

今日内容：

- (1) 面向对象的基本概念
- (2) 类的概念（分层）、类的对象的关系、定义一个类
- (3) 类中成员函数的实现（类内和类外）
- (4) 对象的创建和撤销（构造函数和析构函数）、构造函数可以有参数、支持重载、允许按参数缺省方式调用，但是不可以显示方式调用；析构函数没有参数、不支持重载（没有参数），但是析构函数可以显示调用
- (5) 初始化表达式（注意初始化的顺序不是由初始化表中的顺序决定的）
- (6) 复制构造函数（注意缺省复制构造函数带来的问题）
- (7) 特殊数据成员的初始化（常量成员、引用成员、类对象成员、静态成员）
- (8) 特殊成员函数（静态成员函数、`const` 成员函数）、`const` 对象
- (9) 指向对象的指针
- (10) 对象的大小（内存对齐）
- (11) `this` 指针
- (12) 对象数组
- (13) 为对象动态分配内存（`new` 和 `delete`）（注意不能用 `malloc` 和 `free`）

## 第二章 类和对象

### 2.1 面向对象设计的基本理念

面向对象的程序与结构化的程序不同，由 C 编写的结构化的程序是由一个个的函数组成的，而由 C++编写的面向对象的程序是由一个个的对象组成的，对象之间通过消息而相互作用。

在结构化的程序设计中，我们要解决某一个问题，就是要确定这个问题能够分解为哪些函数，数据能够分解为哪些基本的类型，如 `int`、`double` 等。也就是说，思考方式是面向机器结构的，不是面向问题的结构，需要在问题结构和机器结构之间建立联系。

面向对象的程序设计方法的思考方式是面向问题的结构，它认为现实世界是由对象组成的。面向对象的程序设计方法解决某个问题，要确定这个问题是由哪些对象组成的，对象间的相互关系是什么。

- C++的四大特性：**抽象、封装、继承、多态**
- 源文件名由 `*.c` 改成 `*.cpp` (CPP=C Plus Plus=C++)
- 用结构定义变量时，不再需要 `struct` 前缀。

### 2.2 面向对象基本概念

“对象” (object) 是个抽象的概念，现实世界中的任何事物都可以看成是对象，动物、植物、摩托车、汽车等等都是对象，对象之间有很大的差异，如人和汽车，但有的对象间有相似之处，比如摩托车和自行车，它们有共同的特征 (有轮子)，同样的功能 (人的交通工具)，也有不同的特征，如“轮子个数”，“车子重量”等等，基于此，可将“有轮子”，“可更换轮胎”、“能作为人的交通工具”抽象成一个类别 (class)，可称之为“车”类，摩托车和自行车是该类别的对象。

类的提取往往是从两个方面来考虑的，一是**特征** (C++常称为“**属性**”)、另一个是**功能** (C++中常称为“**行为**”)，具备类中定义的“属性”和“行为”的对象都是该类的对象，因此，我们可以说，电动车也是“车”类的对象。

2.1.1 类的概念

C++用类来描述对象，类是对现实世界中相似事物的抽象，同是“双轮车”的摩托车和自行车，有共同点，也有许多不同点。“车”类是对摩托车、自行车、汽车等相同点和不同点的提取与抽象，如所示。

类的定义分为两个部分：数据（相当于属性）和对数据的操作（相当于行为）。从程序设计的观点来说，类就是数据类型，是用户定义的数据类型，对象可以看成某个类的实例（某个类的变量），类和对象的关系与前面介绍的“结构”和“结构体变量”的关系相似，但又有不同，在本章稍后类的定义一节中后具体说明这一问题。

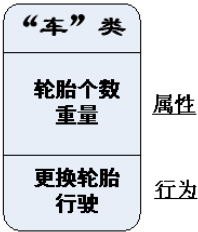


图 8.1 “车”类示意图

	狼	意大利蜜蜂
界Kingdom	动物界Animal	动物界Animal
门Phylum	脊索动物门Chordata	节肢动物门Arthropoda
纲Class	哺乳纲Mammalia	昆虫纲Insecta
目Order	食肉目Carnivora	膜翅目Hymenoptera
科Family	犬科Canidae	蜜蜂科Apidae
属Genus	犬属Canis	蜜蜂属Apis
种Species	狼lupus	意大利蜂mellifera

2.1.2 类是分层的

每一大类中可分成若干小类，也就是说，类是分层的，如图 8.2 所示。可将所有的图形抽象成“图形”类，该类中共同的属性有很多，这里只取“颜色”这个属性，对所有图形而言，都可定义“显示”操作。同时，“图形”类可进一步分为“一维图形”类、“二维图形”类和其他类，根据形状的不同，“一维图形”类可进一步分为“直线”类和“折线”类，“二维图形”类又可分为“正方形”类和“圆”类。下层的类除了“继承”了上层类中定义的属性和行为外，还可增

加新的属性和行为（如“圆”类相比“二维图形”类增加了“圆心”和“半径”属性，增加了“求面积”这一行为），甚至可以在下层类中重新定义上层类已定义的属性和行为（如“直线”类、“折线类”、“正方形”类和“圆”类中都重新定义了“图形”类中已定义的“显示”操作）。

