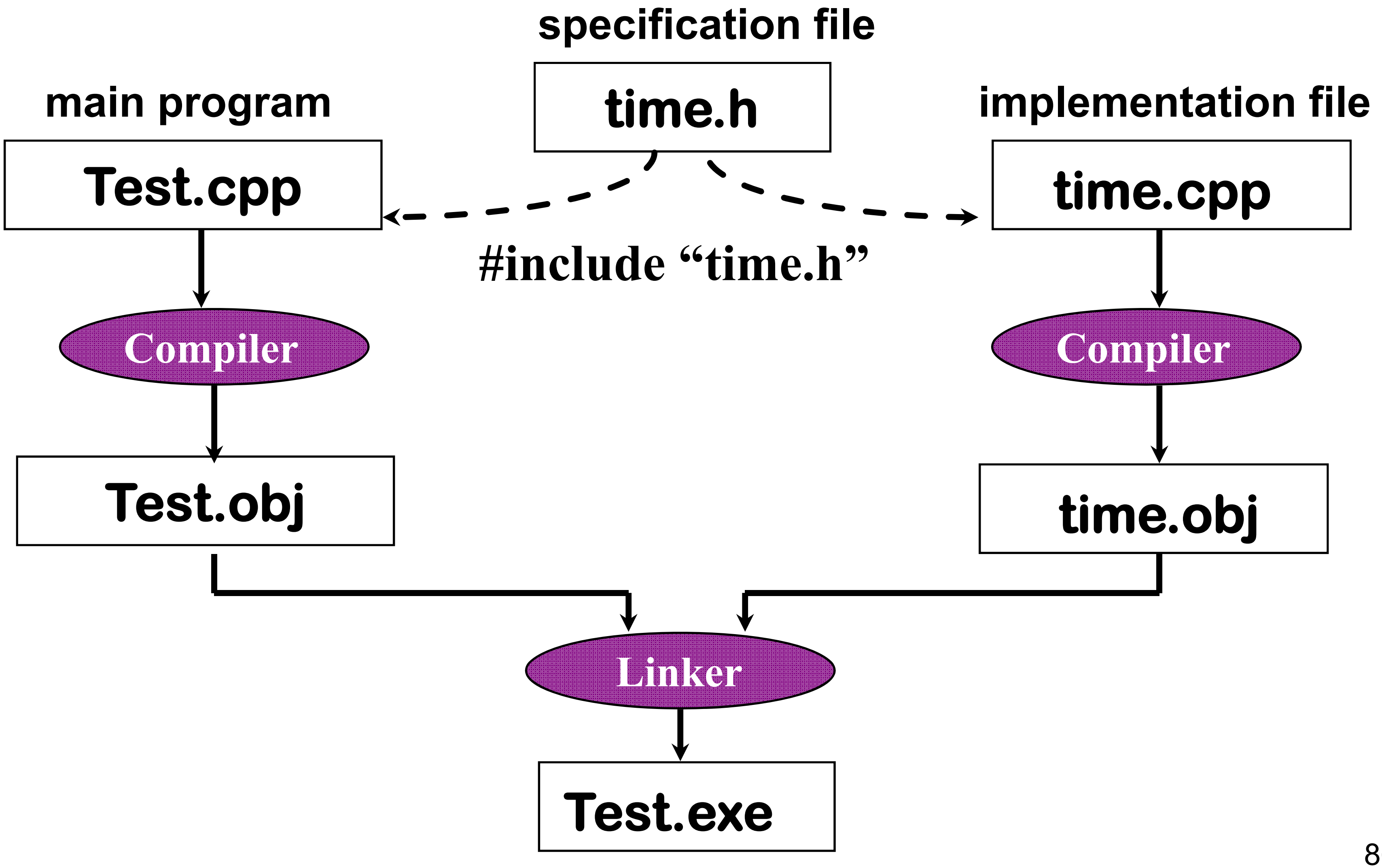
Client codes: Test.cpp

```
#include <iostream>
#include "time.h"

using namespace std;

int main()
{
   Time time(5, 30, 0 );

   time.Increment();
   time.Write();

   return 0;
}
```

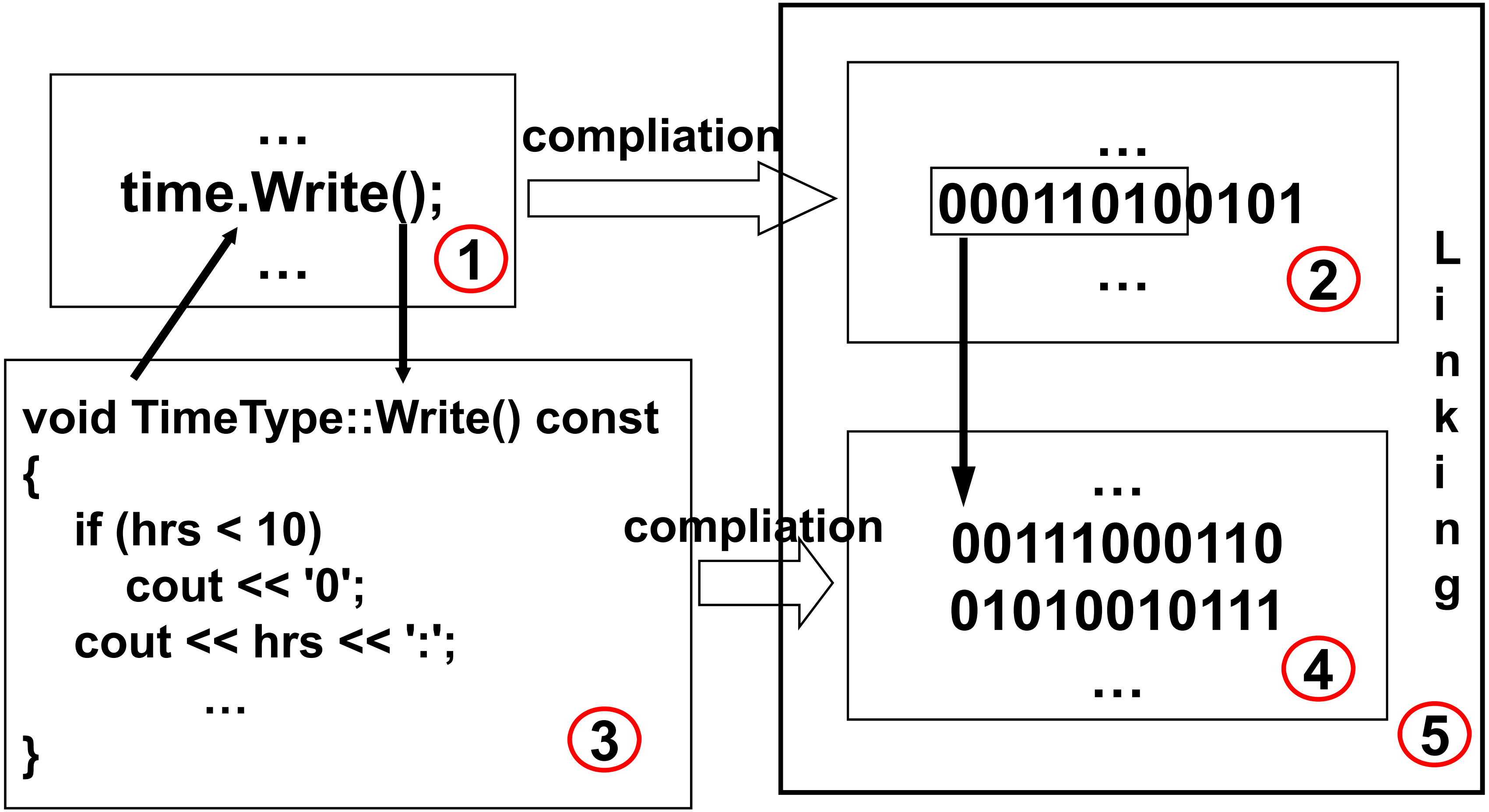
Separate Compilation and Linking of Files

