C++面向对象程序设计 主编:李涛 高教出版社

高级语言程序设计II

四川大学计算机学院

四川大学计算机学院

主讲教师:

个人主页:

http://cs.scu.edu.cn/~chenliangyin

高级语言程序设计II 四川大学精品课程

教材: 《C++:面向对象程序设计》

李涛 主编

游洪跃 陈良银 李琳等编

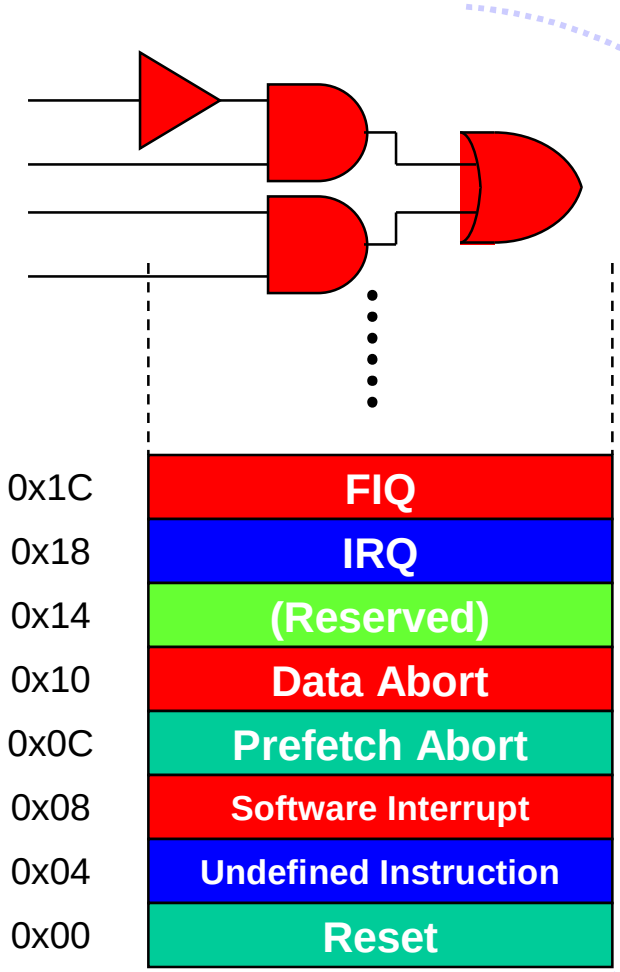
高等教育出版社

2006年2月出版

本书内容

第1章	绪论
第2章	C++类和对象
第3章	继承
第4章	多态性
第5章	模板
第6章	C++常见问题
第7章	Visual C++编程基础
第8章	对话框、常用消息、菜单和工具条
第9章	单文档界面和多文档界面
第10章	图形设备接口
实验	（待安排）

第二章 类与对象



ARM Vector Table

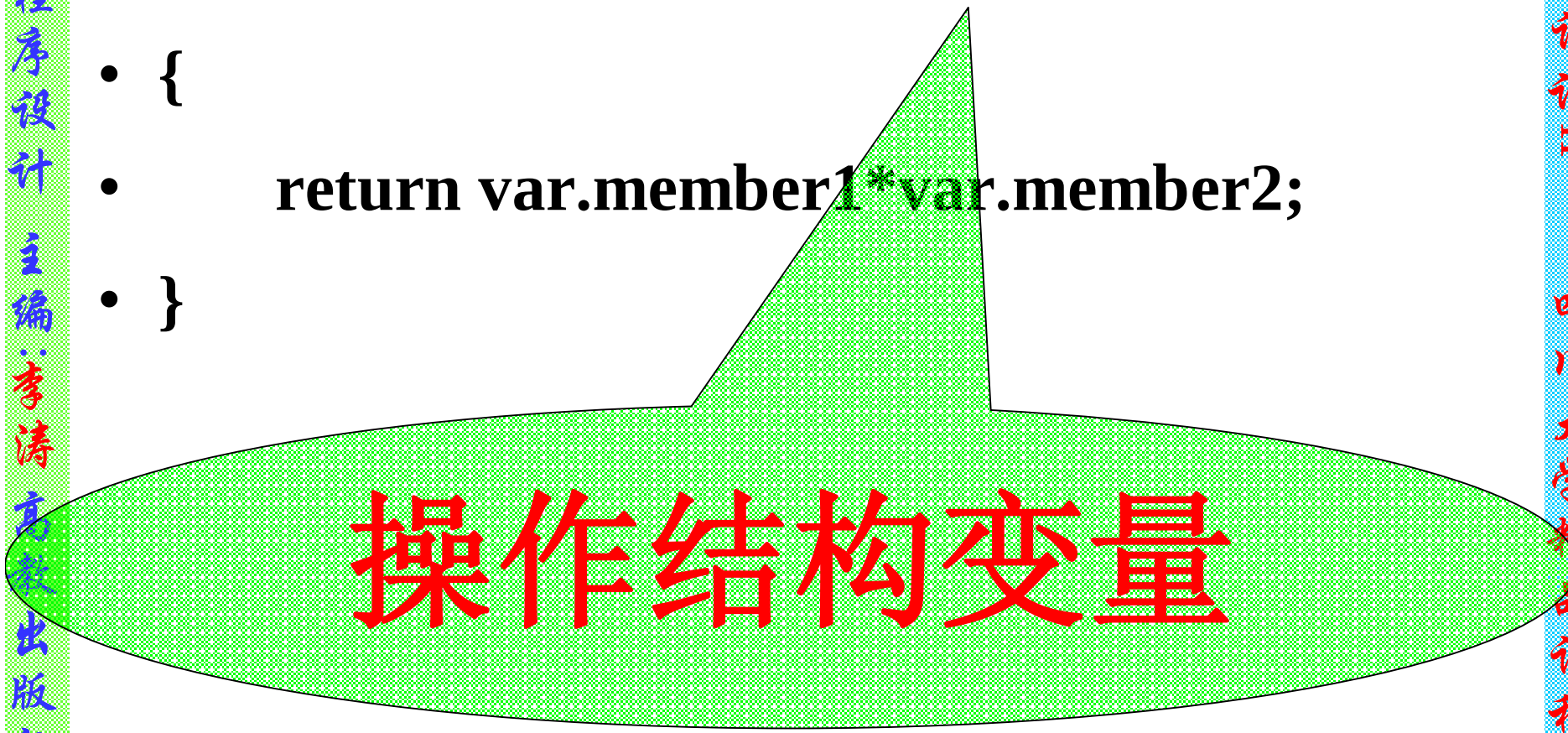
1. 类的定义
2. 类与对象
3. 输入与输出流

类的诞生（一）

- 类（Class）是面向对象思想中的一个重要组成部分。如何在计算机中将“类”表达出来？
- 在C语言中，当定义结构体（struct）：
- struct SAMPLE
- {
- int member1, member2;
- } var1;
- 以后，就可以使用结构中的成员：
- int var = var1.member1*var1.member2;

类的诞生（二）

- 将结构体成员的使用代码写成一个函数，得到：
- int caluc(struct SAMPLE var)
- {
- return var.member1*var.member2;
- }



- 然后，将上面的函数入到结构体中
- struct SAMPLE
- {
- int calcu()
- {
- return member1*member2;
- }
- int member1, member2;
- };

