第7章 类

——C++程序设计

对外经济贸易大学 雷擎
leiqing@uibe.edu.cn

内容

- 7.1 定义抽象数据类型
- 7.2 访问控制与封装
- 7.3 类的其他特性
- 7.4 类的作用域
- 7.5 构造函数再探
- 7.6 类的静态成员
- 对象数组

7.1 定义抽象数据类型

- 什么是类
  - 类是一种复杂的抽象数据类型
  - 类定义中包括构造函数、数据成员和成员函数的三部分定义，每一部分都可以省略。

- 使用关键字class定义类

```
class classname{
        constructors defination;
        data members defination;
        member functions;
}
```

类定义的一般形式

- 对类类型的声明，可得到其一般形式如下：

```
class 类名   {
        private :
                私有的数据和成员函数;

        public :
                公用的数据和成员函数;
                构造函数
};
```

```
struct Student { //声明了一个名为Student的结构体类型
    int num;
    char name[20];
    char sex;
};
Student stud1, stud2;
//定义了两个结构体变量stud1和stud2，它只包括数据，没有包括操作

class Student {//以class开头
   int num;
   char name[20];
   char sex; //以上3行是数据成员

   void display( ) {//这是成员函数
      cout<<"num:"<<num<<endl;
      cout<<"name:"<<name<<endl;
      cout<<"sex:"<<sex<<endl;
      //以上4行是函数中的操作语句
   }
};
Student stud1,stud2; //定义了两个Student 类的对象stud1和stud2
```

数据成员定义

- 在类体内声明数据成员时，不允许对数据成员初始化。

const数据成员

- 在类的成员定义中，使用修饰符const说明的数据成员称为常数据成员。

- 常数据成员必须初始化，并且不能被修改。常数据成员是通过构造函数的成员初始化列表进行初始化的。

- 格式：

  const 类型名 变量名;

定义成员函数

- 所有成员必须在类内部声明，但成员函数体可以定义在类内也可以定义在类外

- 定义成员函数的一般形式为：

```
返回值类型 成员函数名（参数表）
        {
            函数体;
        }
```

在类的外部定义成员函数

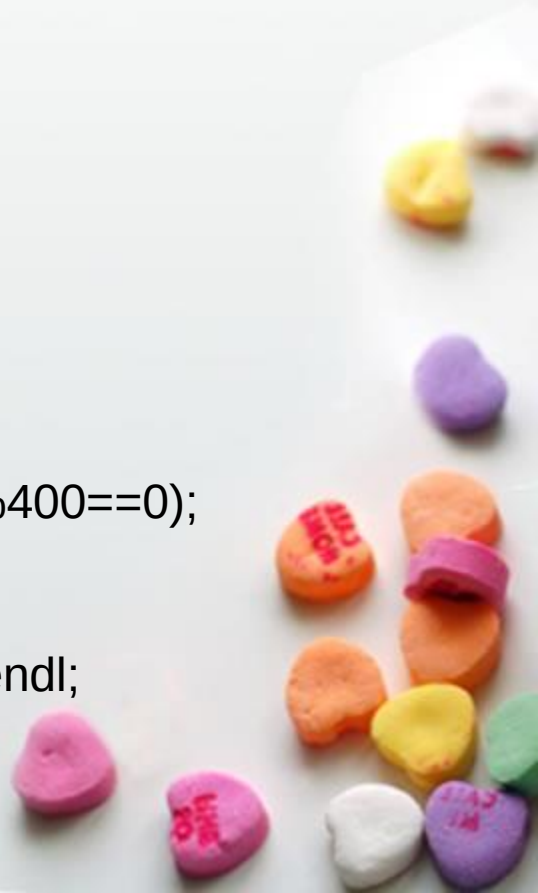
- 在函数的名称之前加上其所属的类名及域运算符(scope operator)“::”。
- 定义成员函数的一般形式为：

```
返回值类型 类名::成员函数名（参数表）
{
    函数体;
}
```

- 域运算符用来指明哪个函数或哪个数据属于哪个类，所以使用类中成员的全名是：类名::成员名。

```
class TDate {
    public:
            void SetDate(int y, int m, int d);
            int IsLeapYear();
            void Print();
    private:
            int year, month, day;
};


void TDate::SetDate(int y, int m, int d){
            year=y;
            month=m;
            day=d;
}
int TDate::IsLeapYear(){
            return(year%4==0 && year%100!=0) || (year%400==0);
}
void TDate::Print(){
            cout<<year<< ‘,’ <<month<< ‘,’ <<day<<endl;
}
```



使用头文件

- 可以将类定义和成员函数的定义分开，前者是类的外部接口，后者是类的内部实现。

- 一般习惯将类的定义放在一个头文件中，以后的程序中需要使用这个类的时候，只需通过文件包含命令将头文件包含到程序中即可。

```
//tdate.h
class TDate {
    public:
            void SetDate(int y, int m, int d);
            int IsLeapYear();
            void Print();
    private:
            int year, month, day;
};
```

```
//tdate.cpp
#include<iostream.h>
#include”tdate.h”
void TDate::SetDate(int y, int m, int d){
            year=y;
            month=m;
            day=d;
}
int TDate::IsLeapYear(){
            return(year%4==0 && year%100!=0) || (year%400==0);
}
void TDate::Print(){
            cout<<year<<‘,’<<month<<‘,’<<day<<endl;
}
```