- .cpp被编译成.obj文件，同一程序中的各个obj文件被链接成.exe可执行文件。
- 在C++中，多文件程序中的各.cpp文件不但被单独编译（separate compilation），而且可以在不同的时刻编译。
- 对于一个类，例如Time，其.h及.obj文件都应该可以被用户使用。用户把前者加入（include）到自己的程序中，使得可以编写利用Time类的代码；用户也需要后者链接到他自己的程序上，以便创建可执行文件。

Separate Compilation and Linking of Files



Separate Compilation and Linking of Files



1 test.cpp 2 test.obj 3 Time.cpp 4 Time.obj
5 Test.exe

SPECIFICATION FILE: ExtTime.h

```
#include   “time.h”
enum  ZoneType {EST, CST, MST, PST, EDT, CDT, MDT, PDT } ;

class  ExtTime  :  public  Time       // Time is the base class
{
public :
      ExtTime ( int  initHrs ,  int  initMins ,  int  initSecs ,
                    ZoneType    initZone ) ;      // constructor
      ExtTime ( ) ;                                // default constructor
      void Set ( int  hours,  int  minutes,  int   seconds ,
                    ZoneType   timeZone ) ;
      void Write ( )  const ;

private :
         ZoneType  zone ;      //  added data member
} ;
```

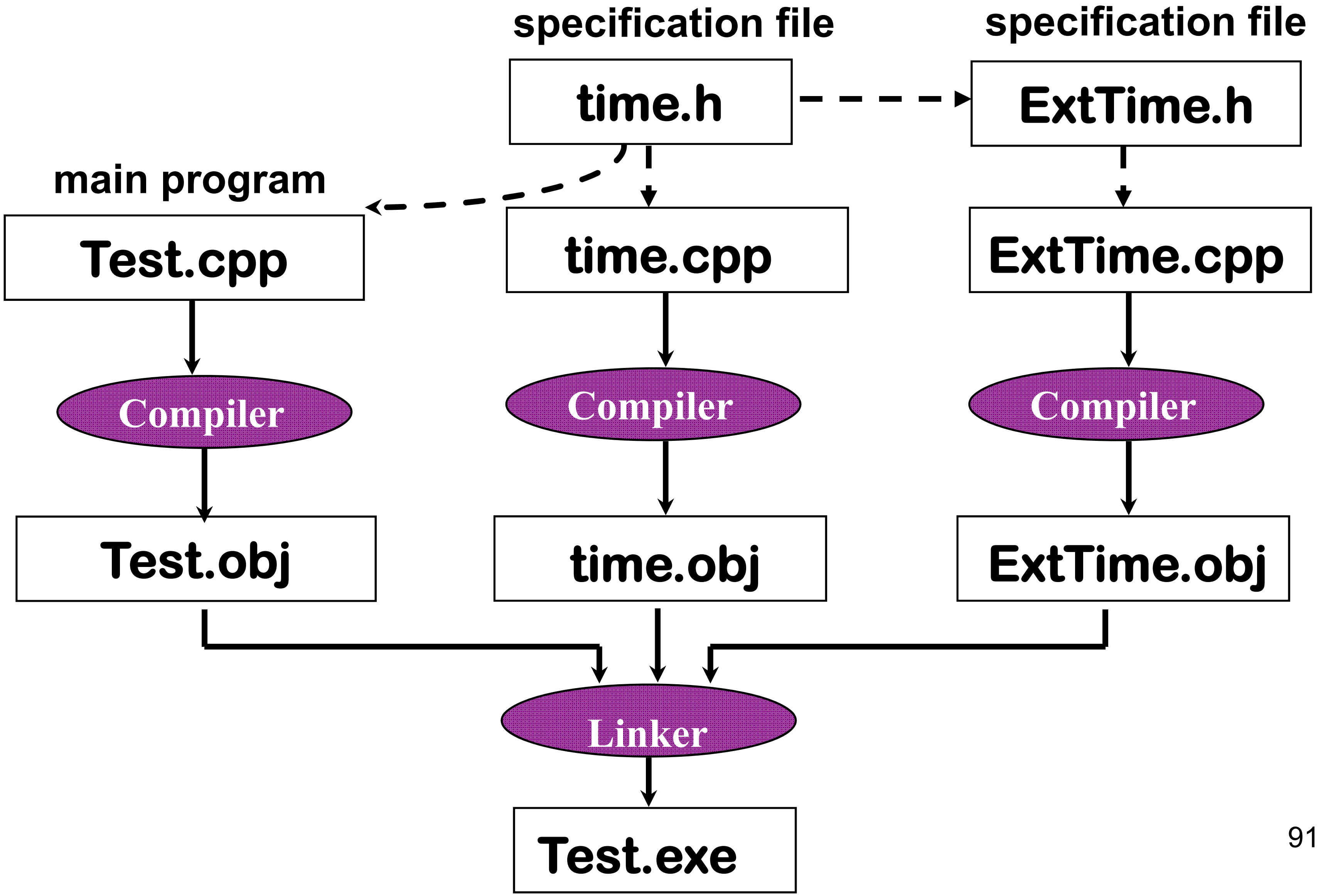
IMPLEMENTATION FILE: ExtTime.cpp

```
#include “ExtTime.h"

//各成员函数的实现
                              :
```

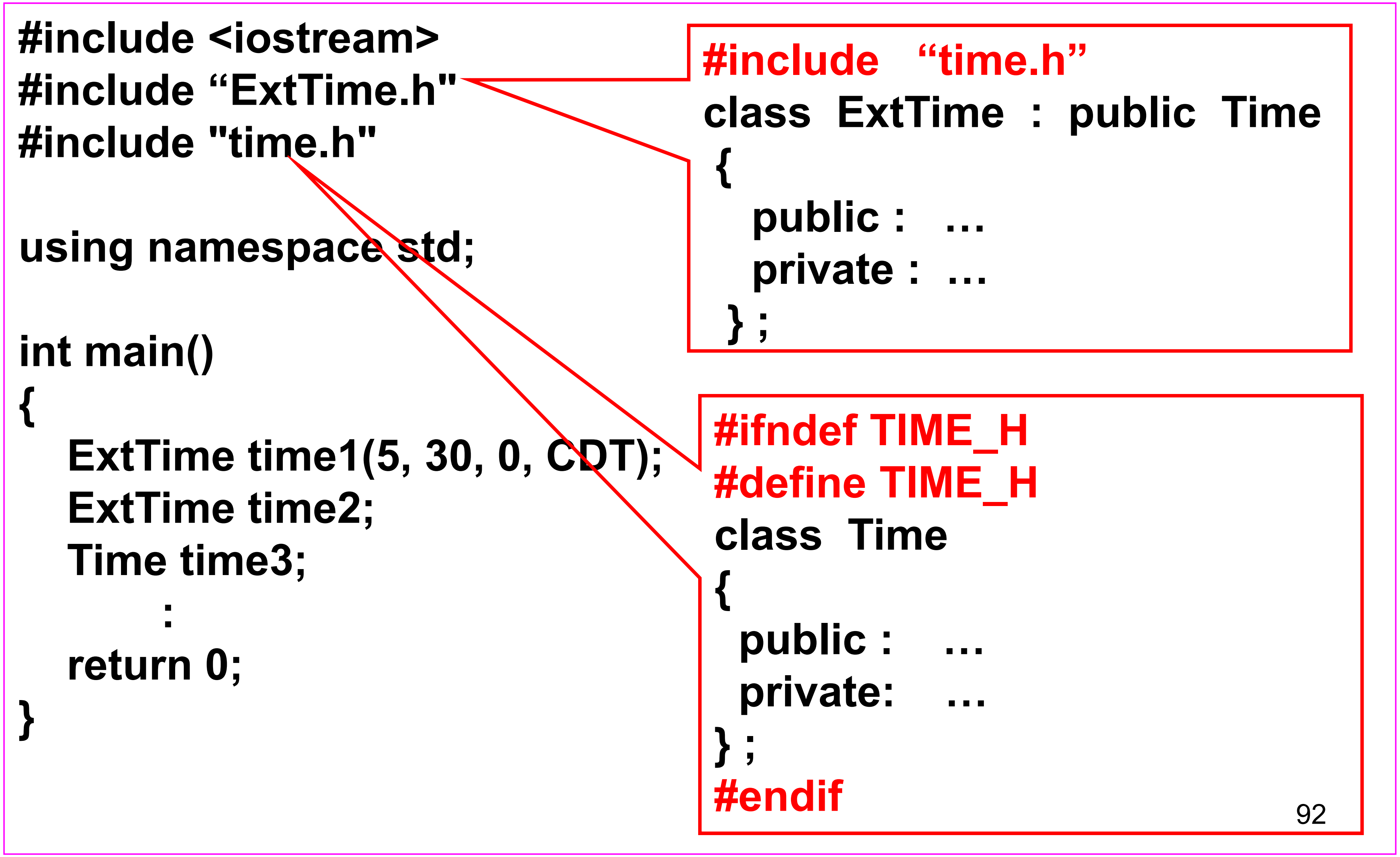
Client Code: Test.cpp

```
#include “ExtTime.h"
#include "time.h"

using namespace std;

int main()
{
   ExtTime time1(5, 30, 0, CDT);
   ExtTime time2;
   Time time3;

   time1.Set( 2, 2, 3, MDT);
   time3.Write();
         :
   return 0;
}
```