函数成员

数据成员

类的诞生（三）

- 这样的结构体就具有了一定的属性（member1和member2），也具有一定的行为（函数calcu）， 它就是“类”的雏形。

- 该结构体的使用方法如下：
- struct SAMPLE var;
- var.member1 = var.member2 = 10;
- int temp = var.calcu();

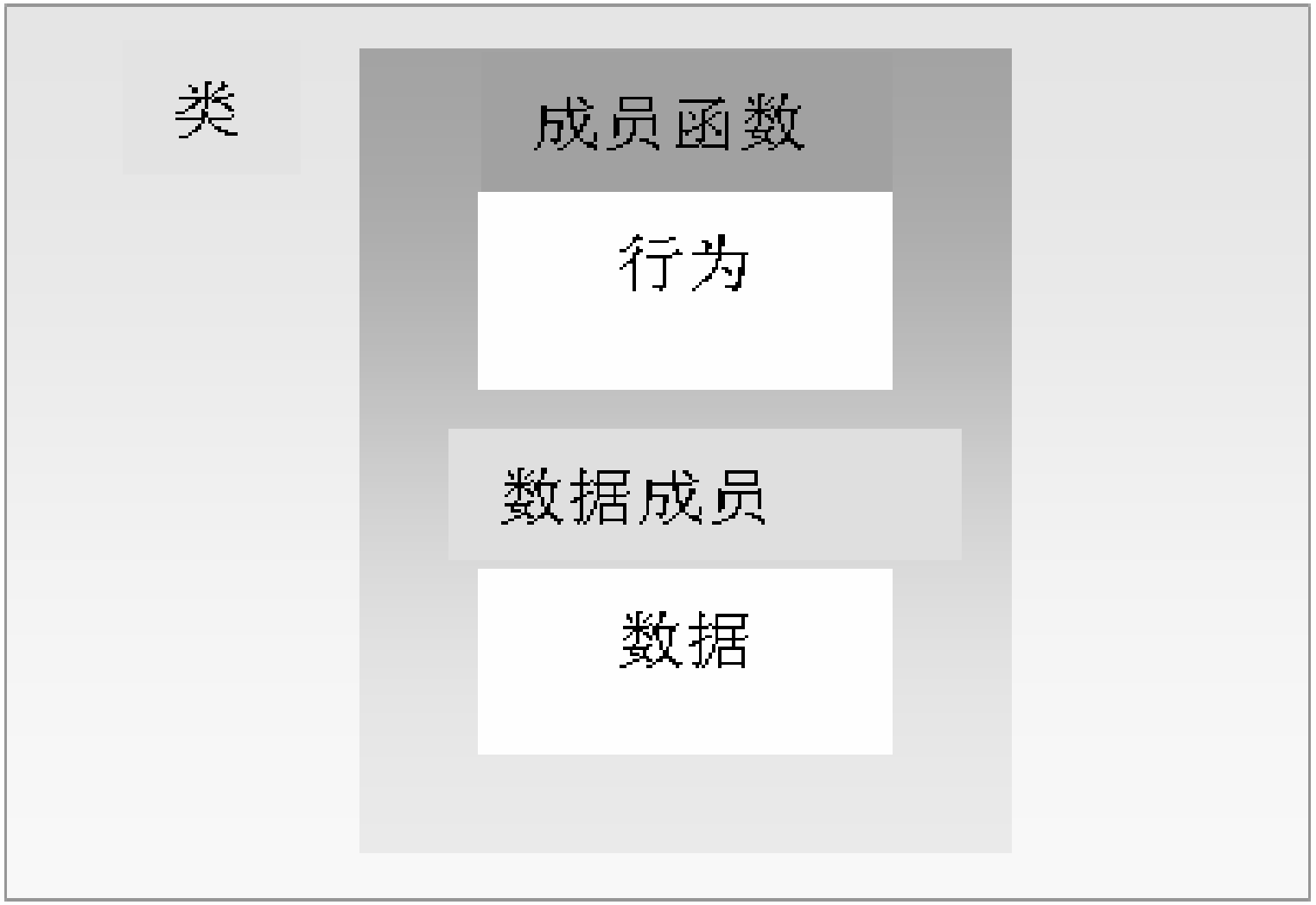


图 2.1 类的结构

C++类的声明与定义（一）

- class CCompanyStaff
- { //BEGIN

```
//声明成员函数
void SetBasicSal ( float sal ); // 设置基本工资

//声明（定义）数据成员
int m_iStaffNum ; // 工作编号
char m_cName[20]; // 姓名
float m_fRateOfAttend ; // 出勤率
float m_fBasicSal ;// 基本工资
```

- }; // END

struct à class

- 多了一个函数（SetBasicSal），其它都象结构体。

- **CCompanyStaff类的定义以关键字class开始。**

其后是类名。

- “{}”表示类定义的开始和结束，最终以分号结束。
- 一般在类中只声明成员函数的原型，而函数的实现（即函数体的定义）则放在类外完成。
- “函数原型”，即只声明函数名、参数类型和返回值类型，而不包括函数体代码。

C++类的声明与定义（二）

- 函数的定义则可在函数（SetBasicSal）声明之后：
- void CCompanyStaff::SetBasicSal(float sal)
- {
- m_fBasicSal = sal ;
- }
- 函数定义：函数名前多了一个类作用域运算符（CCompanyStaff::），用于标识该函数定义属于哪一个类。

C++类的声明与定义（三）

- 对于C++语言的编程习惯：
- 1、将类的声明存放于“*.h”或“*.hpp”的头文件中，每个类一个声明文件。
- 2、将类的定义存放于“*.cpp”文件中，与相应的声明文件一一对应

.c à .cpp

- 封装成类的好处——可以实现数据隐藏。
- 封装也确定了类成员的访问属性。

对象的封装性

- C++语言引入面向对象思想，引入类（对象）的概念之后，必然会具有一些新的特性。
- 面向对象思想中类（对象）的基本特性之一：封装性。
- 封装性：直观理解，类将属于它的数据（成员变量）和针对数据的操作（成员函数）包裹在一起就是一种封装。
- 封装的目的：保护类（对象）的实现。
- 对封装属性进行细分：公有的；私有的；保护的。

C++的封装（一）

- C++语言为了实现面向对象的封装，引入了三个新的“修饰”关键字：
- **public**（公有的）：公开的，可见的。对象成员（变量与方法）可以在对象外使用。
- **private**（私有的）：不可见的。成员只能在对象内部使用。
- **protected**（保护的）：受到保护的。成员也只能在内部使用。（以后再讨论）
- C++类的成员（变量和方法）具有了各自不同的属性。

C++的封装（二）

- class CCompanyStaff
- {
  - public:
  - void SetBasicSal (float sal); // 设置基本工资
  - private:
  - int m_iStaffNum ; // 工作编号
  - char m_cName[20] ; // 姓名
  - float m_fRateOfAttend ; // 出勤率
  - float m_fBasicSal ; // 基本工资
- }; // END