引入const成员函数

- 声明：

  <类型标志符> 函数名（参数表）const;

- 说明：

  （1）const是函数类型的一部分，在实现部分也要带该关键字。

  （2）const关键字可以用于对重载函数的区分。

  （3）把整个函数修饰为const，意思是“函数体内不能对成员数据做任何改动。

  （4）常成员函数不能更新类的数据成员，也不能调用该类中没有用const修饰的成员函数，只能调用常成员函数。

  （5）const用在成员函数后 主要是针对类的const 对象。如果声明类的一个const实例，那么它就只能调用有const修饰的函数。

const对象

- 用const修饰的对象叫对象常量，其格式如下：

<类名> const <对象名>

或者

const <类名> <对象名>

- 声明为常对象的同时必须被初始化，并从此不能改写对象的数据成员。

7.1.4 构造函数

- 构造函数实现类对象的初始化过程。
- 默认构造函数(default constructor)
  - 系统隐式定义的一个无参的构造函数，在类没有显示定义任何构造函数式用于类对象初始化。
- 构造函数是一种特殊的成员函数，与其他成员函数不同，不需要用户来调用它，而是在建立对象时自动执行。

构造函数的定义

- 构造函数的名字必须与类名同名，而不能由用户任意命名，以便编译系统能识别它并把它作为构造函数处理。
- 构造函数不具有任何类型，不返回任何值，即使void也不可以。
- 构造函数可以重载。
- 构造函数的功能是由用户定义的，用户根据初始化的要求设计函数体和函数参数。
- 构造函数被声明为公有成员函数。

```
class Time {
    public :
            Time( )   {
                    hour=0;
                    minute=0;
                    sec=0;
            }
            Time(int hr, int min, int sec)   {
                    hour=hr;
                    minute=min;
                    sec=sec;
            }
    private :
            int hour;
            int minute;
            int sec;
    };
```

构造函数的调用

- 如果一个类对象是另一个类的数据成员，则在那个类对象创建时所调用的构造函数中，对该成员对象自动调用其构造函数。
- 类的对象初始化时，构造函数的调用顺序是：按照对象成员在类中声明的顺序依次调用其构造函数，最后执行其自己的构造函数的函数体。
- 每个类只负责初始化它自己的对象。

```
class TutorPair{
public:
  TutorPair()
  {
    cout <<"constructing tutorpair.\n";
    noMeetings=0;
  }
protected:
  Student student;
  Teacher teacher;
  int noMeetings;    //会晤次数
};
```

```
void main()
{
  TutorPair tp;
  cout <<"back in main.\n";
}
```

运行结果：

```
constructing student.
constructing teacher.
constructing tutorpair.
back in main.
```

对象的拷贝

- 对象在几种情况下会被拷贝
  - 初始化变量
  - 以值的方式传递或返回一个对象

拷贝构造函数

- 拷贝构造函数是一种特殊的构造函数，它在创建对象时，是使用同一类中之前创建的对象来初始化新创建的对象。

- 拷贝构造函数通常用于：
  - 通过使用另一个同类型的对象来初始化新创建的对象，即声明变量时赋值。
  - 复制对象把它作为参数传递给函数。
  - 复制对象，并从函数返回这个对象。

对象的拷贝时构造函数的调用

- 如果在类中没有显式地声明一个拷贝构造函数，那么，编译器将会自动生成一个默认的拷贝构造函数，该构造函数完成对象之间的浅拷贝。

- 浅拷贝：也就是在对象复制时，调用默认的拷贝构造函数完成对象中的数据成员进行简单的赋值。

- 但当类中有指针成员，如果类带有指针变量，并有动态内存分配时，使用系统默认拷贝构造函数会存在风险，则必须显式定义一个拷贝构造函数。这就是“浅拷贝”、“深拷贝”问题。

拷贝构造函数的定义

- 格式：

```
classname (const classname &obj) {
        // 构造函数的主体
}
```

- 为什么构造函数的参数一定要是对象的引用呢？如果不是引用，而是通过传值的方式将实参传递给形参，这中间本身就要经历一次对象的拷贝的过程，而对象拷贝则必须调用拷贝构造函数，如此一来则会形成一个死循环，无解。所以拷贝构造函数的参数必须是对象的引用。

浅拷贝会出现的问题

- 假如类有一个成员变量的指针，char *m_data;
  - 问题一，浅拷贝只是拷贝了指针，使得两个指针指向同一个地址，这样在对象块结束，调用函数析构的时，会造成同一份资源析构2次，即释放同一块内存2次，造成程序崩溃。
  - 问题二，浅拷贝使得obj.m_data和obj1.m_data指向同一块内存，任何一方的变动都会影响到另一方。
  - 问题三，在释放内存的时候，会造成obj1.m_data原有的内存没有被释放