Separate Compilation and Linking of Files



Client Code: Test.cpp



```
#include <iostream>
#include “ExtTime.h"
#include "time.h"

using namespace std;

int main()
{
    ExtTime time1(5, 30, 0, CDT);
    ExtTime time2;
    Time time3;
            :
    return 0;
}
```

```
#include   “time.h”
class  ExtTime  :  public  Time
 {
    public :   …
    private :  …
  } ;
```

```
#ifndef TIME_H
#define TIME_H
class  Time
{
  public :    …
  private:    …
} ;
#endif
```

Avoiding Multiple Inclusion of Header Files

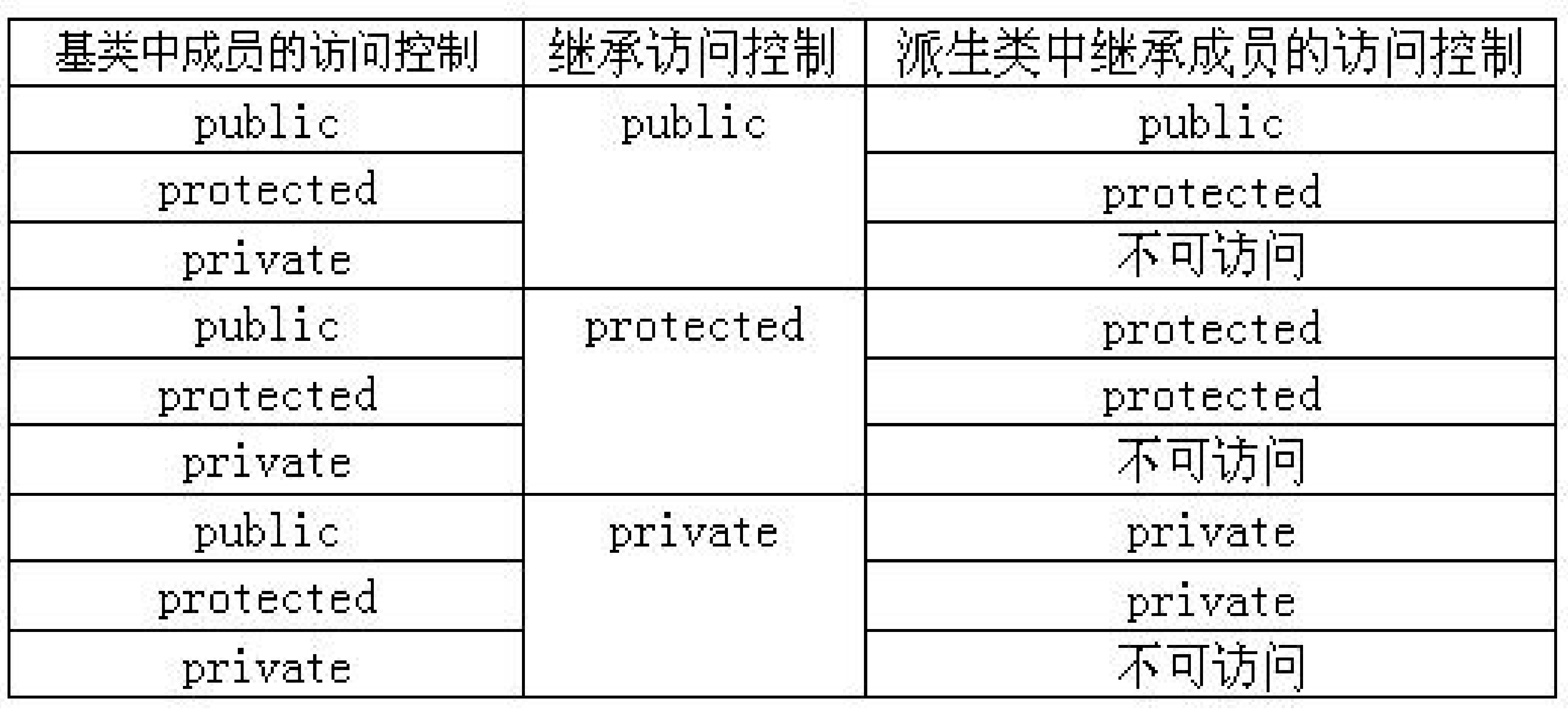
- often several program files use the same header file containing typedef statements, constants, or class type declarations--but, it is a compile-time error to define the same identifier twice.
- this preprocessor directive syntax is used to avoid the compilation error that would otherwise occur from multiple uses of #include for the same header file

```
#ifndef   Preprocessor_Identifier
#define  Preprocessor_Identifier
                        :
#endif
```

Adjustments of the inherited members

- **When the inherited members can not meet requirements of the derived class, they should be adjusted.**
- **Adjustments include**
  - **Resuming the access control**
  - **Redefinition of the inherited members.**
  - **Rename of the inherited members.**
  - **Hiding the inherited members.**

继承成员的访问控制规则



基类中成员的访问控制	继承访问控制	派生类中继承成员的访问控制
public	public	public
protected		protected
private		不可访问
public	protected	protected
protected		protected
private		不可访问
public	private	private
protected		private
private		不可访问

恢复访问控制方式

- 基类中的public或protected成员，因使用protected或private继承访问控制而导致在派生类中的访问方式发生改变，可以使用“访问声明”恢复为原来的访问控制方式
- 访问声明的形式

  基类名::成员名；（放于适当的成员访问控制后）
- 使用情景
  - 在派生类中希望大多数继承成员为protected或private，只有少数希望保持为基类原来的访问控制方式。

恢复访问控制方式[例]

```
class BASE {
public:
      void set_i(int x)
      {
          i = x;
      }
      int get_i()
      {
          return i;
       }
protected:
       int i;
};
```

```
class DERIVED: private BASE
{
public:
     BASE::set_i;   // 访问声明
     BASE::i;
     void set_j(int x)
     {
         j = x;
     }
      int get_ij()
      {
          return i + j;
      }
protected:
      int j;
};
```

恢复访问控制方式[例]

```
int main()
{
    DERIVED obj;       // 声明一个派生类的对象

    obj.set_i(5);         // set_i()已从私有的转为public
    obj.set_j(7);         // set_j()本来就是公有的
    cout << obj.get_ij() << "\n";       // get_ij()本来就是公有的

    return 0;
}
```