C++的封装（三）

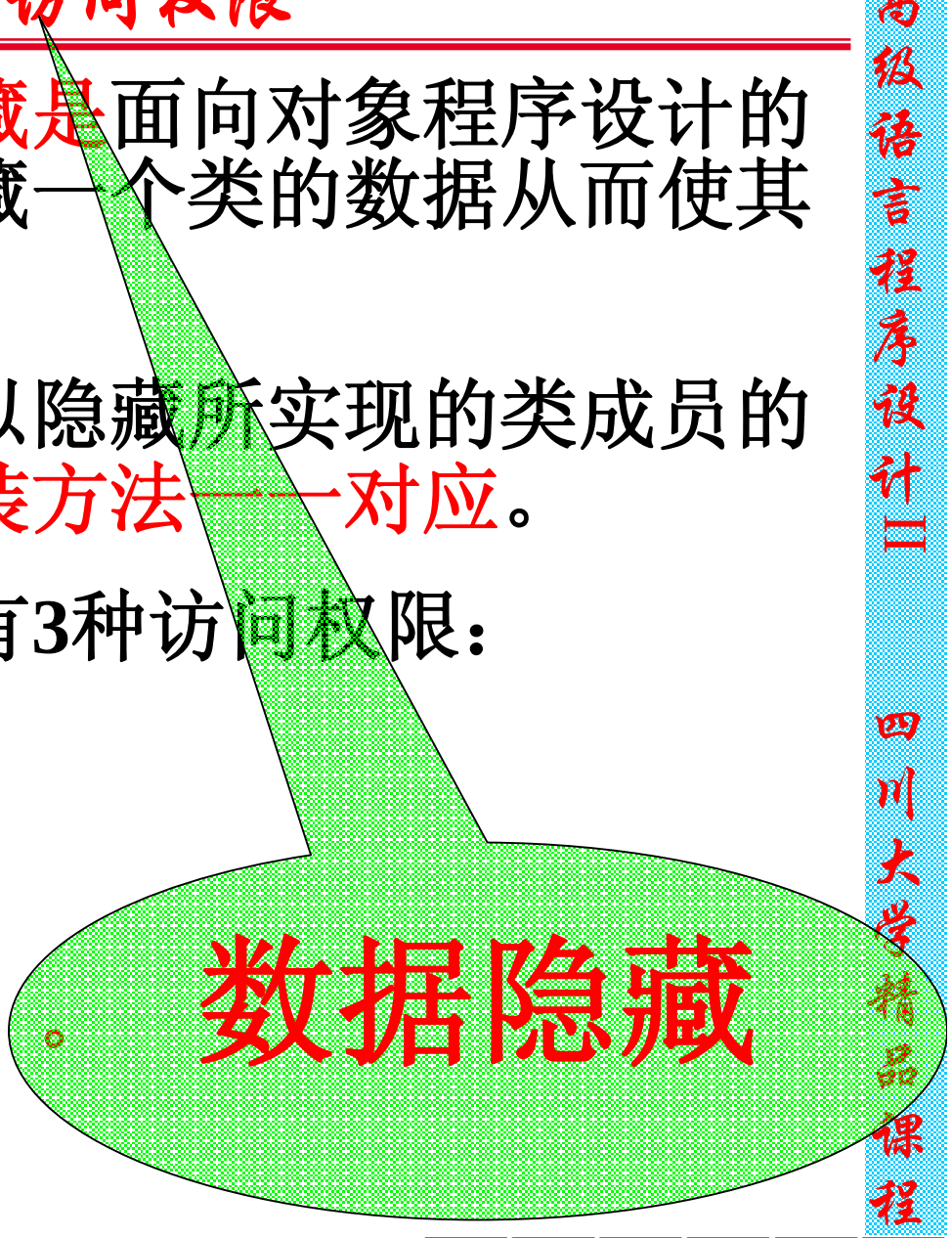
- CCompanyStaff　staff;
- staff.SetBasicSal(600); //合法的，因为SetBasicSal是一个公开（public）的方法。
- staff.m_iStaffNum = 100; //非法的，因为m_iStaffNum是一个私有（private）的变量。
- void CCompanyStaff::SetBasicSal(float sal)
- {
- 　　m_fBasicSal = sal; //SetBasicSal是对象的成员，所以它的定义中可以访问m_fBasicSal私有变量。
- }

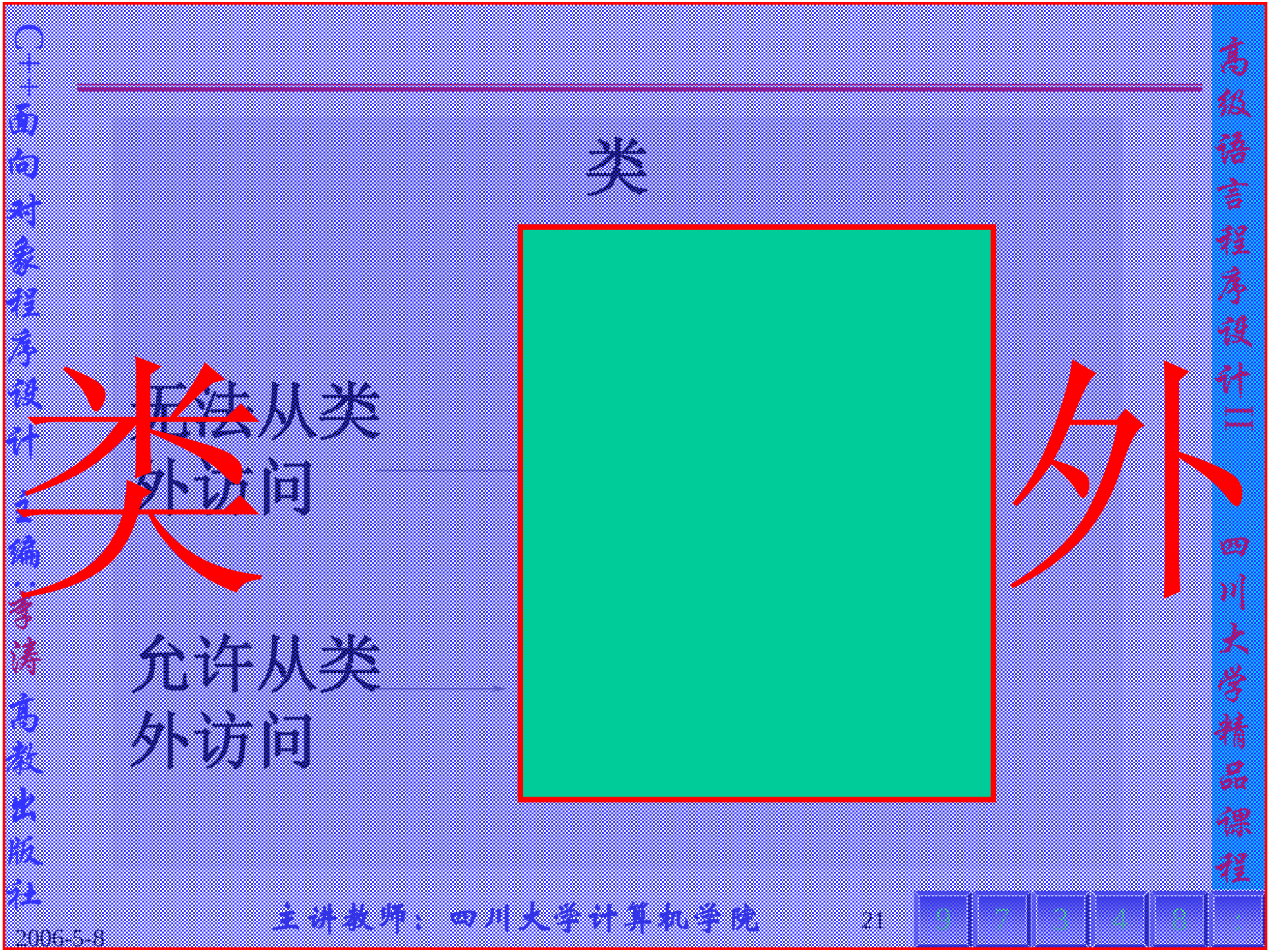
C++的封装（四）

- 1、C++类定义中的缺省属性为私有的（private）。
- 2、每个修饰符的作用范：从一个修饰符开始，直到另外一个修饰符时结束（或类结束了）。
- 3、C++类中的成员（变量和方法）都应该明确地指明它们各自的属性。