深拷贝

- 在类定义时，显式地声明一个拷贝构造函数，使在对象拷贝时不使用系统默认的拷贝函数，而调用自定义的拷贝函数内的代码，完成资源的申请和赋值，这种拷贝叫深拷贝

- 在拷贝构造函数中深拷贝采用在堆内存中申请新的空间来存储数据

- 深拷贝不但需要对指针进行拷贝，而且要对指针指向的内容进行拷贝，经深拷贝后的指针是指向两个不同地址的指针。

深拷贝与浅拷贝的区别

- 简单一点说，就是如果在对象赋值时要使用深拷贝，就需要在构造函数定义时，定义一个特殊的构造函数，在拷贝时调用这个函数；否则，就是浅拷贝，直接由系统赋值。
- 深拷贝时，会申请对象所需要的所有内存，而浅拷贝时，会只申请部分。例如，在有指针的情况下，浅拷贝只是增加了一个指针变量，指针的值指向已经存在的原有对象的内存，即，两个指针指向同一个内存；而深拷贝就是增加一个指针并且申请一个新的内存，使这个增加的指针指向这个新的内存，并把原有对象指针指向内存的值赋给新的内存，两个指针分别指向两个不同的内存。
- 采用深拷贝的情况下，释放内存的时候就不会出现在浅拷贝时重复释放同一内存的错误！

对象的赋值

- 当使用了赋值运算符时，会发生对象的赋值操作。

```
total = trans; // process the next book

// default assignment for Sales_data is equivalent to:
total.bookNo = trans.bookNo;
total.units_sold = trans.units_s
```

类的析构函数

- 析构函数(destructor)也是一个特殊的成员函数，它的作用与构造函数相反，它的名字是类名的前面加一个“~”符号。
- 当对象的生命期结束时，会自动执行析构函数。
- 析构函数不返回任何值，没有函数类型，也没有函数参数。因此它不能被重载。一个类可以有多个构造函数，但只能有一个析构函数。
- 析构函数的作用并不仅限于释放资源方面，它还可以被用来执行“用户希望在最后一次使用对象之后所执行的任何操作”

```
class Student{
public:
  Student()  {
    cout <<"constructing student.\n";
    semesHours=100;
    gpa=3.5;
  }
  ~Student()  {
    cout <<"destructing student.\n";
  }
//其他公共成员
protected:
  int semesHours;
  float gpa;
};
```

```
class Teacher{
public:
  Teacher()  {
    cout <<"constructing teacher.\n";
  }
  ~Teacher()  {
    cout <<"destructing teacher.\n";
  }
};
```

7.2 访问控制与封装

- 一般形式

构造函数和析构函数必须要放在public下

```
class 类名{
        private :
                私有的数据和成员函数;
        protected:
                受保护的数据和成员函数;
        public :
                公用的数据和成员函数;
};
```

成员访问限定符(member access specifier)

- public
  - 定义在public限定符之后的成员，在整个程序内可访问。
- protected
  - 定义在protected限定符之后的成员，不能被类外访问(这点与私有成员类似)，但可以被派生类的成员函数访问。
- private
  - 定义在private限定符之后的成员，只能被类的成员函数访问，不能被类外访问。

- 注意
  - 在声明类类型时，成员访问限定符声明的次序任意，没有限制。
  - 如果在类的定义中既不指定private，也不指定public，则系统就默认为是私有的。

```
class Sales_data {
    public: // access specifier added
            Sales_data() = default;
            Sales_data(const std::string &s, unsigned n, double p):
                        bookNo(s), units_sold(n), revenue(p*n) { }
            Sales_data(const std::string &s): bookNo(s) { }
            Sales_data(std::istream&);
            std::string isbn() const { return bookNo; }
            Sales_data &combine(const Sales_data&);

    private: // access specifier added
            double avg_price() const
                        { return units_sold ? revenue/units_sold : 0; }
            std::string bookNo;
            unsigned units_sold = 0;
            double revenue = 0.0;
    };
```

7.2.1 友元

- 类可以允许其他类或函数访问它的非公有成员，方法是令其他类或函数成为它的友元(friend)。

- 如果类想把一个函数作为它的友元，只需增加一条以friend关键字开始的函数声明语句。

```
class Sales_data {
    // friend declarations for nonmember Sales_data operations added
    friend Sales_data add(const Sales_data&, const Sales_data&);
    friend std::istream &read(std::istream&, Sales_data&);
    friend std::ostream &print(std::ostream&, const Sales_data&);

    public:
        Sales_data() = default;
        Sales_data(const std::string &s, unsigned n, double p):
        bookNo(s), units_sold(n), revenue(p*n) { }
        Sales_data(const std::string &s): bookNo(s) { }
        Sales_data(std::istream&);
        std::string isbn() const { return bookNo; }
        Sales_data &combine(const Sales_data&);
    private:
        std::string bookNo;
        unsigned units_sold = 0;
        double revenue = 0.0;
};
Sales_data add(const Sales_data&, const Sales_data&);
std::istream &read(std::istream&, Sales_data&);
std::ostream &print(std::ostream&, const Sales_data&);
```