继承成员重定义

- 派生类中修改继承成员函数的语义（即，修改函数体，而保持函数原型不变）。
- 派生类中的名字支配（屏蔽）基类中的名字。

BASGRAPH.H

```
// 说明：类POINT描述了某一个位置是隐藏的还是显示的。
// 以public继承表示x_pos和y_pos在POINT中是protected
class POINT: public LOCATION {
public:
         :
        void show();      // 在当前位置显示点
        void hide();       // 将点隐藏起来
        // 将当前点移动到新位置(x, y)并显示它
        void move_to(int x, int y);
protected:
        :
};
```

BASGRAPH.H

```
// 说明：类CIRCLE描述了一个在屏幕上由POINT派生出来的圆。
// 由POINT类派生，从而也继承了LOCATION类
class CIRCLE: public POINT {
public:
                        :
          void show(); // 在屏幕上画出圆
          void hide(); // 将圆隐藏起来
          void move_to(int x, int y); // 将当前圆移动到新位置(x, y)
                        :
protected:
                        :
};
```

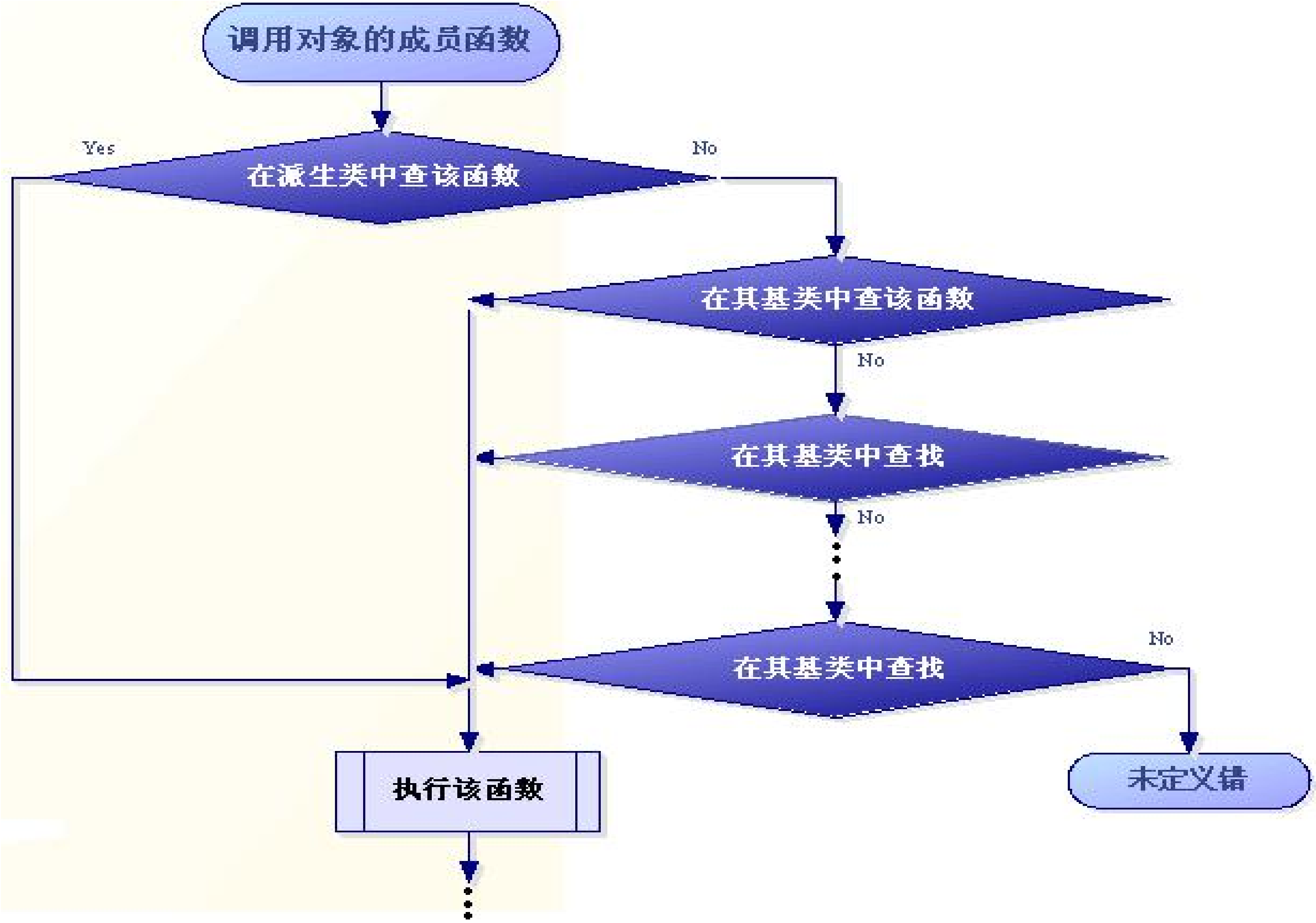
LOCATION.CPP

```
void POINT::show()
{
        if (! is_visible()) {
            visible = TRUE;
             putpixel(x_pos, y_pos, getcolor());
        }
}

void CIRCLE::show()
{
       if (! is_visible())
      {
             visible = TRUE;
            circle(x_pos, y_pos, radius);
      }
}
```

GRAFDEMO.CPP

```
#include "basgraph.h" // 基本图形元素的类界面
#include <conio.h>       // 利用其中的getch()函数暂停
int main()
{
                              :
  // 声明一个圆，圆心在(100, 200)，半径为50
  POINT point( 20, 10 );
  CIRCLE circle(100, 200, 50);

  circle.show();  // 显示圆
  point.show();  // 显示点

  circle.move_to(200, 250);  // 移动圆
  point.move_to(100,20);     // 移动点
                              :
}
```

编译器对成员函数调用的处理



重载继承成员

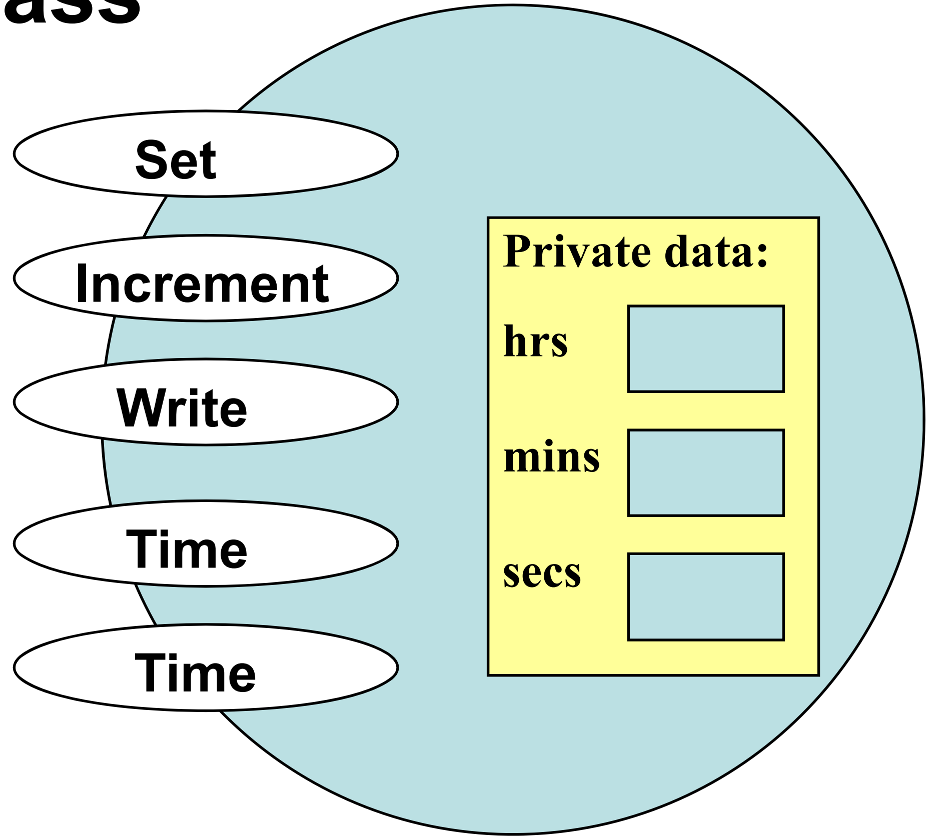
- 函数名相同，但函数首部不同（即参数列表不同；当然，函数实现一般也不同）。
- 利用重载，实现新的功能。

class Time Specification

```
//  SPECIFICATION FILE ( time.h )

class  Time
{
public :
      void  Set ( int  hours , int  minutes , int  seconds ) ;
      void  Increment ( ) ;
      void  Write ( )  const ;
      Time ( int  initHrs, int  initMins,  int  initSecs ) ;  // constructor
      Time ( ) ;     //  default constructor

private :
      int hrs ;
      int mins ;
      int secs ;
} ;
```