类成员的访问权限

- 封装所实现的数据隐藏是面向对象程序设计的一个关键特性——隐藏一个类的数据从而使其他类无法访问。
- 隐藏由封装实现，所以隐藏所实现的类成员的访问控制权限也和封装方法一一对应。
- 对应于封装，类成员有3种访问权限：
- 公有类型（public）、
- 私有类型（private）
- 保护类型（protected）。





- 公有类型的成员定义了类的接口，由关键字public声明，在类外只能访问公有成员。
- 私有类型的访问权限为私有的成员由关键字private声明，它们只能被类本身的成员函数访问，来自类外部的任何访问都是非法的。
- 保护类型的成员与私有成员类似，区别仅在于继承过程中，保护类型的成员可以被所在类的派生类成员函数访问；而这一点对于私有成员来说是非法的。

- 类的成员函数可以访问类的所有成员，没有任何限制。
- 而类对象访问类的成员就要受到访问控制符的限制。

类对象和类的成员函数对数据成员的访问权限不同。

- 访问权限举例：
- sclass2_1_demo.h
- smain2_1.cpp

- 外部接口是类外访问类中私有数据的桥梁。
- 声明时，类中不同访问权限的成员可以按任意顺序出现。但依然涉及到一个编程习惯问题：
- “以数据为中心”
- “以行为为中心”

国际公约。

- 建议把一个类的数据成员都声明为私有的访问权限。
- 这样做有两个好处：
- 一是信息隐藏，即实现封装，把类的内部实现和外部表现分开，让使用者无需了解类的实现细节；
- 二是数据保护，将类的重要信息保护起来，以免被其他程序不恰当地修改。

成员函数的实现

- 被隐藏的数据可以有成员函数来访问，对数据的操作也体现在成员函数中：
- 成员函数决定对象的操作行为。
- 它是程序算法的实现部分。
- 它也是对封装的数据进行操作的唯一途径。
- 它有两种方式：类外实现和类内实现。

数据对成员函数无法隐藏。

1. 一般实现方式

- 类外实现成员函数的形式如下：
- 返回值类型 类名::成员函数名（形式参数表）
- {
- 函数体
- }

例2.2 企业员工类的实现。

- sclass2_2_companyStaff.h
- s2_2\sclass2_2_companyStaff.cpp
- s2_2\smain2_2.cpp

2. 内联函数方式

- 内联函数是指程序在编译时用函数的代码替换每一处函数调用的地方 。
- 内联的优点——以空间换时间。
- 内联的两种方式：
- 系统默认：sclass2_2_0_companyStaff.h
- 和函数前加关键字**inline**：
- **inline** <函数返回值类型> <函数名>（<形参表>）{ <函数体语句> }
- smain2_3.cpp 。

内联和宏替换

- 宏替换是在编译前由预处理程序进行预处理，它只做简单的字符替换而不涉及语法检查。
- 而内联函数是在编译时处理的，编译程序能识别内联函数，对它进行语法检查。

类与对象

- 类（class）：具有相同或相近的性质和行为的事物集合。
- 对象（object）：**the instance of a class**。一个类的实例，即类的性质（变量）具体化之后成为 对象。
- 一个对象是类的一种特殊情况。
- 一个类会有很多的对象，
- 一个对象属于一个类。
- 它们是一对多的关系。

C++中的对象（一）

- 当C++中类的成员变量被赋与特定的值之后，它即成为一个C++的对象。
- **CCompanyStaff　　staff;　//一个对象变量**
- **staff.m_iStafNum = 12345;**
- **staff.m_cName = “张三” ;**
- **staff成为类CCompanyStaff的一个对象，它有了一个工作编号，对应一个人名。**
- 当然，还会有很多其它的对象，比如**staff1、staff2等等**

C++中的对象（二）

- 1、对象针对计算机而言，就是代码中的一个变量。
- 2、C++中对象变量的使用与C语言中结构体变量的使用方法一样。
- CCompanyStaff staff1, *pstaff1, staff3[10];
- staff1.m_iStafNum = 11111;
- pstaff à m_iStafNum = 22222;
- staff[0].m_iStafNum = 33333;

C++类的对象化（一）

- C++中对象是类的变量，但它不象变量那么单纯。
- C++提供一套特殊的机制实现类到对象的转换。

```
class  CCompanyStaff
{
public:
        CCompanyStaff();
        ~CCompanyStaff();
void SetBasicSal ( float sal ); // 设置基本工资
private:
        int m_iStaffNum ; // 工作编号
}; // END
```

对象的创建和销毁

- 创建对象时，“对象存放在何处？？”
- 需要向操作系统申请一定的内存空间用于存放新建的对象。
- 为对象分配存储空间主要有静态分配和动态分配两种方式。

- 可以将对象或静态成员存放在栈中或静态存储区域中。
-
- 动态内存分配是指在堆（也称自由内存）中分配存储单元，即为对象动态从堆中分配内存。
- 使用操作符**new**分配内存空间；
- 使用操作符**delete**释放内存空间。

构造函数与析构函数

- C++语言为了保证一个对象被初始化（类的实例化），定义了一组特殊的方法（函数），专门用于对象生成时的初始化。
- 构造函数（constructor）——与类名称相同，没有返回值。它在对象生成之时自动执行。
- 析构函数（destructor）——在类名前加~作为函数名的函数，没有返回值，也没有参数。它在对象结束时自动执行。

- C++要求类设置一个专门的成员函数来负责类中所有对象的初始化，这个成员函数就是构造函数。
- 构造函数的作用就是在对象被创建时利用特定的值构造对象，将对象初始化到一个特定的状态。
- 声明一个构造函数的语示格式如下：
- **public:**
- 类名（< 参数表 >）；

- 构造函数可以由程序设计人员自己编写。
- 也可以由系统提供。
- 例2.4 构造函数举例。
- sclass2_4_companyStaff.h
- smain2_4.cpp

重载构造函数

- 所谓重载构造函数，是指同一个构造函数名，具有不同的实现。
- 例2.5 在员工管理系统中，创建一个对象时，还可以一次性地给对象的姓名、出勤率、基本工资等几个数据成员都赋初值，这就需要重载构造函数。
- sclass2_5_companyStaff.h
- sclass2_5_companyStaff.cpp
- smain2_5.cpp