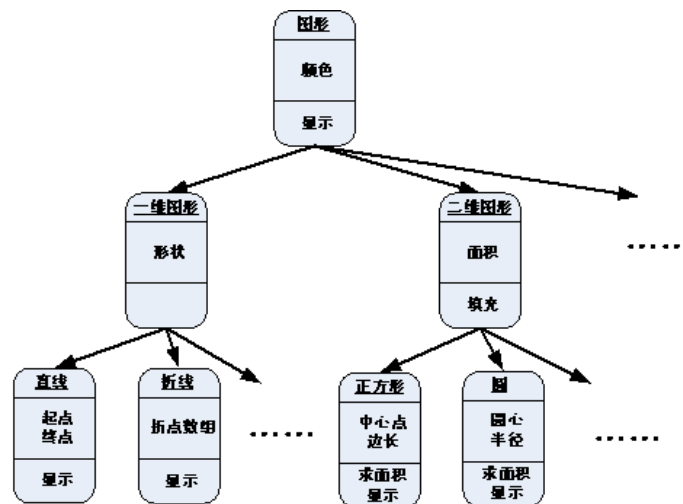


图 8.2 类是分层次的

### 2.1.3 类和对象的关系

对象需要从属性和行为两个方面进行描述，类是对象的封装。类的使用主要有以下几个步骤：

(1) **定义类**：C++中，分别用数据成员和函数成员来表现对象的属性和行为。类的定义强调“**信息隐藏**”，将实现细节和不允许外部随意访问的部分屏蔽起来。因此，在类定义中，需要用 `public` 或 `private` 将类成员区分开（此外，还有 `protected` 型的数据成员，后面课程会有介绍），外界不能访问程序的 `private` 成员，只能访问 `public` 数据成员，对象间的信息传送也只能通过 `public` 成员函数，保证了对象的数据安全。

(2) **实现类**：即进一步定义类的成员函数，使各个成员函数相互配合以实现接口对外提供的功能，类的定义和实现是由类设计者完成的。

(3) **使用类**：通过该类声明一个属于该类的变量（即对象），并调用其接口（即 `public` 型的数据成员或函数成员），这是使用者的工作。

## 2.3 C++类的定义

先来看一下类是如何定义的，对一些通用的问题，前人已经定义好了很多的类，比如微软的 MFC 类库，程序员不必关心其内部细节，只要抱着“**拿来主义**”的态度就好，但对某些特殊问题来说，必须由自己提炼模型，进行类的定义。

### 2.3.1 类定义的基本形式

C++中使用关键字 `class` 定义一个类，其基本形式如下：

```
class 类名
{
private:
    //私有成员变量和函数
protected:
    //保护成员变量和函数
public:
    //公共成员变量和函数
}; //不要漏写了这个分号;
```

`private`, `public`, `protected` 的区别

`private`: 只能由**该类中的函数**、其**友元函数**访问,不能被任何其他访问,该类的对象也不能访问。

`protected`: 可以被**该类中的函数**、**子类的函数(派生类)**、以及其**友元函数**访问,但不能被该类的对象访问

`public`: 可以被**该类中的函数**、**子类的函数**、其**友元函数**访问,也可以由**该类的对象**访问

### 2.3.2 类定义示例

对一台计算机来说，它有如下特征：

属性：品牌、价格。

方法：输出计算机的属性。

如下代码 8-1 实现了 `computer` 类的定义：

```
//文件example801.h
class computer
{
private:    //私有成员列表，这里的private可以省略，因为默认是访问私有数据
    char brand[20];
    float price;    //不能在这里初始化，如float price=0是错误的
public:    //公共成员列表（接口）
    void print();
    void SetBrand(char* sz);
    void SetPrice(float pr);
};
```

注意：关键字 `private` 和 `public` 出现的顺序和次数可以是任意的

### 2.3.3 class 和 struct

class 的定义看上去很像 struct 定义的扩展，事实上，类定义时的关键字 class 完全可以替换成 struct，**class 和 struct 的唯一区别在于：struct 的默认访问方式是 public，而 class 为 private。**

提示：通常使用 class 来定义类，而把 struct 用于只表示数据对象、没有成员函数的类。

## 2.4 C++类的实现

类的实现就是定义其成员函数的过程，类的实现有两种方式：

- (1) 在类定义时同时完成成员函数的定义。
- (2) 在类定义的外部定义其成员函数。

### 2.4.1 在类定义时定义成员函数

成员函数的实现可以在类定义时同时完成  
参照代码：

```
#include <iostream>
using namespace std;

//在类中直接实现成员函数
class student
{
private:
    int num;
    float souce;
protected:
    int protec;
public:
    int count;
    void print()
    {
        cout<<"count = "<<count<<endl;
        cout<<"num = "<<num<<endl;
    }
    void SetCount(int con)
    {
        count = con;
    }
};

//类的成员变量 类的成员函数
//类的实例化：定义一个类的变量（对象）
int main()
{
    student stu1; //要有初始化的工作

    return 0;
}
```

## 2.4.2 在类定义的外部定义成员函数

在类定义的外部定义成员函数时，应使用**作用域操作符**（::）来标识函数所属的类，即有如下形式：

返回类型 类名::成员函数名(参数列表)

```
{  
    函数体  
}
```

其中，返回类型、成员函数名和参数列表必须与类定义时的函数原型一致。

```
#include <iostream>  
using namespace std;  
  
class student  
{  
private:  
    int num;  
    float souce;  
protected:  
    int protec;  
public:  
    int count;  
    void print();  
    void SetCount(int con);  
};  
//在类外实现类的成员函数  
void student::print()  
{  
    cout<<"count = "<<count<<endl;  
    cout<<"num = "<<num<<endl;  
    cout<<"protec"<<protec<<endl;  
}  
void student::SetCount(int con)  
{  
    count = con;  
    protec = 101;  
}  
  
//类的成员变量 类的成员函数  
//类的实例化：定义一个类的变量（对象）  
int main()  
{  
    student stu1;        //要有初始化的工作  
  
    stu1.count = 100;  
    stu1.print();  
    stu1.SetCount(200);  
    stu1.print();  
    return 0;  
}
```

## 2.5 C++类的使用

定义了一个类之后，便可以如同用 int、double 等类型符声明简单变量一样，创建该类的对象，称为类的实例化。由此看来，**类的定义实际上是定义了一种类型，类不接收或存储具体的值，只作为生成具体对象的“蓝图”，只有将类实例化，创建对象（声明类的变量）后，系统才为对象分配存储空间。**

### 2.5.1 对象的作用域、可见域和生存期

对象的作用域、可见域和生存期与普通变量，如 int 型变量的作用域、可见域和生存期并无不同，**对象同样有局部、全局和类内（稍后将对对象成员进行介绍）之分**，对于在代码块中声明的局部对象，在代码块执行结束退出时，对象会被自动撤销，对应的内存会自动释放（当然，如果对象的成员函数中使用了 new 或 malloc 申请了动态内存，却没有使用 delete 或 free 命令释放，对象撤销时，这部分动态内存不会自动释放，造成内存泄露）。