Class Interface Diagram

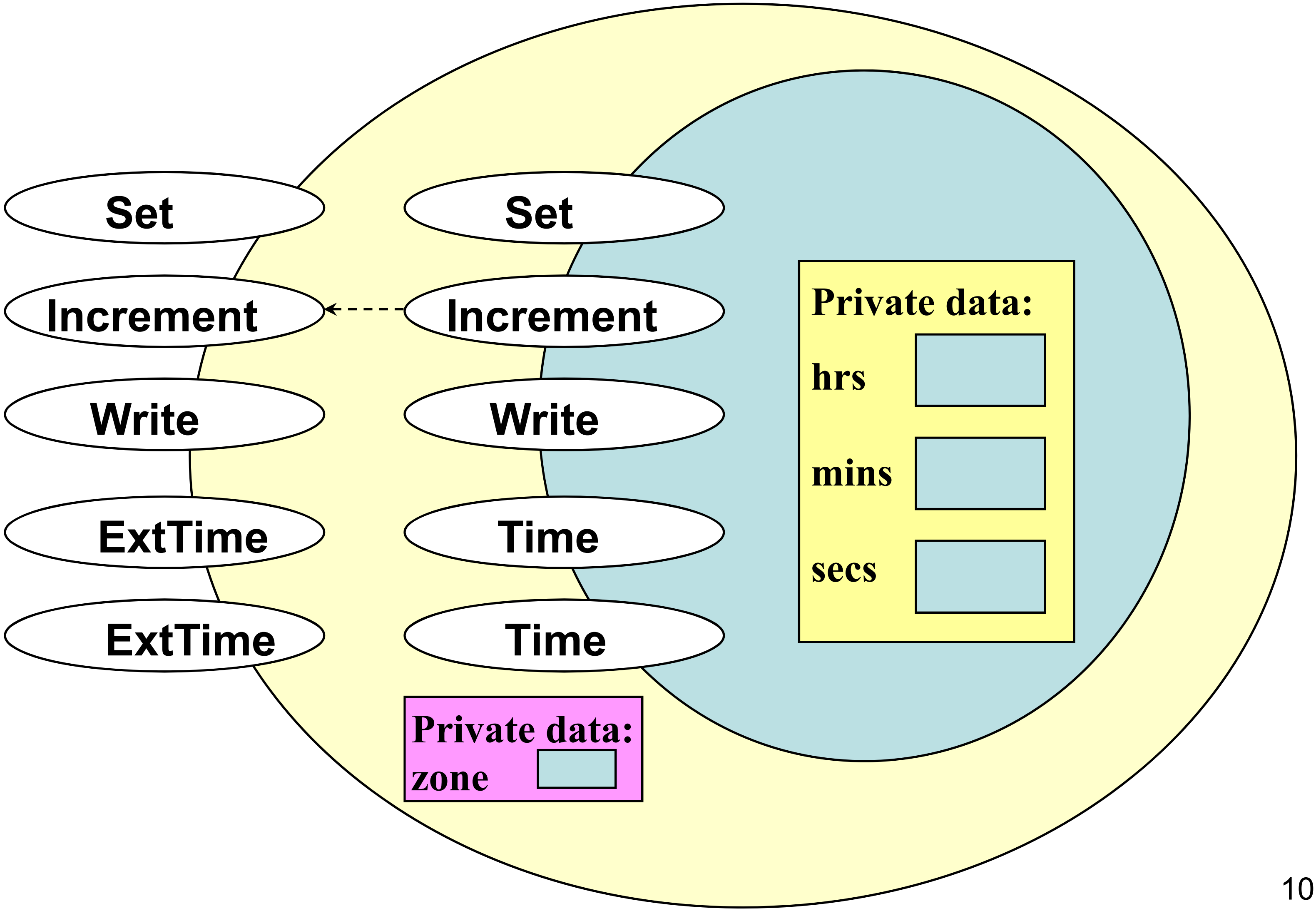
Time class



利用继承加入新特性

```
// SPECIFICATION   FILE                          ( exttime.h)
#include   “time.h”
enum  ZoneType {EST, CST, MST, PST, EDT, CDT, MDT, PDT } ;

class  ExtTime  :  public  Time       // Time is the base class
{
public :
        ExtTime ( int  initHrs ,  int  initMins ,  int  initSecs ,
                         ZoneType    initZone ) ;      // constructor
        ExtTime ( ) ;                                   // default constructor
        void Set ( int  hours,  int  minutes,  int   seconds ,
                         ZoneType   timeZone ) ;
        void Write ( )  const ;

private :
            ZoneType  zone ;      //  added data member
} ;
```

Interface Diagram of class ExtTime



函数重载：函数名相同，函数首部不相同

```
//  IMPLEMENTATION FILE ( time.cpp )
void Time::Set( int hours, int minutes, int seconds )

{
   hrs = hours;
   mins = minutes;
   secs = seconds;
}
```

```
//  IMPLEMENTATION FILE ( Exttime.cpp )
void ExtTime::Set( int hours, int minutes, int seconds,
                        ZoneType timeZone )
{
   Time::Set(hours, minutes, seconds);
   zone = timeZone;
}
```

函数重定义：函数首部相同，实现不同

```
// ( time.cpp )

void Time::Write() const
{
   if (hrs < 10)
      cout << '0';
   cout << hrs << ':';
   if (mins < 10)
      cout << '0';
   cout << mins << ':';
   if (secs < 10)
      cout << '0';
   cout << secs;
}
```

```
//  ( Exttime.cpp )

void ExtTime::Write() const
{
   static string zoneString[8] =
   {
      …
   };

   Time::Write();
   cout << ' ' << zoneString[zone];
}
```

屏蔽继承成员

- 目的：
  - 使得客户代码通过派生类对象不能访问继承成员。
- 方法：
  - 使用继承访问控制**protected**和**private**（真正屏蔽）
  - 在派生类中成员访问控制**protected**或**private**之后定义与继承成员函数相同的函数原型，而函数体为空（非真正屏蔽，仍可通过使用“基类名::成员名”访问）　　　　**//程序7.4.2**

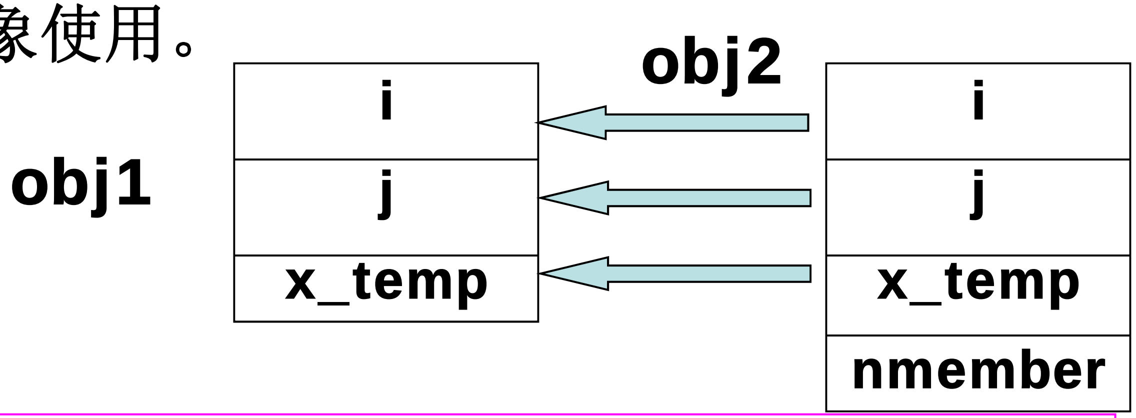
继承成员重命名

```
string str = “abc”;
cout << str.length();
cout << str.size();
```