类之间的友元关系

- 如果类指定了友元类，则友元类的成员函数可以访问此类的所有成员。
- 例如：指定Windows_mgr为Screen的友元

```
class Screen {
// Window_mgr members can access the private parts of class
    Screen
friend class Window_mgr;
// ... rest of the Screen class
};
```

友元成员函数

- 除了指定整个类作为友元之外，还可以只把某个类的成员函数声明为友元。声明时，必须明确指出该成员函数的类。
- 例如：声明Windows_mgr的成员函数clear为Screen的友元函数

```
class Screen {
// Window_mgr::clear must have been declared before class
    Screen
friend void Window_mgr::clear(ScreenIndex);
// ... rest of the Screen class
};
```

声明友元时注意的问题

- 友元声明只能在类的内部。最好在类定义开始或结束前集中声明友元。
- 友元声明只是指定了访问权限，如果类的用户要调用友元函数，还需要进行函数声明。
- 友元关系不存在传递性
- 对于重载的函数，友元声明认为是不同的函数，需要根据需要对每一个单独声明。

7.3 类的其他特性

- 类型成员
- 类的成员的类内初始值
- 可变数据成员
- 内联成员函数
- 从成员函数返回*this
- 定义使用类类型

定义一个类型成员

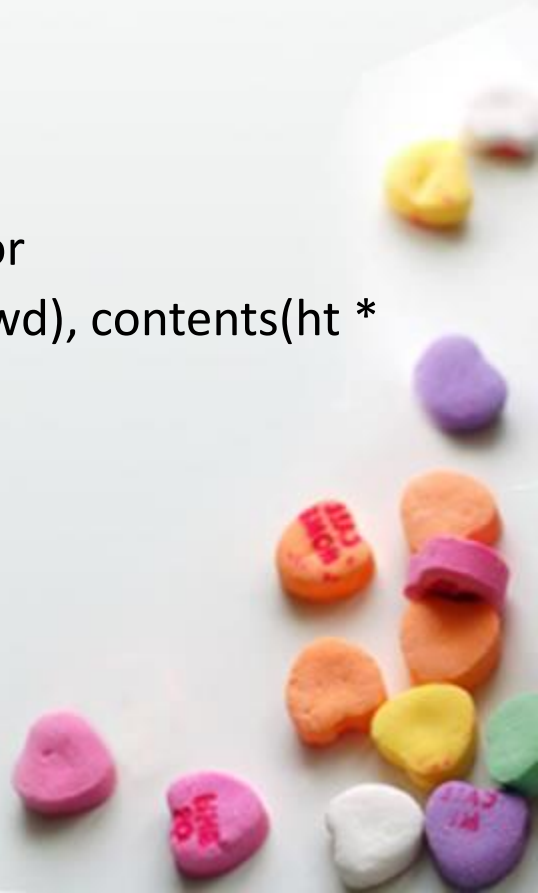
```
class Screen {
    public:
        typedef std::string::size_type pos;
        //using pos=std::string::size_type
    private:
        pos cursor = 0;
        pos height = 0, width = 0;
        std::string contents;
};
```

- 隐藏了数据的类型是String，使用类的用户只知道数据的类型是pos

多个构造函数定义

- 当类中定义多个构造函数时，默认的构造函数需要显示声明才能使用。

```
class Screen {
    public:
        typedef std::string::size_type pos;
        Screen() = default;
        // needed because Screen has another constructor
        Screen(pos ht, pos wd, char c): height(ht), width(wd), contents(ht *
    wd, c) { }
        ......
    private:
        ......
};
```



令成员作为内联函数

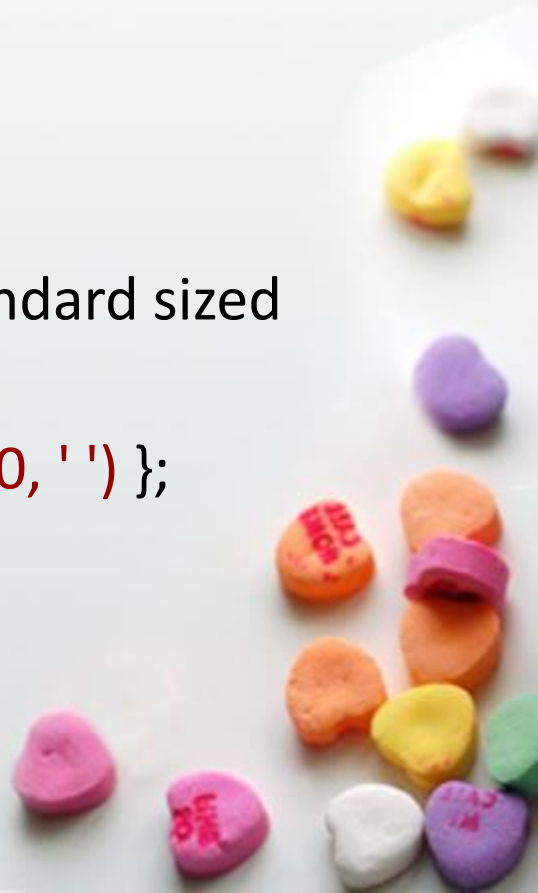
- 定义在类内部的成员函数是自动inline的。
- 也可以在类外部定义成员函数时显示声明inline

```
inline // we can specify inline on the definition
Screen &Screen::move(pos r, pos c){
    pos row = r * width; // compute the row location
    cursor = row + c ;
    // move cursor to the column within that row
    return *this; // return this object as an lvalue
}
```