**跟踪调试**，查看同一个类的不同对象的成员变量和成员函数在内存中的地址分配情况。结论：**成员变量占据不同的内存区域(堆、栈)；成员函数共用同一内存区域(代码段)。**

new 和 delete 类似 malloc 和 free 动态申请和释放内存。

类名 \* 指针 = new 类名；

delete 指针 （释放空间）

**每一个对象拥有自己的数据成员，但同一类的对象是共用成员函数的**，成员函数怎么区别不同的对象： this 指针，对象调用成员函数是将对象的地址传递给 this 指针，在成员函数中通过 this 指针访问不同对象的成员。



```

#include <iostream>
using namespace std;

class computer //类定义
{
private:
    char brand[20];
    float price;
    void print_num();
public:
    int num;
    void print();
    void SetBrand(char * sz);
    void SetPrice(float pr);
};

void computer::print_num()
{
    cout<<"库存: "<<num<<endl;
}

void computer::print(/*Simple * this*/) //成员函数的实现
{
    //print_num(); //在这里可以调用私有函数
    cout << "品牌: " << this->brand << endl;
    cout << "价格: " << this->price << endl;
}

void computer::SetBrand(char * sz)
{
    strcpy(brand, sz); //字符串复制
}

void computer::SetPrice(float pr /*Simple * this*/)
{
    //this->price = pr;
    price = pr;
}

int main() //主函数
{
    computer com1; //声明了computer类对象（或说类变量）com1

    com1.SetBrand("Dell"); //调用public成员函数SetBrand设置品牌brand
    com1.SetPrice(6000); //调用public成员函数SetPrice设置品牌price
    // com1.price = 6000; //在类外不能直接访问私有数据成员
    com1.num = 101;
    // com1.print_num(); //在类外不能直接访问私有成员函数
    com1.print(); //信息输出

    computer * com2 = new computer; //为computer类开辟空间返回指针

    com2->SetBrand("Lenovo");
    com2->SetPrice(5000);
    com2->num = 100;
    com2->print();

    delete com2; //释放开辟的空间
    return 0;
}

```

## 2.6 对象的创建和撤销

在上面的代码中，通过自定义的公共成员函数 SetBrand 和 SetPrice 实现数据成员的初始化，实际上，C++为类提供了两种特殊的成员函数来完成同样的工作：

1. 一是构造函数，在对象创建时自动调用，用以完成对象成员变量等的初始化及其他操作（如为指针成员动态申请内存空间等）；**如果程序员没有显式的定义它，系统会提供一个默认的构造函数。**
2. 另一个是析构函数，在对象撤销时自动调用，用以执行一些清理任务，如释放成员函数中动态申请的内存等。**如果程序员没有显式的定义它，系统也会提供一个默认的析构函数。**

### 2.6.1 构造函数的作用

当对象被创建时，构造函数自动被调用。构造函数有一些独特的地方：**函数的名字与类名相同，没有返回类型和返回值，即使是 void 也不能有**。其主要工作有：

- （1）给对象一个标识符。
- （2）为对象数据成员开辟内存空间。
- （3）完成对象数据成员的初始化（函数体内的工作，由程序员完成）。

上述 3 点也说明了构造函数的执行顺序，在执行函数体之前，构造函数已经为对象的数据成员开辟了内存空间，这时，在函数体内对数据成员的初始化便是顺理成章了。

下边代码给出了 point 类的显式构造函数

```
class point
{
private:
    int xPos;
    int yPos;
public:
    point();
};
point::point()
{
    xPos = 0;
    yPos = 0;
}
```

## 2.6.2 构造函数可以有参数

编译器自动生成的缺省构造函数是无参的，实际上，构造函数可以接收参数，在对象创建时提供更大的自由度，如下边的代码。

一旦用户定义了构造函数，系统便不再提供默认构造函数。

```
#include <iostream>

using namespace std;

class point    //point 类定义，在定义同时实现其成员函数
{
private: //私有成员，分别代表 x 轴和 y 轴坐标
    int xPos;
    int yPos;
public:
    point(int x, int y) //有参构造函数
    {
        cout << "对象创建时构造函数被自动调用" << endl;
        xPos = x;
        yPos = y;
    }
    void print() //输出信息
    {
        cout << "xPos: " << xPos << ", yPos: " << yPos << endl;
    }
};

int main()
{
    // point pt0;    //错误的调用，因为我们已经显示的定义了一个带
    参数的构造函数

    // pt0.print();    //输出 pt0 的信息

    point pt1(3, 4); //调用有参构造函数声明 point 类变量（类对象）
```

```

    pt1.print();    //输出 pt1 的信息
    return 0;
}

```

### 2.6.3 构造函数支持重载

一旦程序员为一个类定义了构造函数，编译器便不会为类自动生成缺省构造函数，因此，如果还想使用无参的构造函数，如“point pt0;”的形式必须在类定义中显式定义一个无参构造函数。这样，构造函数就会出现两个，会不会有问题呢？不会，构造函数支持重载，在创建对象时，根据传递的具体参数决定采用哪个构造函数。

例程为：

---

```

//构造函数和析构函数
#include <iostream>
using namespace std;

class point
{
private:
    int xPos;
    int yPos;
public:
    point();           //声明构造函数，它的名字与类名相同，没有返回类型
    point(int x,int y); //重载构造函数
    ~point();          //声明析构函数
    void print();      //行为函数
};

point::point()        //无参构造函数
{
    xPos = 0;
    yPos = 0;
    cout<<"point1:在创建对象时被调用"<<endl;
}

point::point(int x,int y) //有参构造函数
{
    xPos = x;
    yPos = y;
    cout<<"point2:在创建对象时被调用"<<endl;
}

point::~~point()      //析构函数
{
    cout<<"在对象退出时被调用"<<endl;
}