- 目的：
  - 解决名字冲突。
  - 在派生类中选择更合适的术语命名继承成员。
- 方法
  - 在派生类中定义新的函数，该函数调用旧函数；屏蔽旧函数。
  - 在派生类中定义新的函数，该函数的函数体与旧函数相同。

类型兼容性

- 赋值运算的类型兼容性
  - 可以将后代类的对象赋值给祖先类对象，反之不可。
  - 每个派生类对象包含一个基类部分，这意味着可以将派生类对象当作基类对象使用。



```
BASE  obj1;
Y1      obj2;
obj1 = obj2 ;  //把obj2中基类部分的内容赋给obj1
obj2 = obj1 ;  //错误
```

赋值运算的类型兼容性（续）

```
Y1继承BASE，且
    BASE  obj1;
    Y1      obj2;
```

- 指向基类对象的指针也可指向公有派生类对象

```
BASE *p ;                Y1 *p1;
p = &obj1;✓              p1 = &obj1; ✗
p = &obj2;✓              p1 = &obj2;✓
p = p1 ; ✓
```

- 只有公有派生类才能兼容基类类型（上述规则只适用于公有派生）。

类型兼容性（续）

- 参数传递与对象初始化的类型兼容性
  - 与赋值运算的类型兼容性相同

例

```
class Base {            //B.h
public:
        void display() ;
};

class D1: public Base {
 public:
        void display() ;
};

class D2: public D1 {
public:
        void display() ;
};
```

例

```
//B.cpp

#include “B.h”

void Base::display()
{    cout<<"Base::display()"<<endl;  }

void D1:: display()
{    cout<<"B2::display()"<<endl;      }

void D2:: display()
{    cout<<"D2::display()"<<endl;      }
```

例

```
void fun(Base *ptr)
{       ptr->display();        }

int main()
{       Base b;        //声明B0类对象
        D1 d1;         //声明D1类对象
        D2 d2;         //声明D2类对象
        Base *p;       //声明Base类指针
        p=&b;          //Base类指针指向Base类对象
        fun(p);
        p=&d1;         //Base类指针指向D1类对象
        fun(p);
        p=&d2;         //Base类指针指向D2类对象
        fun(p);
}
```

[例]

运行结果：

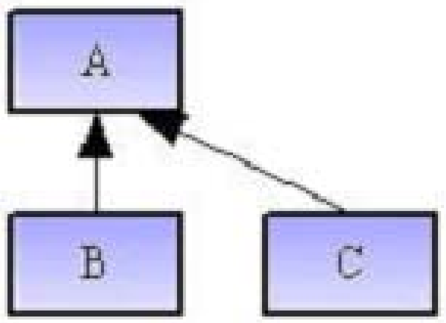
```
Base::display()
Base::display()
Base::display()
```

解释：

形参是指针类型，其基类型为Base

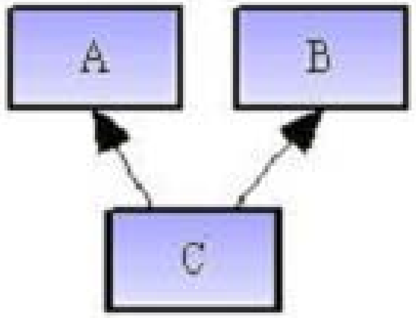
Inheritance supported by C++

Single Inheritance

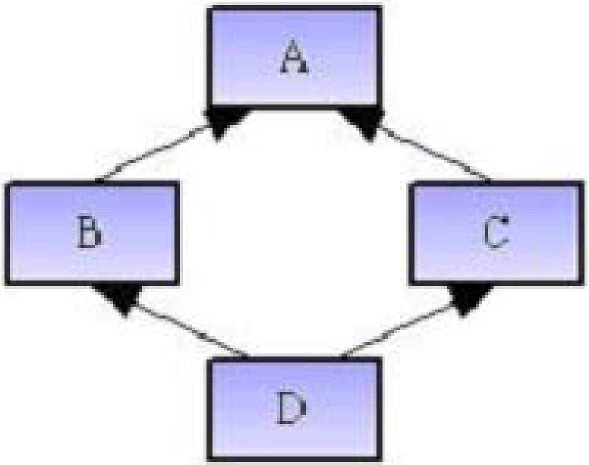


一个派生类只有一个直接基类

Multiple Inheritance

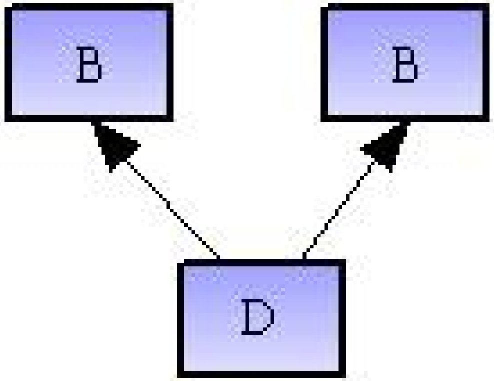


一个派生类具有两个或两个以上直接基类

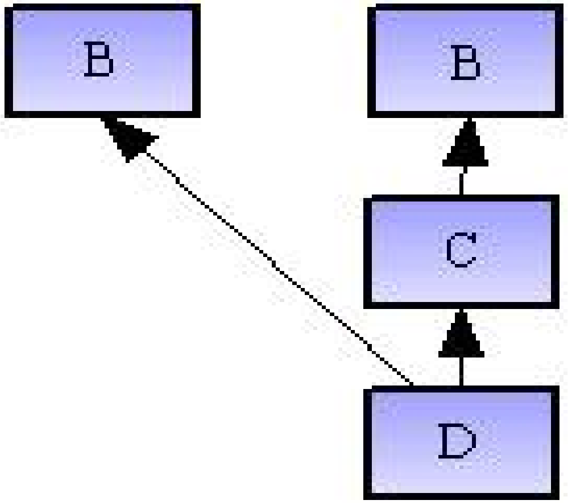


多重继承的一种特殊形式
派生类2次或2次以上重复继承某个祖先类

Inheritance NOT supported by C++



```
class D : public B , public B
  {......};
```



Multi-inheritance

- Multi-inheritance: a derived class has more than one base classes.
- Represents the concept: C is both A and B.
- Takes the form as follows.

```
class 派生类名：继承访问控制1  基类名1，
                继承访问控制2  基类名2，...
{
      成员声明；
}
```

Notice：Each inheritance access control applies to its tailing base class ONLY.

