C++程序设计教程(第二版)

Chapter 8 Class

苏州大学计算机科学与技术学院

引言

- 类

  类是一种可以作为交易的概念代码．类是自定义的数据类型．类与应用程序不同，类的功能虽然独立，但不能独立启动，就象汽车零件那样.

- 类机制
  - 要通过编程的方法来维护类的数据表达，值范围和操作并不是简单的，因为要限制一些操作只能让类自己来做，以免发生问题时，无人敢对其负责.
  - 类机制就是通过语言的规定性来实现一些技术，使类具有数据封装，信息屏蔽，多态等特征，起到数据类型的作用，而达到方便编程的目的.

主要内容

1. 从结构到类(From Structure to Class)
2. 成员函数 (Member Functions)
3. 操作符 (Operators)
4. 再论程序结构 (Program Struture Restatement)
5. 屏蔽类的实现 (Shield Class Implementations)
6. 静态成员 (Static Members)
7. 友元 (Friends)

8.1 从结构到类

- 结构体：数据类型

```
struct  Date{
        int day, month, year;
}
```

- 单纯堆积数据空间构造的类型，不含有相关操作，数据和操作是完全分离的！
- 所有相关操作都必须由使用结构体的程序员自己承担，不便于代码的移植和重用
- 仅是一个不完整的自定义数据类型

8.1 从结构体到类

- 类
  - 概念：不但描述数据空间,还描述其操作的自定义类型
  - 定义：

```
class CDate
{
    private:
        int year,month,day;
    public:
        void setdate(int,int,int);
        ……
};
```

  - 举例（f0802）

- 说明：
- 1.class:定义类的关键字
- 2.CDate: 类的名字
- 3.public:/private:规定了类成员的访问权限
- 4.类成员包括*数据成员*和*成员函数*
- 4.{}:类定义的边界
- 5.";":类定义结束符，不可省略

多文件结构的f0802:f0802.h

```
#ifndef  _DATE_H
#define  _DATE_H

class CDate
{
private:
   int year, month, day;
public:
   void set(int,int,int);   // 赋值操作
   bool isLeapYear();       // 判断闰年
   void print();            // 输出日期
};
#endif
```

多文件结构的f0802：f0802. cpp

```
void CDate::set(int y,int m,int d){
  year=y; month=m; day=d;
}
//------------------------------
bool CDate::isLeapYear(){
  return (year%4==0 && year%100!=0)||(year%400==0);
}
//------------------------------
void CDate::print(){
  cout<<setfill('0');
  cout<<setw(4)<<year<<'-'<<setw(2)<<month<<'-
'<<setw(2)<<day<<'\n';
  cout<<setfill(' ');
}//------------------------------
```

- 规定函数所从属的类型
- ::是域作用运算符

多文件结构的f0802：driver_f0802.cpp

```
int main(){
   CDate d;
   d.set(2000,12,6);
   if(d.isLeapYear())
      d.print();
}//=================================
```

8.1 从结构体到类

- 类
  - 数据成员
  - 成员函数
  - 成员的权限
    - private: // 私有成员
    - protected: // 保护成员

    } 外部函数无法访问的成员
    - public: // 公有成员：外部函数可以直接访问的成员
  - 类的定义包含两个部分：
    - 类定义本体
    - 成员函数的定义
  - 一般情况下类的定义和应用程序为多文件结构

8.1 从结构体到类

- 类成员访问权限说明：
  - 公有的（public）：公有的成员可以被程序中的任何代码访问；
  - 私有的（private）：私有的成员只能被类*本身的成员函数*及*友元类的成员函数*或*友元函数*访问，包括其派生类的成员函数都不能访问它们；
  - 保护的（protected）：保护的成员与私有成员类似，只是除了*类本身的成员函数*和说明为*友元类的成员函数*可以访问保护成员外，该类的*派生类的成员函数*也可以访问。