```

```

void point::print()
{
    cout<<"xPos = "<<xPos<<"yPos = "<<yPos<<endl;
}
int main()
{
    point pin1;
    point pin2(3,11);
    cout<<"测试构造函数执行时机"<<endl;
    pin1.print();
    pin2.print();

    cout<<"测试析构函数执行时机"<<endl;
    return 0;
}

```

执行结果：

```

point1:在创建对象时被调用
point2:在创建对象时被调用
测试构造函数执行时机
xPos = 0yPos = 0
xPos = 3yPos = 11
测试析构函数执行时机
在对象退出时被调用
在对象退出时被调用
Press any key to continue

```

## 2.6.4 构造函数允许按参数缺省方式调用

看代码：

```

//构造函数和析构函数
#include <iostream>
using namespace std;

class point
{
private:
    int xPos;
    int yPos;
public:
    point(int x = 100,int y = 100);
    void print();
};
point::point(int x,int y)
{
    xPos = x;
    yPos = y;
    cout<<"在创建对象时被调用"<<endl;
}
void point::print()
{
    cout<<"xPos = "<<xPos<<"yPos = "<<yPos<<endl;
}

```

```

int main()
{
    point pin1;           //两个值都采用默认值
    pin1.print();

    point pin2(3,11);    //两个参数都不采用默认值
    pin2.print();

    point pin3(3);       //第一个参数采用默认值
    pin3.print();

    /*
    point pin4(,11);    //第二个参数采用默认值
    pin4.print();      */

    return 0;
}

```

```

在创建对象时被调用
xPos = 100 yPos = 100
在创建对象时被调用
xPos = 3 yPos = 11
在创建对象时被调用
xPos = 3 yPos = 100
Press any key to continue

```

运行结果：

## 2.6.5 初始化表达式

除了在构造函数体内初始化数据成员外，还可以通过**成员初始化表达式**来完成。成员初始化表可用于初始化类的任意数据成员（注意：后面要介绍的 static 数据成员除外），该表达式由逗号分隔的数据成员表组成，初值放在一对圆括号中。只要将成员初始化表达式放在构造函数的头和体之间，并用冒号将其与构造函数的头分隔开，便可实现数据成员表中元素的初始化

```

point(int x,int y)
{
    cout<<"有参构造函数的调用"<<endl;
    xPos=x;
    yPos=y;
}
//等价于：
point(int x,int y):xPos(x),yPos(y)
{
    cout<<"有参构造函数的调用"<<endl;
}

```

那么在执行这条语句的时候先执行初始化成员初始化表达式，然后再进入函

数执行。

例程：

```
#include <iostream>
using namespace std;

class point
{
private:
    int xPos;
    int yPos;
public:
    point(int x,int y);
    ~point();
    void print();
};

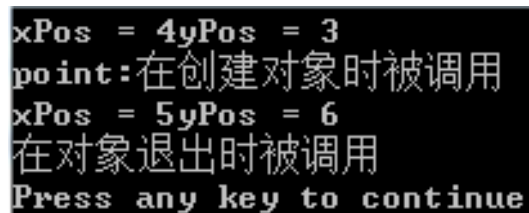
point::point(int x,int y):yPos(3),xPos(4)
{
    cout<<"xPos = "<<xPos<<"yPos = "<<yPos<<endl;
    xPos = x;
    yPos = y;
    cout<<"point:在创建对象时被调用"<<endl;
}

point::~~point()
{
    cout<<"在对象退出时被调用"<<endl;
}

void point::print()
{
    cout<<"xPos = "<<xPos<<"yPos = "<<yPos<<endl;
}

int main()
{
    point pin2(5,6);
    pin2.print();
    return 0;
}
```

执行结果：

A screenshot of a terminal window showing the output of the C++ program. The text is as follows:  
xPos = 4yPos = 3  
point:在创建对象时被调用  
xPos = 5yPos = 6  
在对象退出时被调用  
Press any key to continue

每个成员在初始化表中只能出现一次，初始化的顺序不是由成员变量在初始化表中的顺序决定的，而是由成员变量在类中被申明时的顺序决定的。理解这一点有助于避免意想不到的错误。

例程：

```

#include <iostream>
using namespace std;

class point
{
private:
    int yPos;    //先定义
    int xPos;    //后定义
public:
    point(int x):xPos(x), yPos(xPos) //初始化表取决于成员声明的顺序
    {
    }
    void print()
    {
        cout << "xPos: " << xPos << ", yPos: " << yPos << endl;
    }
};

int main()
{
    point pt1(3);    //调用有参构造函数声明变量pt1
    pt1.print();
    return 0;
}

```

执行结果：

```

xPos: 3, yPos: -858993460
Press any key to continue

```

如果更换一下这个：

```

private:
    int xPos;    //先定义
    int yPos;    //后定义

```

则执行结果是：

```

xPos: 3, yPos: 3
Press any key to continue

```

## 2.6.6 析构函数

构造函数在创建对象时被系统调用，而析构函数在对象被撤销时被自动调用，相比构造函数，析构函数要简单的多。析构函数有如下特点：

- (1) 与类同名，之前冠以波浪号，以区别于构造函数。
- (2) 析构函数没有返回类型，也不能指定参数，因此，析构函数只能有一个，不能被重载。
- (3) 对象超出其作用域被销毁时，析构函数会被自动调用。

如果用户没有显式地定义析构函数，编译器将为类生成“缺省析构函数”，



缺省析构函数是个空的函数体，只清除类的数据成员所占据的空间，但对类的成员变量通过 new 和 malloc 动态申请的内存无能为力，因此，对于动态申请的内存，应在类的析构函数中通过 delete 或 free 进行释放，这样能有效避免对象撤销造成的内存泄漏。