类数据成员的初始值

- 使用=的初始化形式，或使用花括号括起来直接初始化。

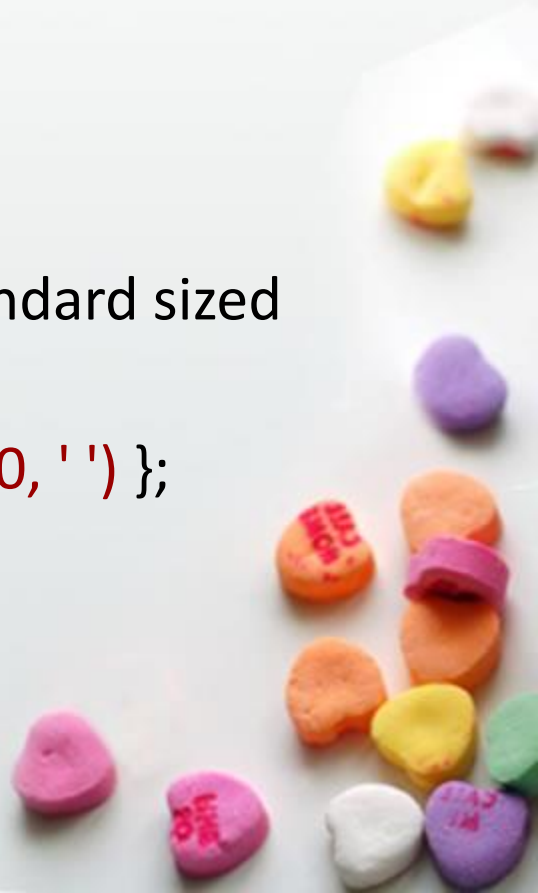
```
class Window_mgr {
    private:
        // Screens this Window_mgr is tracking
        // by default, a Window_mgr has one standard sized
    blank Screen
        std::vector<Screen> screens{Screen(24, 80, ' ') };
};
```



类数据成员的初始值

- 使用=的初始化形式，或使用花括号括起来直接初始化。

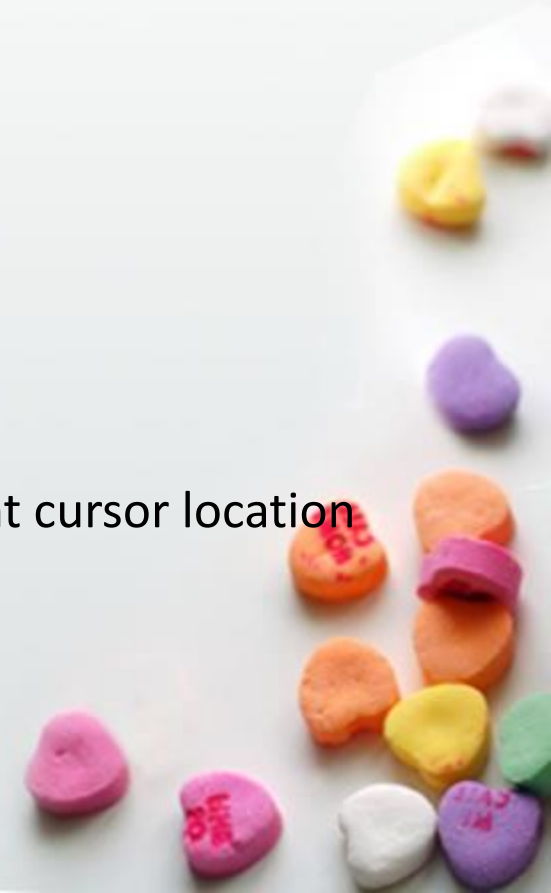
```
class Window_mgr {
    private:
        // Screens this Window_mgr is tracking
        // by default, a Window_mgr has one standard sized
    blank Screen
        std::vector<Screen> screens{Screen(24, 80, ' ') };
};
```



7.3.2 返回*this的成员函数

- this指针是指向本类对象的指针，它的值是当前被调用的成员函数所在的对象的起始地址。

```
class Screen {
    public:
        Screen &set(char);
        // other members as before
};
inline Screen &Screen::set(char c){
    contents[cursor] = c; // set the new value at the current cursor location
    return *this; // return this object as an lvalue
}
```