- 需要注意一点，当构造函数带默认参数时，要谨防出现歧义。
- 例2.6 下面这个程序存在歧义。
- sclass2_6.h
- smain2_6.cpp
- 当创建对象d2时，有二义性。

拷贝构造函数

- 拷贝构造函数是用来复制对象的一种特殊的构造函数。
- 声明拷贝构造函数的语法格式如下：
- class 类名
- {
- public:
- 　类名（ const类名 &对象名 ）；
- };

只有一个参数

- 例2.7 通过水平坐标和垂直坐标来确定屏幕上的一个点。
- sclass2_7_point.h
- smain2_7.cpp

- **CPoint b(a);**
- **CPoint c = a;**

析构函数

- 析构函数与构造函数的作用几乎正好相反，当一个对象消失时，或用**delete**删除用**new**创建的对象时，系统都会自动调用类的析构函数，做一些清理工作。
- 声明一个析构函数的语法格式如下：
- **class Demo**
- **{**
- **public:**
- **Demo（< 参数表 >）;**
- **~Demo（ void ）;**
- **}**

- 析构函数不能重载。

```
// 析构函数
~CCompanyStaff( void )
{
        cout << "对象" << m_cName << "消亡" << endl;
}
```

- 当某对象消亡时，系统会自动调用该对象的析构函数。
- 而且调用的顺序是：最后创建的对象最先消亡，即最先调用其析构函数；
- 相反地，最先创建的对象最后消亡，即最后调用其析构函数。
- 如果不显式地定义析构函数，系统也会生成一个默认的析构函数，它是一个空的析构函数，不做任何事情。

C++类的对象化（二）

```
class  CCompanyStaff
{
public:
        CCompanyStaff();
        CCompanyStaff(int num, char* pName);
        ~CCompanyStaff();
void SetBasicSal ( float sal ); // 设置基本工资
……
}; // END
CCompanyStaff    staff(12345, “张三”);
```

C++类的对象化（三）

- 1、构造函数与析构函数被“隐含”调用，即不管愿意与否，它们都会被“强制”地执行。
- 2、C++类都有缺省的构造函数，即没有参数的构造函数。它也是“强制”的，没有定义每个类也有一个这样的构造函数。
- 3、编程习惯：不管有用没用，希望编程人员为每个类定义明确的构造与析构函数

对象成员的访问

- 可以通过对象名，也可以通过对象地址来访问一个对象：
- < 类名 > < 对象名 >;
- < 对象名 > .< 成员名 >
  // 访问公有数据成员
- < 对象名 > .< 成员名 >（< 参数表 >）
  // 访问公有成员函数
- **CCompanyStaff staff1("LiHua") ;**
- **staff1.SetBasicSal(4000.0);**

- < 类名 > *< 对象指针名>;
- < 对象指针 > -> < 成员名 >
  // 访问公有数据成员
- < 对象指针 > -> < 成员名 >（< 参数表 >）
  // 访问公有成员函数
- **CCompanyStaff *pstaff ;**
- **pstaff = new CCompanyStaff("LiuMei", 0.95, 3000.0);**
- **pstaff -> GetName( )**

- 例2.8 在员工管理系统中，建立两个对象分别用两种方式去访问类成员。
- sclass2_8_companyStaff.h
- sclass2_8_companyStaff.cpp
- smain2_8.cpp

普通对象指针

- 例2.9 普通对象指针举例。
- sclass2_9_objPointer.h
- smain2_9.cpp
- CPointerExam *pointer;
- pointer = &obj;
- pointer -> SetNum (2);
- 对象指针在使用之前一定要初始化，为其动态分配存储空间；
- 使用完毕必须释放该对象指针所代表的资源。

对象的this指针

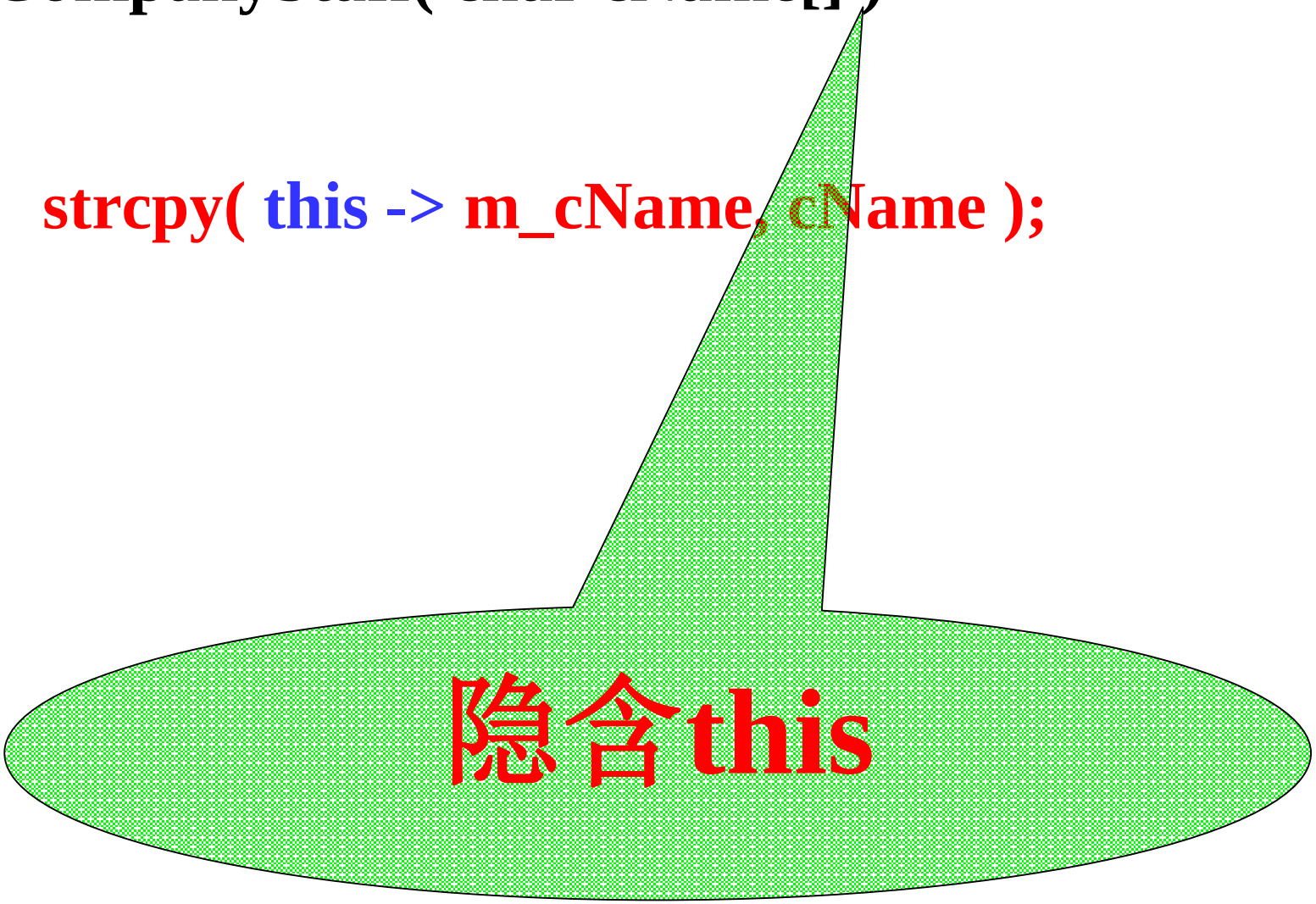
- 每个对象都拥有自己独立的数据成员。
- 而类中的所有对象使用相同的成员函数，成员函数在内存中只有一份。
- 每个对象隐含了一个常量指针，称为this指针，用于指向当前发送消息的对象，以识别当前调用成员函数的对象究竟是谁 。
- 当通过一个对象调用成员函数时，系统先将该对象的地址赋给this指针，成员函数在对对象的数据进行操作时，就隐含地使用了this指针。