例程：

```
#include <iostream>
#include <cstring>
using namespace std;
class computer
{
private:
    char * brand;           //指针成员(电脑名指针)
    float price;           //电脑单价
public:
    computer(const char * sz, float p)    //构造函数
    {
        brand = new char[strlen(sz) + 1]; //对象创建时为brand分配一块动态内存
        strcpy(brand, sz); //字符串复制
        price = p;
    }
    ~computer()                //析构函数（不能有返回值和参数）
    {
        delete[] brand; //对象别撤销时，释放内存，避免泄露（注意要加[]）
        brand = NULL;
        cout << "清理现场" << endl;
    }
    void print() //信息输出
    {
        cout << "品牌: " << brand << endl;
        cout << "价格: " << price << endl;
    }
};
int main()
{
    computer comp("Dell", 7000); //调用构造函数声明computer变量comp
    comp.print();                //信息输出
    return 0;
}
```

## 2.6.7 显式调用析构函数

程序员不能显式调用构造函数，但却可以调用析构函数控制对象的撤销，释放对象所占据的内存空间，以更高效地利用内存，如：

如果把上边的主函数更换成这个：

```
int main()
{
    computer comp("Dell", 7000);
    comp.print();
    comp.~computer();           //显式调用析构函数，comp被撤销
    return 0;
}
```

则执行结果是：

```
品牌： Dell
价格： 7000
清理现场
清理现场
Press any key to continue
```

注意这里 comp 对象释放时析构函数也被隐式调用了一次

虽然可以显式调用析构函数，但不推荐这样做，因为可能带来重复释放指针类型变量所指向的内存等问题。

## 2.7 复制构造函数

C++中经常使用一个常量或变量初始化另一个变量，例如：

```
double x=5.0;
```

```
double y=x;
```

使用类创建对象时，构造函数被自动调用以完成对象的初始化，那么能否像简单变量的初始化一样，直接用一个对象来初始化另一个对象呢？答案是肯定的，以 point 类为例：

```
point pt1(2,3);
```

```
point pt2=pt1;
```

后一个语句也可写成：

```
point pt2(pt1);
```

上述语句用 pt1 初始化 pt2，相当于将 pt1 中每个数据成员的值复制到 pt2 中，这是表面现象。实际上，系统调用了一个复制构造函数。如果类定义中没有显式定义该复制构造函数时，编译器会隐式定义一个缺省的复制构造函数，**它是一个 inline、public 的成员函数**，其原型形式为：

```
point::point (const point &);
```

## 2.7.1 复制构造函数调用机制

复制构造函数的调用示例：

- (1) point pt1(3,4); //默认构造函数
- (2) point pt2(pt1); //复制构造函数
- (3) point pt3 = pt1; //复制构造函数

例程：

```
#include <iostream>
using namespace std;

class point //定义一个点类
{
private:
    int xPos;
    int yPos;
public:
    point(int x,int y); //构造函数
    point(const point & pt); //复制构造函数
    ~point(); //析构函数
    void print();
};

point::point(int x,int y):yPos(y),xPos(x)
{
    xPos = x;
    yPos = y;
    cout<<"point:在创建对象时被调用"<<endl;
}

point::~point()
{
    cout<<"在对象退出时被调用"<<endl;
}

point::point(const point & pt)
{
    cout<<"复制构造函数被调用"<<endl;
    xPos = pt.xPos;
    yPos = pt.yPos;
}

void point::print()
{
    cout<<"xPos = "<<xPos<<"yPos = "<<yPos<<endl;
}

void fun_1(point pin)
{
    cout<<"fun_1: pin:";
    pin.print();
}

point fun_2()
{
    point pin(10,11);
    cout<<"fun_2: pin:";
    pin.print();
    return pin;
}

int main()
{
    point pin1(5,6);
    point pin2 = pin1;
    pin1.print();
    pin2.print();

    return 0;
}
```

执行结果：

```
point:在创建对象时被调用
复制构造函数被调用
xPos = 5yPos = 6
xPos = 5yPos = 6
在对象退出时被调用
在对象退出时被调用
Press any key to continue
```

如果把主函数改成：

```

int main()
{
    point pin1(5,6);
    point pin2 = pin1;
    fun_1(pin1);
    cout<<"test end fun_1!!!!!"<<endl;
    pin2 = fun_2();
    cout<<"test end fun_2!!!!!"<<endl;
    pin2.print();

    return 0;
}

```

则运行结果是：

```

point:在创建对象时被调用
复制构造函数被调用
复制构造函数被调用
fun_1: pin:xPos = 5yPos = 6
在对象退出时被调用
test end fun_1!!!!!
point:在创建对象时被调用
fun_2: pin:xPos = 10yPos = 11
复制构造函数被调用
在对象退出时被调用
在对象退出时被调用
test end fun_2!!!!!
xPos = 10yPos = 11
在对象退出时被调用
在对象退出时被调用
Press any key to continue

```

（注意：在函数调用和函数返回时也会调用复制构造函数，而且执行完就调用析构函数）

## 2.7.2 缺省复制构造函数带来的问题

缺省的复制构造函数并非万金油，在一些情况下，必须由程序员显式定义缺省复制构造函数，先来看一段错误代码示例（使用缺省的赋值构造函数）

```

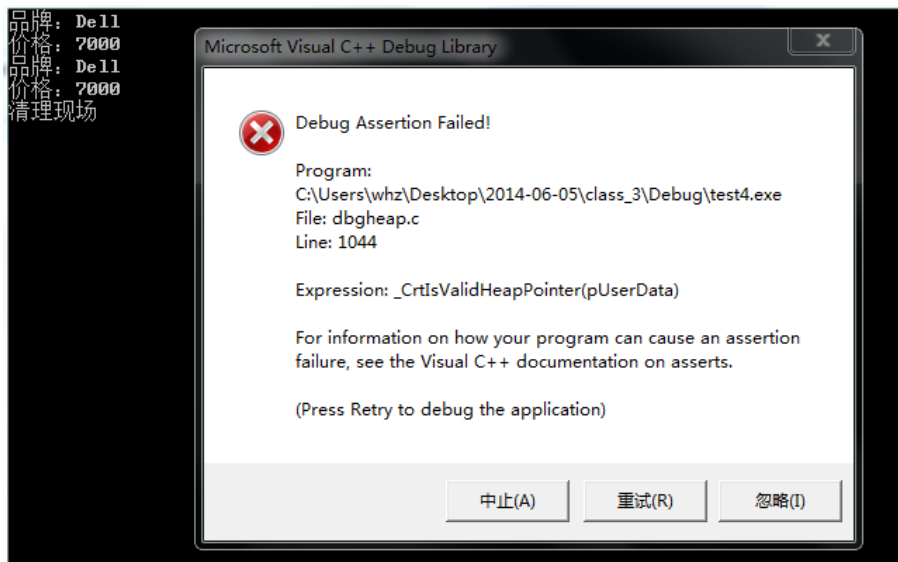
#include <iostream>
using namespace std;

class computer
{
private:
    char * brand;    //电脑名
    float price;     //单价
public:
    computer(const char * sz, float p)    //构造函数
    {
        brand = new char[strlen(sz) + 1]; //构造函数中为brand指针动态分配内存
        strcpy(brand, sz);
        price = p;
    }
    ~computer()    //析构函数
    {
        delete[] brand;    //析构函数中释放申请到的动态内存
        cout << "清理现场" << endl;
    }
    void print()
    {
        cout << "品牌: " << brand << endl;
        cout << "价格: " << price << endl;
    }
};

int main()
{
    computer comp1("Dell", 7000); //声明computer类对象comp, 并初始化
    comp1.print();
    computer comp2(comp1);    //调用缺省的复制构造函数
    comp2.print();
    return 0;
}