7.3.3 类类型

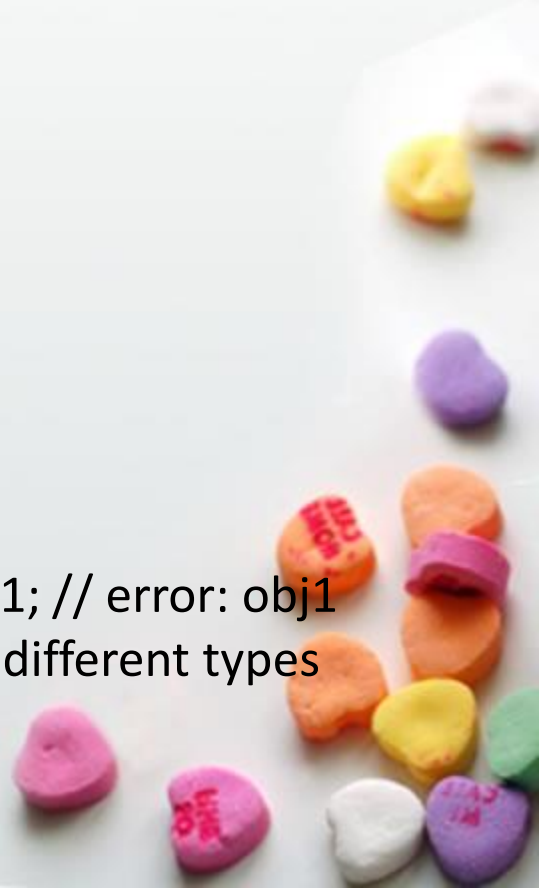
- 类名可以作为类型名使用，直接指向类类型。
- 或者可以把类名跟在class或struct后面。

```
Sales_data item1; // default-initialized object of type Sales_data

class Sales_data item1; // equivalent declaration
```

```
struct First {
    int memi;
    int getMem();
};
struct Second {
    int memi;
    int getMem();
};

First obj1;
Second obj2 = obj1; // error: obj1 and obj2 have different types
```



7.4 类的作用域

- 每个类都会定义它自己的作用域。在类的作用域之外，普通的数据和函数成员只能由对象、引用或指针使用成员访问运算符来访问。

- 看书自学

const数据成员

- 在类的成员定义中，使用修饰符const说明的数据成员称为常数据成员。

- 常数据成员必须初始化，并且不能被修改。常数据成员是通过构造函数的成员初始化列表进行初始化的。

- 格式：

  const 类型名 变量名;

引入const成员函数

- 声明：

  <类型标志符> 函数名（参数表）const;

- 说明：

  （1）const是函数类型的一部分，在实现部分也要带该关键字。

  （2）const关键字可以用于对重载函数的区分。

  （3）把整个函数修饰为const，意思是“函数体内不能对成员数据做任何改动。

  （4）常成员函数不能更新类的数据成员，也不能调用该类中没有用const修饰的成员函数，只能调用常成员函数。

  （5）const用在成员函数后 主要是针对类的const 对象。如果声明类的一个const实例，那么它就只能调用有const修饰的函数。

const对象

- 用const修饰的对象叫对象常量，其格式如下：

<类名> const <对象名>

或者

const <类名> <对象名>

- 声明为常对象的同时必须被初始化，并从此不能改写对象的数据成员。

7.5.1 构造函数初始值列表

- 对于没有在构造函数的初始化值列表中显示初始化成员，则在构造函数体之前执行默认初始化。const或引用成员，必须初始化

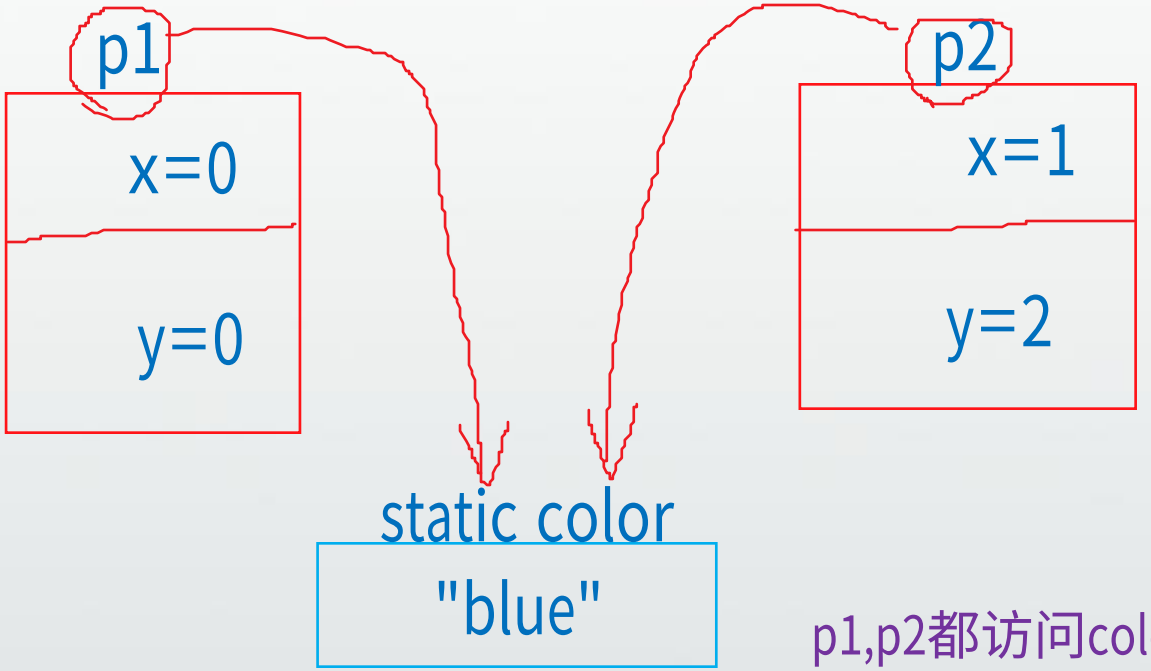
```
class ConstRef {
    public:
        // ok: explicitly initialize reference and const members
        ConstRef::ConstRef(int ii): i(ii), ci(ii), ri(i) { }
    private:
    int i;
    const int ci;
    int &ri;
};
```