- CCompanyStaff(char cName[])
- {
- strcpy(m_cName, cName);
- }

显示指明this

- CCompanyStaff(char cName[], CCompanyStaff *const this)
- {
- strcpy(this -> m_cName, cName);
- }

- CCompanyStaff(char cName[])
- {
- strcpy(this -> m_cName, cName);
- }



- 一般不需要显示指出this，只有当函数需要返回当前对象自身的时候，才显式地使用它。
- 例2.10 this指针举例。
- sclass2_10_this.h
- smain2_10.cpp

```
CSample Add( CSample s1, Csample s2 )
{
        this -> n = s1.n + s2.n;
        return ( *this );
}
```

类成员指针

- 如果指针指向类数据成员的地址，则这个指针称为类数据成员指针；
- < 类型 > < 类名 >::*< 指针名 >
- sclass2_11_dataPointer.h
- smain2_11.cpp
- **int CSample::*p = &CSample::m;**
- **CSample s1;**
- **s1.*p = 20;**
- 类数据成员指针只能指向公有数据成员。

- 如果指针指向类成员函数的地址，则称为类成员函数指针。
- 声明一个类成员函数指针并为其赋值的语法格式如下：
- < 类型 >（< 类名 >::*< 指针名 >）（< 参数表 >）
- < 指针名 > = < 类名 > : : < 成员函数名 >;
- 使用指向成员函数的指针调用函数的格式如下：
- （*< 指针名 >）（< 实参表 >）
-

```
// 主文件：smain2_12.cpp
#include "sclass2_8_companyStaff.h"  // 包含sclass2_8_companyStaff.h的代码
#include<iostream>
using namespace std;
void main( void )
{
    void ( CCompanyStaff :: *pFunc )( float ); // 声明一个类成员函数指针
    CCompanyStaff staff( "LiHua") ;      // 创建一个对象staff
    pFunc = CCompanyStaff:: SetBasicSal;  // 指针初始化指向SetBasicSal函数
    ( staff.*pFunc ) (3000);              // 相当于staff调用SetBasicSal()
    cout << "员工LiHua 的基本工资是" << staff.GetBasicSal();
}
```

对象数组

- 例2.13 对象数组应用举例
- sclass2_13_fruit.h
- smain2_13.cpp
- **CFruit d[4];** // 创建一个对象数组，相应调用4次构造函数
- 如需要建立一个对象数组，必须满足以下条件：
- 至少有一个构造函数没有参数或只带默认参数。

inline与static

- inline（嵌入）专门修饰类中的成员函数，表示该成员函数在使用时会被“替换”。简单地理解，它就是C语言中的宏定义。
- inline后的函数没有调用过程，将有助于代码的执行效率。
- static（静态）修改类中的成员（变量和函数），表明该成员只有一个副本，与具体的对象没有关系。
- static后的成员变量可以用于对象间的数据共享。
- static后的成员函数只能使用static 成员变量。

static成员的使用

- 当一个成员被**static**修饰之后，它的使用不再属于各个对象，而是属于这个类。
- class CCompanyStaff
- {
- public:
  - static void SetBasicSal (float sal);
  - int m_iStaffNum;
- }; // END
- **CCompany::m_iStaffNum = 1234;**
- **CCompany::SetBasicSal(30000);**

静态成员

- 类的静态成员拥有一块单独的存储区。
- 该类的所有对象都共享这块静态存储空间
- 这就为对象提供了一个相互通信的方法。
- 静态成员由关键字**static**标识 。
- 它属于类而不属于对象。
- 它分为静态数据成员和静态成员函数。

- 声明一个静态数据成员的语法格式如下：
- **static** < 数据类型 >< 静态数据成员名 >
- 静态数据成员在使用前也要初始化，但它的初始化不能在构造函数中进行，在类外进行。
- < 数据类型 >< 类名 >::< 静态数据成员名 > = 〈 初始值 〉；
- 其访问语法格式如下：
- < 类名 >： ： < 静态数据成员名 >

- sclass2_14_companyStaff.h
- sclass2_14_companyStaff.cpp
- smain2_14.cpp

- **static int s_iCount;**
- **int CCompanyStaff::s_iCount = 1000;**

- 静态成员函数是一种特殊的成员函数，它属于整个类，也为同类中所有对象共同拥有。
- 只要类存在，静态成员函数就可以使用。
- 可以通过类名和对象名来调用 。
- 其定义语法格式如下：
- < 类名 >::< 静态成员函数名 >(< 参数表 >)
- 例2.15静态成员函数举例。
- sclass2_15.h
- smain2_15.cpp

- 静态成员函数一般不访问普通数据成员，它的作用主要是访问和操作同类中的静态数据成员。
- 类的普通成员函数都拥有this指针 。
- 而静态成员函数没有this指针，但可以通过类名或对象名来实现对它的访问。

对象封装性的局限

- 已经学习过类对于成员的封装（public, private, protected）
- 封装性有效地保护了对象的内部细节，使得对象的使用和对象的实现分开，互相不产生影响。
- 同时，封装性带来负面影响：
- 1、C++为实现对象的封装，必然会做一些额外的工作，从而导致程序的效率下降。
- 2、一个对象封装的太好，也会让该对象很难使用，也很难实现。

突破对象封装

- C++提供了友元（friend）来解决由封装性带来的问题。
- friend关键字修改函数或类，因此对于一个类而言，它有友元函数或友元类。
- friend用于应对编程中一些比较特殊的情况（如提高效率），绝大多数情况下不需要使用。乱使用只会使C++变成C，甚至更糟。

友元关系

- 一个类可以声明一个友元关系，一起来共享类中的所有成员。
- 友元如果是一个函数，则称为友元函数；
- 如果是一个类，则称为友元类。
- 友元函数是在类中由关键字**friend**修饰的非成员函数。
- 友元函数可以是一个普通的函数，也可以是其他类的成员函数。
- 虽然它不是本类的成员函数，但是在它的函数体中可以通过对象名访问类的私有和保护成员。