⌧ 说明：当没有说明类的成员的访问权限修饰符时，默认为private。

8.1 从结构体到类

- 类：对象与变量
  - 变量：由内部数据类型或衍生的结构类型所产生的实体
  - 对象：由类产生的实体，本质上，变量也是对象，只不过粗糙一点罢了
- 类和结构体的比较
  - 结构体是开放的
  - 类是封闭的，具有自成一体性
  - 类的使用使得程序开发的流程分工变得极为清晰
- 类定义举例：(personclass)
  - 定义一个person类，该类具有姓名、年龄、性别、生日四属性。要求提供类对象的初始化成员函数、类对象属性的显示成员函数。

8.2 成员函数

- 成员函数与普通函数的区别
  - 成员函数属于类，成员函数定义是类设计的一部分，其作用域是类作用域. 而普通函数一般为全局函数
  - 成员函数的操作主体是对象，使用时通过捆绑对象来行使其职责，而普通函数被调用时没有操作主体

8.2 成员函数

- 成员函数的定义方法
  - 类内部定义
    - 优点：系统会根据成员函数的性质进行自动内联处理。
    - 缺点：导致类的定义体大小不可预见，失去了类定义体作为类描述的参考功能。
  - 类外部定义：
    - 优点：类的定义体大小可以预见，保留了类定义体作为类描述的参考功能。
    - 缺点：系统不会根据成员函数的性质进行自动内联处理。
    - 解决方法：在成员函数的定义时，显示声明为内联函数
  - 一般方式：小的函数在类内部定义，大的函数在类外部定义。

8.2 成员函数

◆成员函数的定义方法

```
class CDate
{
        int year, month, day;
    public:
        void  set ( int y, int m, int d )   // 默认内联
        {
        year=y; month=m; day=d;
        }
        …
};

inline  bool  CDate::isLeapYear ( )     // 显式内联
{
    return ! ( year%400 ) || !(year%4) && year%100;
}
……
```

8.2 成员函数

- 成员函数的访问
  - 对象方式
    - Date d;
    - d.set(2013,4,9);
  - 对象指针方式
    - Date d;
    - Date *pd = &d;
    - pd->set(2013,4,9);
    - (*pd).set(2013,4,9);
    - Date *p2 = new Date;　　//动态内存分配
    - p2->set(2013,4,9);
    - …
    - delete p2;　　// 内存释放

8.2 成员函数

- 常成员函数
  - 概念：对捆绑的对象，不允许写操作的成员函数。
  - 定义方式：在成员函数行参列表的右括号后面加const。
  - 例如：

```
class Rectangle
{
    private:
        float width, length, area;
    public:
        …
    float getWidth(void) const;
    float getLength(void) const;
    float getArea(void) const;
};
```

⌧优点：
- 便于调试
- 控制软件质量

结论：
- 能成为常成员函数的，应尽量定义成常成员函数

注意：
- 区分float getWidth() const与const float getWidth()
- 前者是常成员函数，后者是返回常量的成员函数

8.2 成员函数

- 常成员函数与const限定的函数参数
  - 常成员函数：在函数中不允许改变所捆绑对象的属性值
  - const限定的函数参数：在函数中不允许修改参数的值
  - 常成员函数中不允许调用非常成员函数

```
bool Date::comp(const Date& a)const
{
    year = 2005;          // error: 常成员函数捆绑的对象
    a.year = 2003;        // error: 常量对象
    return year= =b.yaer &&
           month= =a.month &&
           day= =a.day;
  }
```

- 常成员函数的服务对象一般是类的使用者

8.2 成员函数

- 成员函数重载
  - 类的成员函数允许重载
  - 重载只体现在成员函数之间，与外部函数无关
  - 成员函数重载的规则与普通函数重载的规则相同
  - 举例(f0804)
  - 举例：
    - 对前面的person类进行改进，增加对属性的如下操作：
      - 修改姓名操作(提供C串和string类对象为参数的形式)
      - 年龄增加操作
      - 获取年龄操作
      - 比较两个人的年龄大小操作