Example: device

```
class Device1 {
public:
      Device1()
     {
            volume = 5;
            powerOn = false;
      }
     Device1(int vol, bool onOrOff)
    {
            volume = vol;
            powerOn = onOrOff;
     }
                           :
```

Example: device

```
class Device1 {
public:
     void showPower()
    {
       cout << "The status of the power is :" ;
       switch (powerOn) {
              case true:
                     cout << "Power on. \n";
                     break;
              case false:
                     cout << "power off. \n";
                     break;
       }
    }
                         :
```

Example: device

```
class Device1 {
public:
        void showVol()
        {
                cout << "Volume is " << volume << endl;
        }

protected:
        int volume;             // 音量
        bool powerOn;           // 开关状态
};
```

Example: device

```
class Device2 {
public:
        Device2()
        {
          talkTime = 10;
          standbyTime = 300;
          power = 100;
        }

        Device2(int newTalkTime, int newStandbyTime, float powerCent)
        {

          talkTime = newTalkTime;
          standbyTime = newStandbyTime;
          power = powerCent;
        }
```

Example: device

```
class Device2 {
public:
    void showProperty()
   {
       cout << "The property of the device : "<< endl;
       cout << "talk time: " << talkTime << " hours" <<endl;
       cout << "standbyTime: " << standbyTime << " hours" <<endl;
    }

    void showPower ()
    {    cout <<" Power: " << power << endl;    }

protected:
          int  talkTime;                   //可通话时间（小时）
          int  standbyTime;                //可待机时间（小时）
          float power;                     //剩余电量百分比
};
```

Example: device

```
class DeviceNew: public Device1, public Device2 {
public:
        DeviceNew()
       {    weight = 0.56;  }

        DeviceNew(float newWeight, int vol, bool onOrOff, int newTalkTime,
                         int newStandbyTime, float powerCent) :
        Device2(newTalkTime, newStandbyTime, powerCent),
        Device1(vol, onOrOff)
       {   weight = newWeight;    }

        float getWeight()
       {  return weight;  }

private:
          float weight;          // 重量（克）
};
```

Example: device

```
int main()
{
    DeviceNew  device(0.7, 3, false, 10, 250, 80);        //声明派生类对象

    // getWeight()函数是DEVICE_NEW类自身定义的
    cout << "The weight of the device : " <<device.getWeight()<<endl;

    // showVol()函数是从DEVICE1类继承来的
    device.showVol();

    // showProperty()函数是从DEVICE2类继承来的
    device.showProperty();

     return 0;
}
```

Name class(名字冲突)

- **Name clash**：Ambiguity occurs when there are members with same name in the base classes of a derived class and when client codes attempt to access this name via the objects of the derived class, i.e. the compiler can not decide which version to use.
- solutions
  - Using domain resolution operator to explicitly tell which version to use.
  - Redefinition the clashing members in derived class.

Example 1

```
class BASE1 {
public:         void show() { cout << i << "\n"; }
protected:      int i;
};
class BASE2 {
public:         void show() {cout << j << "\n"; }
protected:      int j;
};
// 多重继承引起名字冲突：DERIVED的两个基类BASE1和
//BASE2有相同的名字show
class DERIVED: public BASE1, public BASE2 {
public:
        void set(int x, int y) {  i = x;  j = y; }
};// 派生类在编译时不出错：C++语法不禁止名字冲突。
```

Using domain resolution operator ::

```
int main()
{
        DERIVED obj;      // 声明一个派生类的对象
        obj.set(5, 7);    // set()是DERIVED类自身定义的

        // obj.show();
        // 二义性错误，编译程序无法决定调用哪一个版本

        obj.BASE1::show();
        // 正确，显式地调用从BASE1继承下来show()

        obj.BASE2::show();
        // 正确，显式地调用从BASE2继承下来show()
                    :
}                                              //程序NameClash
```

Using redefinition

```
class DERIVED: public BASE1, public BASE2 {
public:
        void set(int x, int y) {  i = x;   j = y;   }
        void show()
        {  cout << i << "\n";  cout << j << "\n";   }
};

int main()
{
    DERIVED obj;  // 声明一个派生类的对象
    obj.set(5, 7);    // set()是DERIVED类自身定义的
    obj.show();       // 无二义性问题，调用的是DERIVED中新定义的版本

    obj.BASE1::show();   // 仍然可调用从BASE1继承下来show()
    obj.BASE2::show();    // 仍然可调用从BASE2继承下来show()

    return 0;
}                                                    //程序NameClash1
```

Constructor and destructor in multiple inheritance

- Sequence of calling base class constructors: left to right as in the inheritance declaration.

```
class 派生类名：继承访问控制1  基类名1，
                继承访问控制2  基类名2，...
{
      成员声明；
}
```

Constructor and destructor in multiple inheritance

```
class BASE1
{
public:
        BASE1(int x)
       {
            cout << x << "->Constructing base1 object.\n";
        }
        ~BASE1()
        {
            cout << "Destructing base1 object.\n";
        }
};
```

Constructor and destructor in multiple inheritance

```
class BASE2
{
public:
          BASE2(int x)
          {
               cout << x << "->Constructing base2 object.\n";
          }
          ~BASE2()
          {
              cout << "Destructing base2 object.\n";
          }
};
```

Constructor and destructor in multiple inheritance

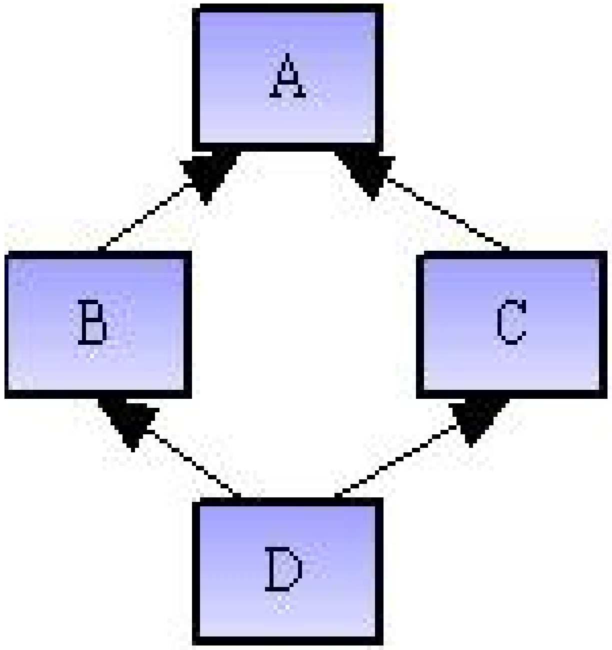
```
class DERIVED: public BASE2, public BASE1
{
public:
      DERIVED(int x, int y): BASE1(x), BASE2(y)
      {    cout << "Constructing derived object.\n";  }
      ~DERIVED()
      {    cout << "Destructing derived object.\n";  }
};
int main()
{
       DERIVED obj(10, 20); // 声明一个派生类的对象
       return 0;
}                                                  //程序07_05_04
```

Output

```
20->Constructing base2 object.
10->Constructing base1 object.
Constructing derived object.
Destructing derived object.
Destructing base1 object.
Destructing base2 object.
```