成员初始化顺序

- 成员初始化的顺序与它们在类定义中出现的顺序一致。

7.6 类的静态成员

- 包括: 静态数据成员和静态成员函数



p1,p2都访问color，是针对整个类的属性，而不单独属于p1或者p2

静态成员声明

- 静态成员是类一种特殊的成员。它以关键字static开头。

```
class Account {
    public:
        void calculate() { amount += amount * interestRate; }
        static double rate() { return interestRate; }
        static void rate(double);
    private:
        std::string owner;
        double amount;
        static double interestRate;
        static double initRate();
};
```

静态数据成员初始化

- 静态数据成员可以初始化，但只能在类的外部进行初始化。不必在初始化语句中加static。其一般形式为：

  数据类型类名::静态数据成员名=初值；

- 例如：

  类外定义不要丢了数据类型

  ```
  double Account::initRate=0.02;
  //对静态数据成员initRate初始化
  ```

- 注意，不能用参数对静态数据成员初始化。如在Account类中这样定义构造函数是错误的：

  ```
  Account(double r):initRate(r){ }
  //错误，initRate是静态数据成员
  ```

- 如果未对静态数据成员赋初值，则编译系统会自动赋予初值0。

静态数据成员使用

- 两种访问方式：
  - 使用类的对象、引用或指针访问静态数据成员；
  - 使用类名访问。

- 类的静态数据成员的两种访问方式的格式：

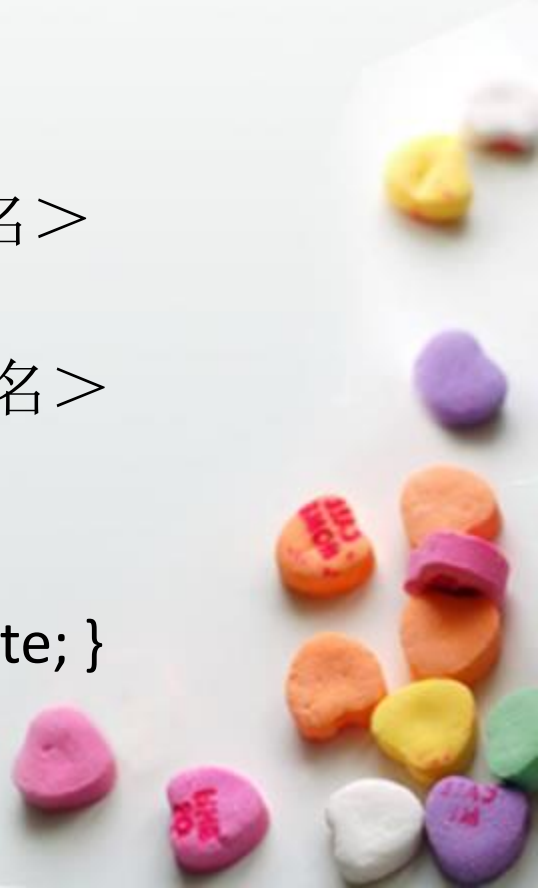
<类对象名>.<静态数据成员名>

或

<类类型名>::<静态数据成员名>

- 成员函数能直接访问静态成员。

```
void calculate() { amount += amount * interestRate; }
```



静态数据成员的特点

- 对于非静态数据成员，每个类对象都有自己的拷贝。而静态数据成员被当作是类的成员。

- 无论这个类的对象被定义了多少个，静态数据成员在程序中也只有一份拷贝，由该类型的所有对象共享访问。也就是说，静态数据成员是该类的所有对象所共有的。对该类的多个对象来说，静态数据成员只分配一次内存，供所有对象共用。所以，静态数据成员的值对每个对象都是一样的，它的值可以更新。

静态成员函数定义和声明

- 定义格式

  static <数据类型> <函数名> (参数列表) {函数体}

或

  <数据类型> <类名>::<函数名> (参数列表) {函数体}

- 声明格式：

  static <数据类型> <函数名> (参数列表) ;