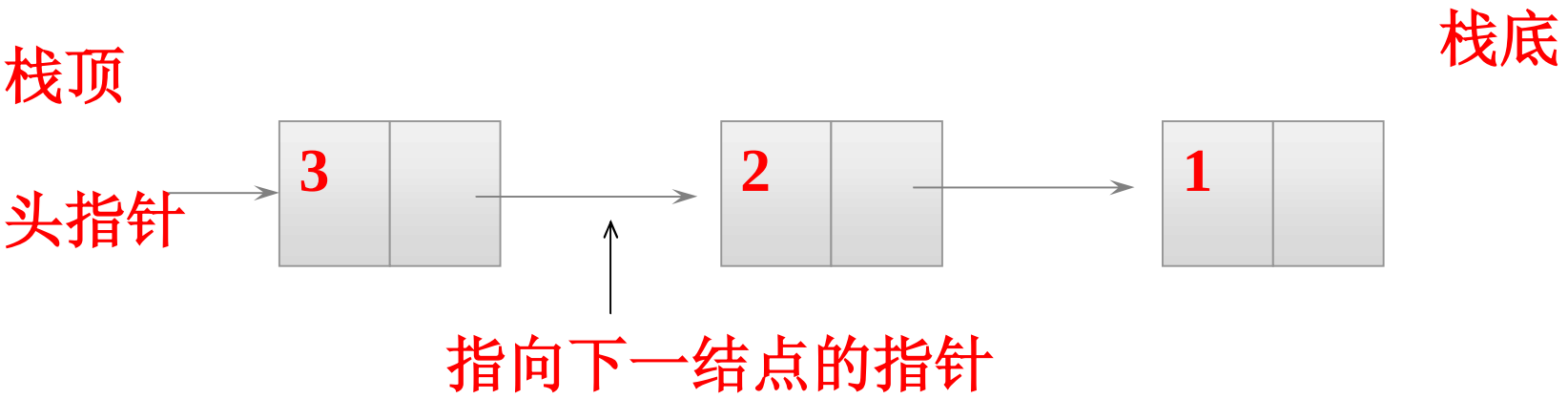
- 例2.16 友元函数与成员函数的比较。
- sclass2_16.h
- smain2_16.cpp

- **friend void FriendFunc (CSample * cp, int a)**
  //增加一个对象指针参数
- **{**
- **cp->i = a;** **//对象指针参数为i指明当前所属对象**
- **}**

- 友元函数的特点：
- 第一、友元函数可以直接访问该类的所有成员，但它不是该类的成员函数，可以像普通函数一样在任何地方调用。
- 第二、友元函数不属于任何类，因此可以放在类说明的任何位置，既可以在public区，也可以在private区。
- 第三、友元函数不需要通过对象或对象指针来调用，可以直接调用即可。

- 在类中可把另一个类声明为友元类，如类B是类A的友元类，则类B中的所有成员函数都是类A的友元函数，都可以访问类A的私有和保护成员。
- 友元类的声明语句如下：
- **class B;　　// 前向引用声明**
- **class A**
- **{**
- **……**
- **friend class B;　// B为A的友元类**
- **……**
- **}**

- 例2.17 编写一个有关栈结构的程序，要求实现入栈和出栈。其中有两个类，一个是结点类CNode，拥有结点值和指向下一结点的指针；另一个是栈类CStack，它拥有栈的头指针。
- 由此生成的链式结构如图2.3所示。



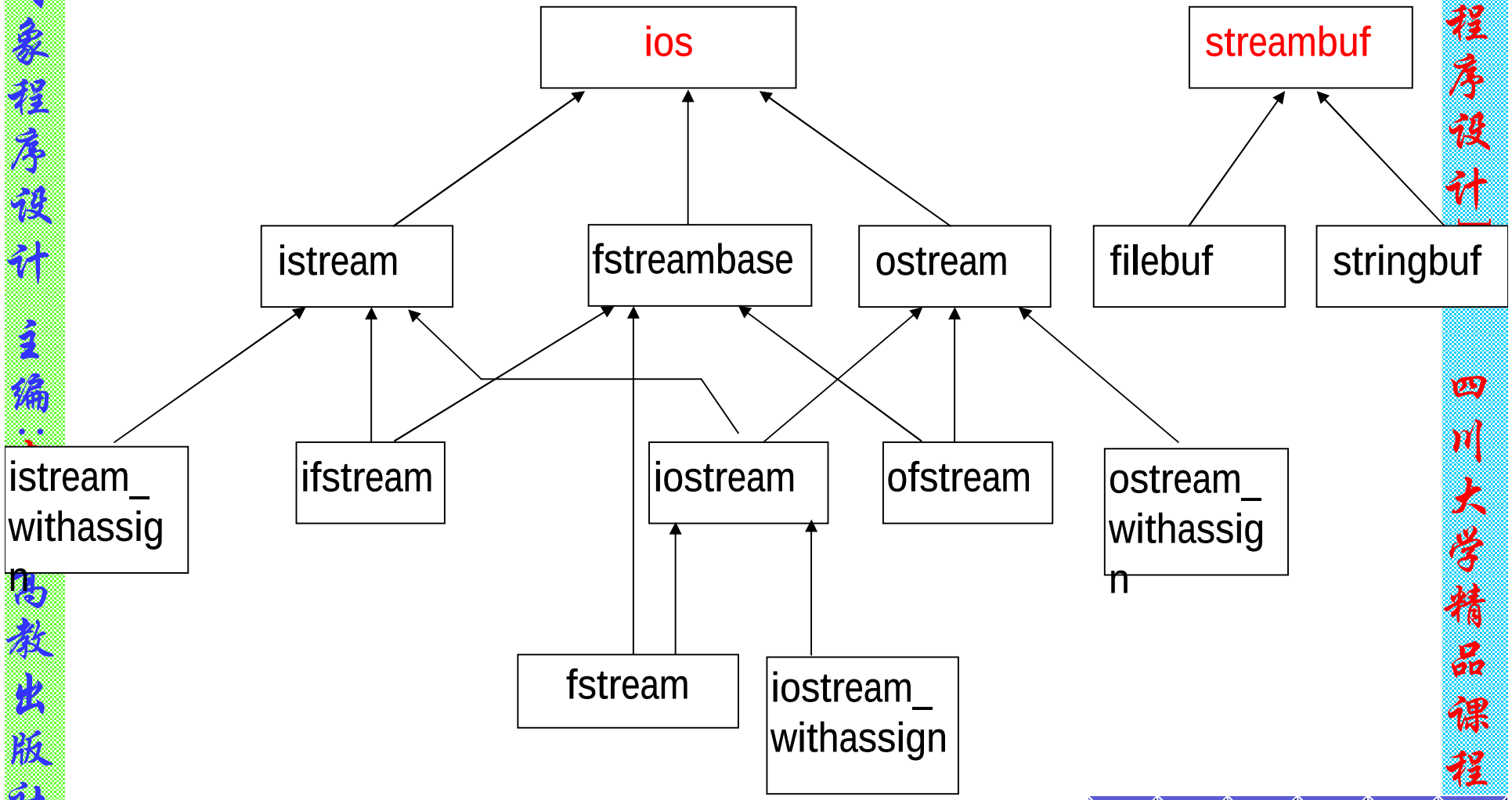
- sclass2_17_stack.h
- sclass2_17_stack.cpp
- smain2_17
- 由于栈类CStack是结点类CNode的友元类，所以栈CStack类的所有成员函数都成为类CNode的友元函数。
- 因此Push( )和 Pop( )可以访问结点类CNode对象的私有成员并对其进行操作。
- 封装是对象与外界之间一堵不透明的墙，而友元恰好在这堵墙上开了一个小孔，它以牺牲信息隐藏、削弱封装性为代价来实现数据共享。