```

执行结果:



其中语句 `computer comp2(comp1)` 等价于:

```
comp2.brand = comp1.brand;
```

```
comp2.price = comp1.price;
```

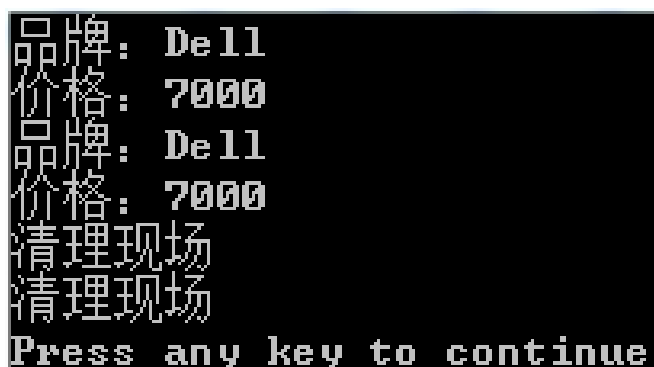
后一句没有问题，但 `comp2.brand = comp1.brand` 却有问题：经过这样赋值后，两个对象的 `brand` 指针都指向了同一块内存，当两个对象释放时，其析构函数都要 `delete[]` 同一内存块，便造成了 2 次 `delete[]`，从而引发了错误。

正确的代码是：

自己写复制构造函数（**牵扯到指针的复制就不能用缺省的复制构造函数**）

```
computer(const computer &cp)    //复制构造函数的实现如下：
{
    brand = new char[strlen(cp.brand) + 1];
    strcpy(brand, cp.brand);
    price = cp.price;
}
```

这样执行的结果就正确了：



```
品牌: Dell
价格: 7000
品牌: Dell
价格: 7000
清理现场
清理现场
Press any key to continue
```

## 2.8 特殊数据成员的初始化

有 4 类特殊的数据成员（**常量成员、引用成员、类对象成员、静态成员**），其初始化及使用方式与前面介绍的普通数据成员有所不同，下面展开具体讨论。

### 2.8.1 `const` 数据成员的初始化

数据成员可以由 `const` 修饰，这样，一经初始化，该数据成员便具有“只读属性”，在程序中无法对其值修改。

事实上，**在构造函数体内或复制构造函数体内初始化 `const` 数据成员是非法的。**  
**`const` 数据成员只能通过成员初始化表达式进行初始化**，见下边错误代码：

```

class point
{
private:
    const int xPos;           //符号常量成员xPos和yPos
    const int yPos;
public:
    point(int x, int y)       //构造函数
    {
        xPos = x;           //错误，无法直接赋值
        yPos = y;
    }
    point(const point & pt) //复制构造函数
    {
        xPos = pt.xPos;
        yPos = pt.yPos;
    }
    void print()
    {
        cout << "xPos: " << xPos << ",yPos: " << yPos << endl;
    }
};

```

正确的代码应该是：

```

class point
{
private:
    const int xPos;
    const int yPos;
public:
    point(int x, int y):xPos(x), yPos(y)           //const数据成员只能在初始化表中进行初始化
    {
    }
    point(const point & pt):xPos(pt.xPos), yPos(pt.yPos) //const数据成员只能在初始化表中进行初始化
    {
    }
    void print()
    {
        cout << "xPos: " << xPos << ",yPos: " << yPos << endl;
    }
};

```

## 2.8.2 引用成员的初始化

对于引用类型的数据成员，同样只能通过成员初始化表达式进行初始化，见下边的代码：

```

#include <iostream>
using namespace std;
class point
{
private:
    int xPos;
    int yPos;
    int &ref1;    //定义引用
    double &ref2; //定义引用
public:
    point(int x, int y, double &z):ref1(xPos), ref2(z) //引用成员的初始化同样要放在初始化表中
    {
        xPos = x;
        yPos = y;
    }
    point(const point &pt):ref1(pt.ref1), ref2(pt.ref2) //复制构造函数也是引用成员的初始化同样要放在初始化表中
    {
        xPos = pt.xPos;
        yPos = pt.yPos;
    }
    void print()
    {
        cout << "xPos: " << xPos << ", yPos: " << yPos;
        cout << ", ref1: " << ref1 << ", ref2: " << ref2 << endl;
    }
    void SetX(int x)
    {
        xPos = x;
    }
};

int main()
{
    double outInt = 5.0;
    point pt1(3, 4, outInt);
    pt1.print();

    point pt2(pt1);
    pt2.print();

    cout << "改变pt1中的x后" << endl;
    pt1.SetX(7);
    pt1.print();
    pt2.print();

    outInt = 6;
    cout << "outInt变化后: " << endl;
    pt1.print();
    pt2.print();

    return 0;
}