补充：this指针

- 数据和操作分离的情况
  - 数据类型

```
struct  Date{
        int day, month, year;
}
```

  - 数据操作

```
void set(Date &d, int, int, int);        //初始化d
void add_year(Date &d, int n);              //d加n年
void add_month(Date &d, int n);           //d加n月
void add_day(Date &d, int n);               //d加n天
```

补充：this指针

- 数据和操作结合的情况

```
class CDate
{
        int year, month, day;
    public:
        void  set ( int y, int m, int d )    // 默认内联
        {
        year=y; month=m; day=d;
        }
        void  AddYear(int n){year += n;}
       void AddMonth(int n)
      {
          month += n;
          if (month > 12){year++;month -= 12;}
      }
       void AddDay(int n){ day += n; …… }
};
```

补充　this指针

```
// 数据和操作分离情况
int main() {
    Date   date;
    set(date,12, 03,21);
    add_year(date,1);
    add_month(date,2);
    add_day(date, 3);
}
```

```
//数据和操作结合情况
int main() {
    Date   date;
    date. set(12, 03,21);
    date.AddYear(1);
    date.AddMonth(2);
    date.AddDay(3);
}
```

- 成员函数如何绑定到特定对象？
  - this指针

补充：this指针

- 何时使用this指针
  1. 用于成员函数返回一个对象引用

```
Screen& Screen::clear( char bkground )
{
    // 重置 cursor 以及清屏幕
    _cursor = 0;
    _screen.assign(                 // 赋给字符串
        _screen.size(),             // size() 个字符
        bkground                    // 值都是 bkground
    );
    // 返回被调用的对象
    return  *this;
}
```

补充：this指针

- 何时使用this指针

2. 获取对象的地址

```
void  Screen::copy( const Screen&  sobj )
{
    // 如果 Screen 对象与 sobj 是同一个对象
    // 无需拷贝
    if ( this != &sobj )
    {
        // 把 sobj 的值拷贝到 *this 中
    }
}
```

小结：

- 通过this指针获取类对象

8.3 操作符

- 一个复数类
  - 如何实现复数类对象的算术运算？
  - 如何实现复数类对象的赋值运算？

- 答案：
  - 成员函数
  - 操作符重载

```
class complex
{
private:
        double re, im;     //
Public:
        //操作
        void add(const complex &cm);
             ……..
}
```

8.3 操作符

- 操作符定义的函数重载特征
  - 操作符在类中定义的目的：方便编程和直观上的理解
  - 操作符定义就是函数定义，调用操作符就是调用函数
  - 操作符定义必须使用关键词：operator
  - 举例(f0805)

```
Point operator+(const Point& a, const Point& b)
{
    Point s;
    s.set(a.x + b.x, a.y + b.y);
    return s;
}
//.....................................................
    Point p, q;
    p.set(3, 2);
    q.set(1, 5);
    Point r = p + q;  // p + q 等价于 operator+(p, q)
```

8.3 操作符

- 操作符定义的作用
  - 操作符定义并不是必要的，只是为了在编程中进行人性化描述，达到更方便地理解程序的目的
- 操作符定义的性质
  - 不允许创建新的操作符： 如 @
  - 个别操作符限制重载：如”::“ ；“.”；”?:”等。对这类操作符的重载会严重破坏C++的语法规则
  - 操作符的优先级别和结合性不允许改变
    - Point a,b,c;
    - Point d=a+b*c;
  - 操作数个数不允许改变
  - 专门处理对象
  - 禁止使重载后的操作符的意义发生根本变化

8.3 操作符

- 运算符的重载形式有三种：重载为类的成员函数、重载为类的友元函数，重载为一般的函数。

- 运算符重载为类的成员函数的语法形式如下：

```
class class_name
{
    ...
    〈返回类型〉 operator 〈运算符〉(〈形参表〉)
    {
        〈函数体〉;
    }
}
```