2.3 C++输入输出流对象

- 标准输入输出流：数据从程序中流入到屏幕或磁盘文件，即输出流；数据从键盘流入到程序中，即输入流。
- 所谓流，是从源到矢的数据流的抽象引用，具体地说，就是数据从一个对象流向另一个对象。
- 在进行I/O操作时，首先执行打开操作，使流和文件发生联系，建立联系后的文件才允许数据流入或流出，输入输出结束后，执行关闭操作使文件与流断开联系。

C++流类库

- C++提供了一个流类库，它具有两个平行的根基类：**streambuf类和ios类**。



- **streambuf类主要负责缓冲区的处理，提供对缓冲区的低级操作。**
- **ios类是流基类，它及其派生类提供用户使用流类所需的接口，支持对streambuf的缓冲。**
- **由它可以派生出输入流类istream和输出流类ostream等。**
- **两个根基类以及由它们派生出的所有流类（当然包括istream类和ostream类）都被定义在名为iostream的头文件中。**
- **因此须用“#include”编译指令将iostream头文件包含进来。**

表 2.1 流类列表

类　名	含　义	包含文件
抽象流基类		
ios	基类流	ios
输入流类		
istream	通用输入流类和其他输入流的基类	istream
ifstream	输入文件流类	fstream
istream_withassign	可复制对象的输入流类	istream
输出流类		
ostream	通用输出流类和其他输出流的基类	ostream
ofstream	输出文件流类	fstream
ostream_withassign	可复制对象的输出流类	ostream
输入/输出流类		
iostream	通用输入/输出流类和其他输入/输出流的基类	iostream
fstream	输入/输出文件流类	fstream
iostream_withassign	可复制对象的输入输出流类	iostream
流缓冲区类		
streambuf	抽象流缓冲区基类	streambuf
filebuf	磁盘文件的流缓冲区类	fstream
stringbuf	字符串的流缓冲区类	sstream

表 2.1 流类列表

类　名	含　义	包含文件
抽象流基类		
ios	基类流	ios
输入流类		
istream	通用输入流类和其他输入流的基类	istream
ifstream	输入文件流类	fstream
istream_withassign	可复制对象的输入流类	istream
输出流类		
ostream	通用输出流类和其他输出流的基类	ostream
ofstream	输出文件流类	fstream
ostream_withassign	可复制对象的输出流类	ostream
输入/输出流类		
iostream	通用输入/输出流类和其他输入/输出流的基类	iostream
fstream	输入/输出文件流类	fstream
iostream_withassign	可复制对象的输入输出流类	iostream
流缓冲区类		
streambuf	抽象流缓冲区基类	streambuf
filebuf	磁盘文件的流缓冲区类	fstream
stringbuf	字符串的流缓冲区类	sstream

- 系统预定义的一些流：
- 进行输入输出常用的标准输入流对象**cin**和标准输出流对象**cout**,
- 还有未被缓冲的标准错误输出**cerr**和被缓冲的标准错误输出**clog**。
-

- cin是通用输入流类istream_withassign的对象，与标准输入设备连接。
- 它通过重载运算符 “>>”执行输入操作，在流操作中将“>>”称为提取运算符。
- cin从输入流中取出数据，数据从提取运算符“>>”处流进程序。

- cout是通用输出流类ostream的对象，与标准输出设备连接。
- 通过重载运算符“<<”执行输出操作，在流操作中，将“<<”称为插入运算符。
- 插入运算符“<<”向输出流发送字符。实际上，位于插入运算符右侧的字符串被存储在“<<”左侧的流中。
- cout << m <<" "<< n << endl;
- cout << m;
- cout <<“ “;
- cout << n;
- cout << endl;

预定义操纵符

- 操纵符是直接插入到流中的格式化指令，一般都定义在ios_base类中和<iomanip>头文件中，分为带参数和不带参数两类。

表 2.2 无参数的 ios 操纵符

操 纵 符	用 途
Ws	输入时跳过空格
Dec	转化为十进制
Oct	转化为八进制
Hex	转化为十六进制
Endl	在输出流中插入换行符并刷新它
Ends	插入空字符以结束输出
Flush	刷新输出流
Lock	锁定文件句柄
Unlock	解锁文件句柄

表 2.3 带参数的 ios 操纵符

操 纵 符	用 途
setw( )	输出时设置字段宽
setfill( )	输出时设置填充字符
Setprecision( )	设置精度 (显示多少位数字)
setiosflags( )	设置指定标志
Resetiosflags( )	消除指定标志

- 例如：
- **cout << setw(5) << n << endl;**
- 表示设置输出n时所占的字节长度为5，输出n后换行。

文件输入输出流

- 输入是指从磁盘文件流向内存。
- 输出是指从内存流向磁盘。
- C++提供了3个文件流类：ofstream，ifstream，fstream，都定义在头文件fstream.h中。
- 其中fstream类以ofstream类和ifstream类为基类。
- ofstream类：输出流类，用于向文件中写入内容；
- ifstream类：输入流类，用于从文件中读出内容；
- fstream类：输入输出流类，用于既要读又要写的文件操作。

- **ofstream:: ofstream(char *pFileName, int mode=ios::out, int prof=filebuf::openprot);**
- 第一个参数用于指业文件路径及文件名字符串，
- 第二个参数说明文件打开方式，
- 第三个参数说明文件保护方式。

文件流对象构造函数

- 例2.18 给出程序的执行结果
- smain2_18.cpp

表 2.4 文件打开方式

操 纵 符	用 途
ios::ate	如果文件存在，输出内容加在末尾
ios::in	具有输入能力（ifstream 默认）
ios::out	具有输出能力（ofstream 默认）
iIos::trunc	如果文件存在，清除文件内容（默认）
ios::nocreate	如果文件不存在，返回错误
ios::noreplace	如果文件存在，返回错误
ios::binary	以二进制方式打开文件

表2.5 文件保护方式

操 纵 符	用 途
filebuf::openprot	兼容共享方式
filebuf::sh_none	独占，不共享
filebuf::sh_read	允许读共享
filebuf::sh_write	允许写共享
filebuf::openprot	兼容共享方式

常用输入输出流成员函数

- 例2.19 下面的程序将用户输入显示到屏幕上，输入字母y时输出OK并结束。
- smain2_19.cpp
- 例2.20 连续读入一串字符，直到遇到字符q时停止，然后输出这个字符串。要求字符个数最多不超过79个。
- smain2_20.cpp

- 例2.21 将英文字母及对应的ASCⅡ值输出到屏幕上。
- smain2_21.cpp

C++流的应用

- 1、C++专门为交互式输入与输出定义了两个对象：cin用于键盘的输入；cout用于显示屏的输出。
- 2、所有的C++流都有两个操作符 << 和 >>
- << ：输出（插入）操作符。流对象<<数据；
- >> ：输入（提取）操作符。流对象>>数据；

输入输出流应用举例

- 例2.22 员工管理系统中的输入输出
- sclass2_22_companyStaff.h
- sclass2_22_companyStaff.cpp
- smain2_22.cpp

Thanks!

The End