静态成员函数的特点

- 静态成员之间可以相互访问，静态成员函数可以访问静态数据成员和访问静态成员函数；
- 非静态成员函数可以任意地访问静态成员函数和静态数据成员；
- 静态成员函数不能访问非静态成员函数和非静态数据成员；

对象数组

- 数组不仅可以由简单变量组成(例如整型数组的每一个元素都是整型变量)，也可以由对象组成(对象数组的每一个元素都是同类的对象)。

- 1.对象数组的定义

  <类名><数组名>[<大小>]...
- 2.对象数组的赋值
- 3.对象数组的使用

```
//定义Student类
#include<iostream>
#include<string>
using namespace std;
class Student {
  public:
     Student(string , int );//声明构造函数
     void Print();//声明信息输出函数
     string num;
     int score;
};
 Student::Student(string n,int s) {
    num=n;
    score=s;
 }
```

```
//定义和初始化数组
Student stud[5]={
   Student("001",90),
   Student("002",94),
   Student("003",70),
   Student("004",100),
   Student("005",60)
  };
```

C++对象指针

- 在建立对象时，编译系统会为每一个对象分配一定的存储空间，以存放其成员。对象空间的起始地址就是对象的指针。可以定义一个指针变量，用来存放对象的指针。

- 静态分配空间：A a；在栈(stack)上分配空间

- 动态分配空间： A * a= new A；在堆(heap)上分配空间

- 栈上空间自动回收，堆空间需要程序员手动回收

对象指针定义

- 定义指向类的对象的指针变量的一般形式为：

  类名 *指针变量名;

- 例如, 类Time的对象指针定义:

  Time *pt; //定义pt为指向Time类对象的指针变量

  或

  Time t1; //定义t1为Time类对象

  pt=&t1; //将t1的起始地址赋给pt

- 这样，pt就是指向Time类对象的指针变量，它指向对象t1。

通过对象指针访问对象和对象的成员

- 例如：

*pt //pt所指向的对象，即t1

(*pt).hour //pt所指向的对象中的hour成员，即t1.hour

pt->hour //pt所指向的对象中的hour成员，即t1.hour

(*pt).get_time ( ) //调用pt所指向的对象中的get_time函数，即t1.get_time

pt->get_time ( ) //调用pt所指向的对象中的get_time函数，即t1.get_time

动态地分配内存和释放内存

- C++语言中，new、new[]、delete和delete[]操作符通常会被用来动态地分配内存和释放内存。
- new、new[]、delete和delete[]是操作符，而非函数；new和delete也是C++的关键字。
- 操作符new用于动态分配单个空间，而new[]则是用于动态分配一个数组，操作符delete用于释放由new分配的空间，delete[]则用于释放new[]分配的一个数组。
- 为了避免内存泄露，通常new和delete、new[]和delete[]操作符应该成对出现

new 和delete的使用

- new格式：

  datatype *pointer = new datatype;

  – 可以动态的分配一个datatype数据类型大小的空*间。例如：*

  *int *p = new int;*

  – *为p指针分*配了一个int型的空间。new操作符根据请求分配的数据类型来推断所需的空间大小。

- delete格式：

  delete pointer;

  – 释放由new分配的动态存储空间。*例如：*

  delete p;

  – 释放前面new例子为p分配了的一个int型的空间

new[] 和delete[]的使用

- new[] 格式：

  datatype *pointer = new datatype[n];

  – 可以动态分配一个长度为n的数组的空间。

  int *arrp = new int[10];

  – 为arrp指针分配了一个数组的空间，该数组有10个int数组成员，如果分配成功，则arrp指针指向首地址。

- delete[] 格式：

  delete[] pointer;

  – 则用于释放掉由new[]分配的数组空间。例如：

  delete[] p;

  – delete[] 则用于释放掉由new[]分配的数组空间。

补充：组合类

- 所谓类的组合是指：
  - 类中的数据成员不仅可以是基本类型的变量，也可以是其他类的对象。
- 通过类的组合可以在已有的抽象的基础上实现更复杂的抽象。
- 类组合中的难点是关于它的构造函数设计问题。

组合类的构造函数定义

- 原则：不仅要负责对本类中的基本类型成员数据赋初值，也要对对象成员初始化。

- 声明形式：

类名::类名(对象成员所需的形参，本类成员形参)

:对象1(参数)，对象2(参数)，……

{ 本类初始化 }

组合类的构造函数调用

- 构造函数调用顺序：先调用对象数据成员的构造函数（按对象数据成员的声明顺序，先声明者先构造）。然后调用本类构造函数的函数体内代码。（析构函数的调用顺序相反）

- 若调用默认构造函数（即无形参的），则内嵌对象的初始化也将调用相应的默认构造函数。

内容

- 7.1 定义抽象数据类型
- 7.2 访问控制与封装
- 7.3 类的其他特性
- 7.4 类的作用域
- 7.5 构造函数再探
- 7.6 类的静态成员
- 对象数组

Q & A