例如:

```
class String {
public:
    String& operator=( const String &);// 赋值操作符的重载集合
    String& operator=( const char * );
    char& operator[]( int ) const;    // 重载的下标操作符
    bool operator==(const char*)const;//等于操作符的重载集合
    bool operator==( const String & ) const;
    String& operator+=( const String &); //+=操作符的重载集合
    String& operator+=( const char * );
    int size() { return _size; }  // 成员访问函数
    char* c_str() { return _string; }
private:
        int _size;
        char *_string;
};
```

例如(续)

```
include <cstring>
inline String& String::operator+=( const String &rhs )
{
     if ( rhs.size != 0) // 如果 rhs 引用的 String 不为空
    {
          char* pstr = this->_string;
          int len = this->size;
       _size += rhs._size;
       // 创建足够大的存储区，以便包含被连接之后的 String
       _string = new char[ _size + 1 ];
       // 把原来的 String 拷贝到新的存储区中,然后附加上 rhs 所指的String
        strcpy( _string, pstr);
        strcpy( _string + len, rhs._string );
         delete[] pstr;      //非常重要: 释放原来的数据空间
      }
     return *this;
}
```

例如(续)

```
inline String& String::operator+=( const char *s )
{
  if ( s ) // 如果 s 不是空指针
  {
            char* pstr = this->_string;
            int len = this->size;_
            size += strlen( s );
      //创建足够大的存储区 以便包含被连接之后的 String
      _string = new char[ _size + 1 ];
// 把原来的 String 拷贝到新的存储区中,然后, 附加上 s 所指的 C 风格字符串
      strcpy( _string, pstr);
      strcpy( _string + len, s );
      delete[] pstr;       //非常重要: 释放原来的数据空间
  }
  return *this;
}
```

8.3 操作符

几点说明

- 当运算符重载为类的成员函数时，函数的参数个数比原来的运算数个数要少一个（后缀++、--除外）；当重载为类的友元函数时，参数个数与原运算数的个数相同。
  - bool operator==(const String &);　　//成员函数
  - bool operator==(const String &, const String &); //友元或普通函数

一个操作符声明为类成员还是名字空间成员的原则

- 如果一个重载操作符是类成员，那么只有当跟它一起被使用的左操作数是该类的对象时它才会被调用，如果该操作符的左操作数必须是其他的类型，那么重载操作符必须是名字空间成员
- C++要求赋值(=)、下标([])、调用(())和成员访问箭头(->)操作符必须被定义为类成员操作符，任何把这些操作符定义为名字空间成员的定义都会被标记为编译时刻错误.
- 说明：一般来讲，单目运算符最好重载为成员函数，而双目运算符则最好重载为友元函数。运算符重载的主要优点就是允许改变使用于系统内部的运算符的操作方式，以适应用户新定义类型的类似运算
- 输入输出操作符必须重载为友元函数！！！
- 建议：将所有运算符都重载为友元函数！！！

例如:

```
bool operator==(const char*)const;//等于操作符的重载集合
bool operator==( const String & ) const;
```

```
#include "String.h"
int main()
{
    String flower;
    // 设置 flower
    if ( flower == "lily" )
    // ...
    else
        if ( “tulip” == flower )
    // ...
}
```

```
// ok: 等价于调用
flower.operator==(“lily”)
```

```
// 错误, why?
没有重载函数匹配
```

8.3 操作符

- 操作符重载时的值返回和引用返回
  - 值返回：返回临时表达式的值，例如：

```
Point operator+(const Point& a, const Point& b)
{
    Point s;
    s.set(a.x+b.x, a.y+b.y);
    return s;
}
```

  - 引用返回： 如果有将返回值放入参数中并需要进行连续操作的需要，便需要以引用形式返回参数。例如：

```
ostream& operator<<(ostream& o, const Point& d)
{
    return o<<"("<<d.x<<","<<d.y<<")\n";
}
```

8.3 操作符

- 增量操作符的重载
  - 增量操作符重载时的特殊性：前后增量的形式相同但意义不同
  - 前增量运算符分析：运算后表达式结果和变量自身结果的一致，且要求进行变量本身的左值连续运算
  - 后增量运算符分析：运算后表达式结果和变量自身结果是不一致的，且不允许进行变量本身的左值连续运算
  - 处理方法：使参数和返回值类型不同
  - 后增量运算符重载时的编译特殊处理：实参和形参个数不一致
  - 举例：f0806