```

执行结果:

```

xPos: 3, yPos: 4, ref1: 3, ref2: 5
xPos: 3, yPos: 4, ref1: 3, ref2: 5
改变pt1中的x后
xPos: 7, yPos: 4, ref1: 7, ref2: 5
xPos: 3, yPos: 4, ref1: 7, ref2: 5
outInt变化后:
xPos: 7, yPos: 4, ref1: 7, ref2: 6
xPos: 3, yPos: 4, ref1: 7, ref2: 6
Press any key to continue

```

注意：咱们前边讲过，引用在定义时必须初始化，但是在类中定义引用不能初始化，只能用初始化表达式进行初始化。



### 2.8.3 类对象成员的初始化

类数据成员也可以是另一个类的对象，比如，一个直线类对象中包含两个 point 类对象，在直线类对象创建时可以在初始化列表中初始化两个 point 对象。

下边的代码是对直线类和 point 类的实现：

```
#include <iostream>
using namespace std;
class point    //点类的定义
{
private:
    int xPos;
    int yPos;
public:
    point(int x = 0, int y = 0) //带缺省调用的构造函数
    {
        cout << "点的构造函数被执行" << endl;
        xPos = x;
        yPos = y;
    }
    point(const point & pt)    //复制构造函数
    {
        cout << "点的复制构造函数被执行" << endl;
        xPos = pt.xPos;
        yPos = pt.yPos;
    }
    void print()
    {
        cout << "{ " << xPos << ", " << yPos << " }";
    }
};

class line    //line类的定义
{
private:
    point pt1; //point类对象作为line类成员，此处若写成point pt1(3,4)，错
    point pt2;
public:
    line(int x1, int y1, int x2, int y2):pt1(x1, y1), pt2(x2, y2) //line对象的有参构造函数
    {
        cout << "线的构造函数被执行" << endl;
    }
    line(const line &l1):pt1(l1.pt1), pt2(l1.pt2)    //line对象的复制构造函数
    {
        cout << "线的复制构造函数被执行" << endl;
    }
    void draw()
    {
        pt1.print();
        cout << " to ";
        pt2.print();
        cout << endl;
    }
};

int main()
{
    line l1(1, 2, 3, 4); //调用有参构造函数
    l1.draw();
    line l2(l1);        //调用复制构造函数
    l2.draw();
    return 0;
}
```

执行结果：

```
点的构造函数被执行
点的构造函数被执行
线的构造函数被执行
< 1, 2> to < 3, 4>
点的复制构造函数被执行
点的复制构造函数被执行
线的复制构造函数被执行
< 1, 2> to < 3, 4>
Press any key to continue
```

（对复制构造函数来说，一旦给出了自己定义的形式，编译器便不会提供缺省的复制构造函数，因此，确保自定义的复制构造函数的有效性很重要。因此，在一些必须使用自定义复制构造函数的场合，掌握特殊成员的用法很必要。所举例子中，尽管有些复制构造函数纯属“画蛇添足”，用系统提供的缺省复制构造函数足以实现想要的功能，但还是给出了完整的书写形式，这就是原因所在。）

## 2.8.4 static 数据成员的初始化

C++允许使用 static（静态存储）修饰数据成员，**这样的成员在编译时就被创建并初始化的（与之相比，对象是在运行时被创建的），且其实例只有一个，被所有该类的对象共享**，就像住在同一宿舍里的同学共享一个房间号一样。静态数据成员和下面章节中介绍的静态变量一样，程序执行时，该成员已经存在，一直到程序结束，任何对象都可对其进行访问。

**静态数据成员的初始化必须在类申明之外进行，且不再包含 static 关键字，格式如下：**

类型 类名::变量名 = 初始化表达式; //普通变量

类型 类名::对象名(构造参数); //对象变量

如 float computer::total\_price = 0;

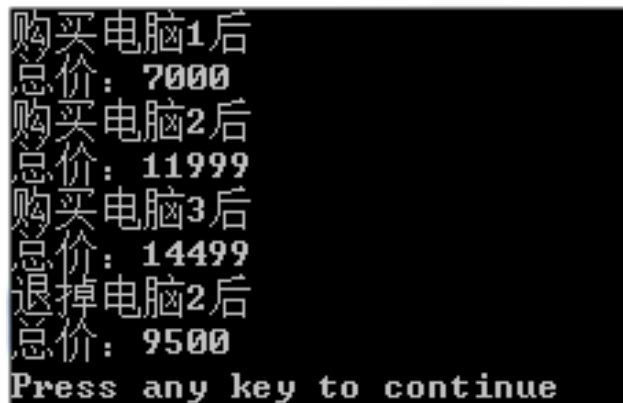
例程代码如下：

```

#include <iostream>
using namespace std;
class computer
{
private:
    float price;           //价格
    static float total_price; //static成员，总价，不依附于某个对象
public:
    computer(const float p) //构造函数，模拟买电脑的操作，对total_price进行累加
    {
        price = p;
        total_price += p;
    }
    ~computer()             //析构函数，模拟退还电脑的操作，从total_price中减去所退电脑的price
    {
        total_price -= price;
    }
    void print()             //输出函数
    {
        cout << "总价: " << total_price << endl;
    }
};
float computer::total_price = 0; //初始化，初始化必须在类申明之外进行
int main()
{
    computer comp1(7000);      //买入电脑1
    cout << "购买电脑1后" << endl;
    comp1.print();
    computer comp2(4999);      //买入电脑2
    cout << "购买电脑2后" << endl;
    comp2.print();
    computer comp3(2500);      //买入电脑3
    cout << "购买电脑3后" << endl;
    comp1.print();             //此处调用comp1.print()、comp2.print()和comp3.print()输出结果相同
    comp2.~computer();         //退掉电脑2
    cout << "退掉电脑2后" << endl;
    comp3.print();
    return 0;
}

```

执行结果:



```

购买电脑1后
总价: 7000
购买电脑2后
总价: 11999
购买电脑3后
总价: 14499
退掉电脑2后
总价: 9500
Press any key to continue