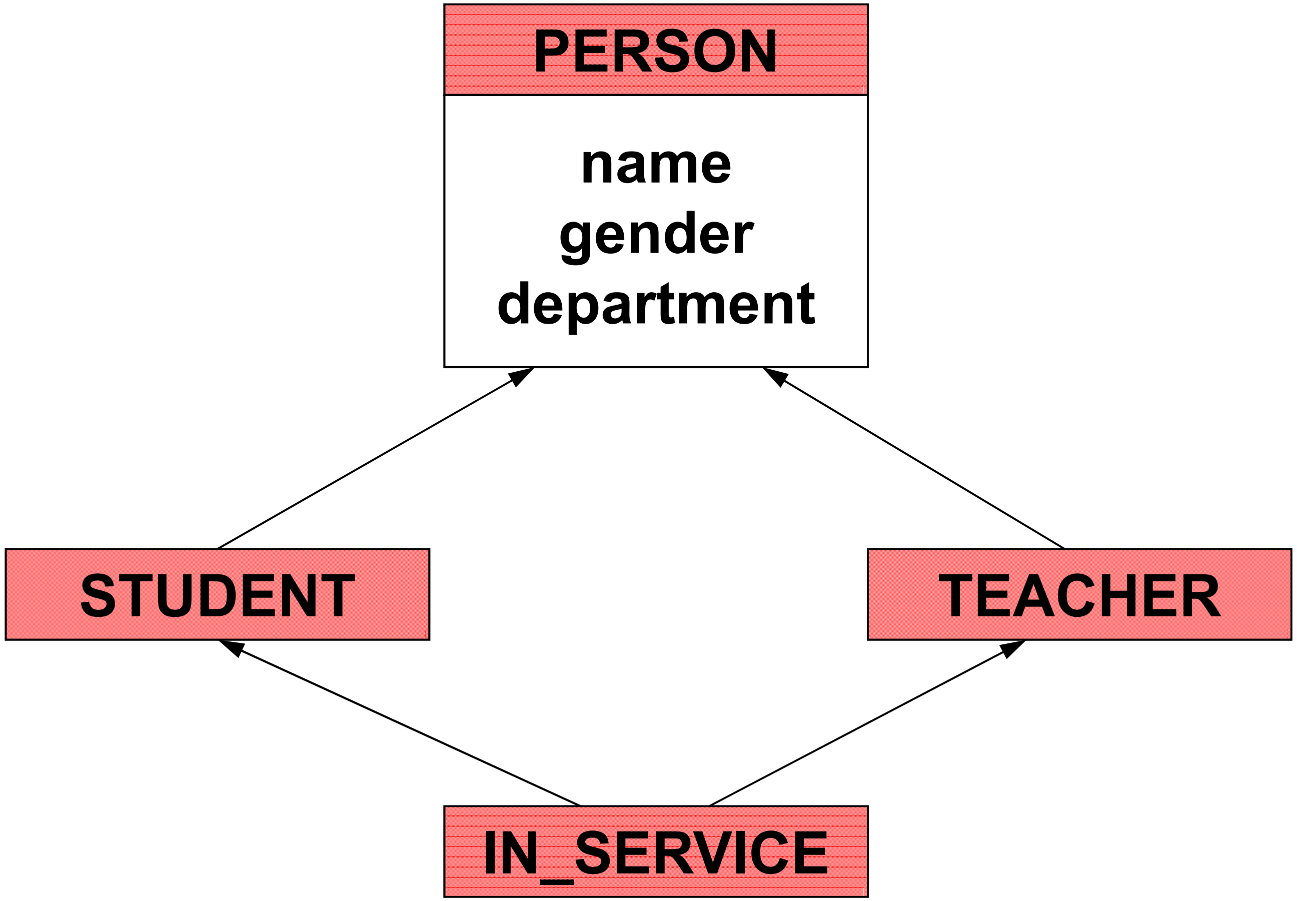
Base inherited twice or more

- **In a derived class, base members may be inherited twice or more in case of multiple inheritance.**



D通过B，C重复继承祖先类A

Example



PERSON

name
gender
department

STUDENT

TEACHER

IN_SERVICE

Two types of double inheritance

- **Duplicate inheritance(复制继承)：There are several copies of the base class members in a derived class object.**
- **Shared inheritance(共享继承)：There is only one copy of the base class members in a derived class object.**

Ambiguity

- In C++， duplicate inheritance is the default case. Ambiguity may occur.
- In C++, duplicate inheritance or shared inheritance is applied to the entire base class, NOT to just some particular members of the base class.

Example of ambiguity

```
class BASE {public:    int i;};
class BASE1: public BASE {
public: int j;
};
class BASE2: public BASE {
public: int k;
};
class DERIVED: public BASE1, public BASE2 {
public: int sum;
};
void main()
{
     DERIVED obj;        // 声明一个派生类对象
     obj.i = 3;      //错误，编译程序无法确定使用i的哪一份副本
     obj.j = 5;      //正确，使用从BASE1继承下来的j
     obj.k = 7;      //正确，使用从BASE2继承下来的k
}                                                      //程序7.6.1
```

obj	BASE1. BASE.i
	BASE1. j
	BASE2. BASE.i
	BASE2. k
	sum
	函数指针

solutions

- Using :: explicitly

```
int main()
{
    DERIVED obj;
    obj.BASE1::i = 3;  //正确
    obj.BASE2::i = 4;  //正确
      ……
}
```

- Shared inheritance：Using virtual base to guarantee ONLY one base class copy in derived class object.

Virtual base(虚基类)

- **Reserved word ‘virtual’ is added before the inheritance access control when the base class is inherited. Then this base class is a virtual base class.**
- **Virtual base class is used for shared inheritance.**
- 普通基类与虚基类之间的唯一区别只有在派生类重复继承了某一基类时才表现出来。

Example

```
class BASE {public:  int i;};
class BASE1: virtual public BASE {
            public: int j;
};
class BASE2: virtual public BASE {
            public: int k;
};
class DERIVED: public BASE1, public BASE2 {
            public: int sum;
};
int main()
{
      DERIVED obj;    // 声明一个派生类对象
      obj.i = 3;          // 正确：从BASE继承的i在DERIVED中只有一份
      obj.j = 5;          // 正确：使用从BASE1继承的j
      obj.k = 7;          // 正确：使用从BASE2继承的k
      return 0;
}
```

obj

BASE.i
BASE1. j
BASE2. k
sum
函数指针

//程序7.6.2

虚基类的构造函数与析构函数

- 若派生类有一个虚基类作为祖先类，则在派生类构造函数中需要列出对虚基类构造函数的调用（否则，调用虚基类的默认构造函数），且对虚基类构造函数的调用总是先于普通基类的构造函数。
- 创建后代类对象时，只有该后代类列出的虚基类构造函数被调用，这样就保证了虚基类的唯一副本只被初始化一次。
- 创建派生类对象时构造函数的调用次序：
  - 最先调用虚基类的构造函数；
  - 其次调用普通基类的构造函数，多个基类则按派生类声明时列出的次序、从左到右调用，而不是初始化列表中的次序；
  - 再次调用对象成员的构造函数，按类声明中对象成员出现的次序调用，而不是初始化列表中的次序
  - 最后执行派生类的构造函数。

例：虚基类的构造函数

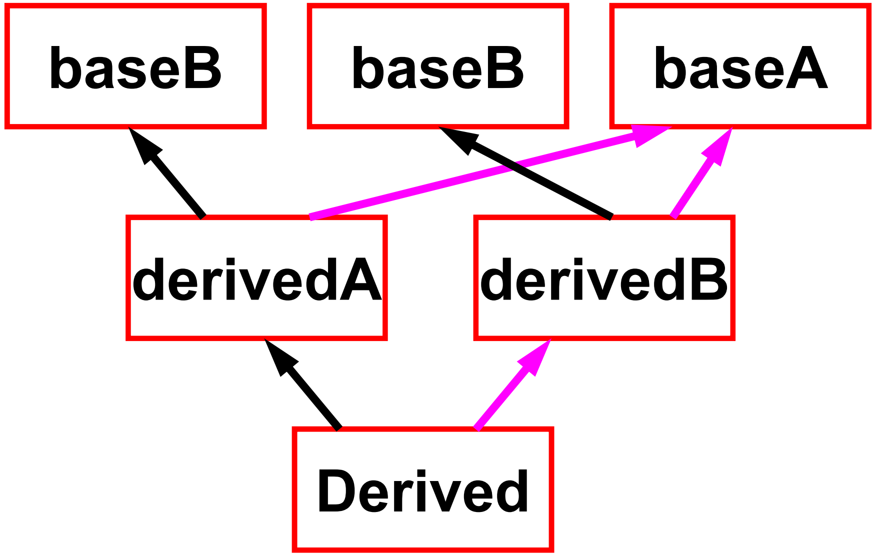
```
class baseA
{
public:
        baseA()
        {
                cout << endl << "This is baseA class." << endl;
        }
};

class baseB
{
public:
        baseB()
        {
                cout << endl << "This is baseB class." << endl;
        }
};
```

例：虚基类的构造函数

```
class derivedA : public baseB, virtual public baseA
{
public:
          derivedA()
          {
                    cout << endl << "This is derivedA class." << endl;
          }
};
class derivedB : public baseB, virtual public baseA
{
public:
          derivedB()
          {
                    cout << endl << "This is derivedB class." << endl;
          }
};
```

例：虚基类的构造函数

```
class Derived : public derivedA, virtual public derivedB
{
public:
        Derived()
        {
                cout << endl << "This is Derived class." << endl;
        }
};

int main()
{
        Derived obj;
        return 0;
}

//程序7.6.3
```



例：虚基类的构造函数

运行结果：

```
This is baseA class.

This is baseB class.

This is derivedB class.

This is baseB class.

This is derivedA class.

This is Derived class.
```