8.3 操作符

- “=”赋值操作符重载
  - 赋值运算符“=”的原有含义是将赋值号右边表达式的结果拷贝给赋值号左边的变量，通过运算符“=”的重载将赋值号右边对象的数据依次拷贝到赋值号左边对象的数据中。
  - 在正常情况下，系统会为每一个类自动生成一个默认的完成上述功能的赋值运算符，当然，这种赋值只限于由一个类类型说明的赋值。

注意：

如果一个类包含指针成员，采用这种默认的按成员赋值，那么当这些成员撤消后，内存的使用将变得不可靠。

Why?

实例

```
#include<iostream>
class Person{
   int age;
   char sex;
   char *name;
public:
   …
     void SetName(char* pname){name=pname;}
     char* GetName(){return name;}

};
```

@ 问题：默认的赋值运算符=可行吗？
- 例如：Person p1,p2;
- p1.Setname(“Jone”);
- p2=p1;

@ Why?

实例（续）

```
#include<iostream>
class Person{
   int age;
   char sex;
  char *name;
public:
   …
   void SetName(char* pname)
   {   name=pname;  }
   char* GetName(){return
   name;}
   Person& operator=(Person&);

};
```

```
Person& Person::operator=(Person& p)
{
      age=p.getAge();
      sex=p.getGender();
      name=new
   char[strlen(p.GetName())+1];
      strcpy(name,p.GetName());
       …
      return p;
}
```

例如： Person p1,p2;

p1.Setname(“Jone”);

p2=p1;

Why?

8.3 操作符

- 输入输出运算符重载
- 注意：
  - 输入输出运算符只能重载为非成员函数，如果需要访问类的私有成员，则必须重载为类友元函数。
- 重载格式：
  - ostream& operator <<(ostream&, const ClassType &);
  - istream& operator>>(istream&, ClassType &);

“<<“运算符重载的通用框架

```
// 重载 output 操作符的通用框架
ostream&
operator <<( ostream& os, const ClassType &object )
{
    // 准备对象的特定逻辑
    // 成员的实际输出
    os << // ...

     // 返回 ostream 对象
    return os;
}
```

“>>“运算符重载的通用框架

```
// 重载 input 操作符的通用框架
istream&
operator >>( istream& is,  ClassType &object )
{
      // 准备对象的特定逻辑
      // 成员的实际输入
      is >>// ...

       // 返回 istream 对象
      return is;
}
```

8.3 操作符

- 举例：
  - 定义一个复数类，并给出该类的如下功能：
    - 加法运算符重载函数
    - 减法运算符重载函数
    - 乘法运算符重载函数
    - 实部前自增函数
    - 输入操作符
    - 输出操作符
    - 给出该类的测试程序
    - 具体见：complex

8.4 再论程序结构

- 访问控制
  - 成员函数 一般为公有public。公有的成员函数在类的外部可以被使用，即外界可以调用成员函数.
  - 数据成员 一般为私有private。私有的数据成员在外部不能被访问，即外界不能访问对象的数据分量，而只能由成员函数内部去处理.
  - 公有和私有可任意设定
  - 访问控制public和private是语言提供给程序员的功能：类的内部和外部被隔绝
  - 类的界面（接口）：类全部公有成员函数的声明

8.4 再论程序结构

- 类的程序结构
  - 举例：f0809
  - 类定义作为头文件，如：point.h
  - 类的实现作为独立编译单元，如：point.cpp
  - 使用类的程序作为另一独立编译单元，如：f0809.cpp
  - 类的头文件和类的实现可以作为一个独立的资源提供给编程者。
  - 内联的成员函数定义一般放在头文件中。头文件中必须使用头文件卫士技术。

8.4 再论程序结构

- 类作用域
  - 类定义作用域：
    - 从类定义结束开始，到从外面包围类定义的块结束(若类定义外无包围块，则结束于文件)；
    - 使用类的程序员在类定义作用域下编程
  - 类作用域：
    - 类定义内部及成员函数定义内部；
    - 实现类的程序员在类作用域下编程
  - 举例(f0810)
  - 类作用域中成员变量被局部变量屏蔽的现象及其理解
  - 类作用域其实是类定义作用域的子集

8.5 屏蔽类的实现

- 使用类的应用程序只需要类定义头文件编程
- 实现类，也只需要类定义头文件，不需要使用类的程序细节
- 确定了类定义(头文件)，便可以从事两方面的编程而互不干涉。
- 类定义成功地屏蔽了类的实现，是类机制的技术体现。
- 举例

8.6 静态成员

静态数据成员

静态数据成员的需要性

- 类中属性的共有性
- 使用全局变量来解决属性共有性问题的缺陷
- 应该属于类，但不能使用普通成员变量的形式来实现属性共有
- 举例：f0812

8.6 静态成员

- 静态数据成员的使用
  - 概念：属于类的全部对象所有的静态成员变量称为静态数据成员，它对于每个类而言只有一个实体，每个对象中不再有它的副本。
  - 定义方法：分为声明和定义初始化两个形式
  - 举例：f0813
  - 合理的静态数据成员定义初始化位置
    - 类定义头文件（类定义体内部和外部）：Ok
    - main函数所在文件的的开头：Ok
    - 类定义内部实现部分（即成员函数定义的位置） :OK

8.6 静态成员

- 静态成员函数
  - 调用时，不捆绑对象，所以，不能直接操作对象和其成员，若需访问该类对象，必须以参数传递之.
  - 静态成员函数一般设计为公有的，以访问私有静态数据成员为目的.
  - 静态成员函数一般不能访问普通成员变量
  - 静态成员函数一般不能调用非静态成员函数。
  - 调用方式是以类名加域操作符::后跟静态成员函数、也可以通过对象调用，但更偏好使用前者。
  - 举例：f0814

8.7 友元

- 友元的概念
  - 需要使用友元的原因：某些类以外的函数（如普通函数等）需要直接访问某个类的保护或私有成员
  - 需要使用友元的目的：提高效率
  - 友元使用的后果：破坏了类的封装特性（除特殊情况，不推荐使用）
  - 举例（f0815)：若以普通函数的身份实现，则要大量调用成员函数去访问私有数据成员，而用友元之后，可以直接访问之。

8.7 友元

- 友元的使用
  - 举例(f0816)
    - 友元函数使用的典型场合
    - 某普通函数需要频繁通过成员函数访问某类的私有成员变量
  - 用在无法成员化的操作符重载中：<<
  - 友元函数不是类的成员函数，它是类的朋友
  - 友元函数声明的位置可在类声明中的任何区域位置（Public、Private或Protected均可）
  - 友元函数的定义一般放在类的实现文件中进行定义
  - 一个类的成员函数可以是另一个类的友元
  - 整个类可以是另一个类的友元，但友元关系不能传递（A->B->C)，不表示(A->C)。

本章小结

- 类的定义：
  - 几类访问权限？各自的具体权限是什么？
- 成员函数
  - 成员函数的运行原理？
  - this指针
  - 什么是常成员函数？
  - 内联成员函数原理
- 操作符重载
  - 操作符重载的函数名称？
  - 操作符重载为几类形式的函数？
  - 如何确定操作符重载函数的参数个数？

本章小结

- 操作符重载
  - 哪几类操作符只能重载为成员函数？
  - 哪几类操作符只能重载为友元函数？
  - 自增自减操作符的重载、赋值操作符重载
- 程序结构和面向对象对编程的影响：
  - 要求理解，但暂时不理解也不影响对C++的学习
- 静态成员
  - 静态数据成员：定义、初始化、引用方式
  - 静态成员函数：声明、定义、操作的数据成员有哪些？
- 友元
  - 概念和使用