```

## 2.9 特殊函数成员

除了构造函数、复制构造函数和析构函数外，其他成员函数被用来提供特定的功能，一般来说，提供给外部访问的函数称为接口，访问权限为 public，而一些不供外部访问，仅仅作为内部功能实现的函数，访问权限设为 private。本节主要讨论函数成员的一些特殊用法。

## 2.9.1 静态成员函数

成员函数也可以定义成静态的，与静态成员变量一样，系统对每个类只建立一个函数实体，**该实体为该类的所有对象共享**。

静态成员函数体内不能使用非静态的成员变量和非静态的成员函数。

（注意：如果要访问具体对象，必须传递参数）

例程：

```
#include <iostream>
using namespace std;
class computer
{
private:
    char *name;           //名字
    float price;          //价格
    static float total_price; //静态数据成员
public:
    computer(const char *chr, const float p) //构造函数，模拟买电脑操作
    {
        name = new char[strlen(chr) + 1];
        strcpy(name, chr);
        price = p;
        total_price += p;
    }
    ~computer()           //析构函数，模拟退掉电脑的操作
    {
        if (name)
        {
            delete[] name;
            name = NULL;
        }
        total_price -= price;
    }
    static void print_total() //静态成员函数，原则上只能访问静态数据成员
    {
        cout << "总价: " << total_price << endl;
    }
    static void print(computer &com); //静态成员函数print()原型，如果要访问某具体对象，必须传递参数
};

void computer::print(computer &com) //静态成员函数print()实现
{
    cout << "名称" << com.name << endl;
    cout << "价格" << com.price << endl;
}
float computer::total_price = 0; //初始化
int main()
{
    computer comp1("IBM", 7000); //声明类对象comp1，初始化，买入
    computer::print(comp1);      //类名加作用域限定符访问static成员函数，传递参数comp1
    computer::print_total();     //类名加作用域限定符访问static成员函数

    computer comp2("ASUS", 4999); //声明类对象comp2，初始化，买入
    computer::print(comp2);      //类名加作用域限定符访问static成员函数，传递参数comp2
    computer::print_total();

    comp2.~computer();           //析构函数调用，退还电脑
    computer::print_total();

    return 0;
}
```

执行结果：

```
名称IBM
价格7000
总价： 7000
名称ASUS
价格4999
总价： 11999
总价： 7000
Press any key to continue
```

### 2.9.2 const 成员函数

前边已经介绍了 const 在函数中的应用，实际上，const 在类成员函数中还有种特殊的用法，把 const 关键字放在函数的参数表和函数体之间，称为 const 成员函数，其特点有二：

- (1) 只能读取类数据成员，而不能修改之
- (2) 只能调用 const 成员函数，不能调用非 const 成员函数

其基本定义格式为：

(1) 类内定义时：

```
类型 函数名(参数列表) const
{
    函数体
}
```

(2) 类外定义时，共分两步：

类内声明：

```
类型 函数名(参数列表) const;
```

类外定义

```
类型 类名::函数名(参数列表) const
{
    函数体
}
```

例程：

```

class point
{
    int x;
    int y;
public:
    point(int xp = 0, int yp = 0) //构造函数
    {
        x = xp;
        y = yp;
    }
    void print() const           //const成员函数内无法修改数据成员，否则编译器报错
    {
        x = 5;                 //1. 试图修改x将引发编译器报错
        set();                 //2. 试图调用非const函数将引发编译器报错
        cout << "x: " << x << " ,y: " << y << endl;
    }
    void set()                 //将set()定义成const函数就能解决问题
    {
    }
};

```

## 2.10 对象的组织

有了自己定义的类，或者使用别人定义好的类创建对象，其机制与使用 `int` 等创建普通变量几乎完全一致，同样可以 `const` 对象、创建指向对象的指针，创建对象数组，还可使用 `new` 和 `delete` 等创建动态对象。

### 2.10.1 `const` 对象

类对象也可以声明为 `const` 对象，一般来说，能作用于 `const` 对象的成员函数除了构造函数和析构函数，便只有 `const` 成员函数了，因为 `const` 对象只能被创建、撤销以及只读访问，改写是不允许的。

例程：

```

#include <iostream>
using namespace std;

class point //类定义
{
    int x;
    int y;

public:
    point(int xp = 0, int yp = 0) //构造函数
    {
        x = xp;
        y = yp;
    }
    ~point() //析构函数
    {
        x = -1;
    }
    void SetX(int xp) //非const成员函数SetX, 设置x
    {
        x = xp;
    }
    void SetY(int yp) //非const成员函数SetY, 设置y
    {
        y = yp;
    }
    void print() const //const成员函数print, 不能修改x和y
    {
        cout << "x: " << x << " ,y: " << y << endl;
    }
};

int main()
{
    point pt(3, 4); //声明一个普通类变量pt
    pt.SetX(5); //使用pt可调用非const成员函数
    pt.SetY(6);
    pt.print(); //pt也可调用const成员函数

    const point ptC(1, 2); //声明一个const对象(类变量)
    //ptC.SetX(8); //错误, ptC是const对象, 只能调用const成员函数
    //ptC.SetY(9); //错误, ptC是const对象, 只能调用const成员函数
    ptC.~point(); //正确, const对象也能调用非const类型的析构函数
    ptC.print(); //正确, const对象只能调用const成员函数

    return 0;
}

```

执行结果:

```

x: 5 ,y: 6
x: -1 ,y: 2
Press any key to continue

```

## 2.10.2 指向对象的指针

对象占据一定的内存空间，和普通变量一致，C++程序中采用如下形式声明指向对象的指针：

类名\* 指针名 [初始化表达式];

初始化表达式是可选的，既可以通过取地址（&对象名）给指针初始化，也可以通过申请动态内存给指针初始化，或者干脆不初始化（比如置为 NULL），在程序中再对该指针赋值。

指针中存储的是对象所占内存空间的首地址。

定义 point 类如下列代码：

```
#include <iostream>
using namespace std;

class point
{
    int x;
    int y;
public:
    point(int xp = 0, int yp = 0)
    {
        x = xp;
        y = yp;
    }
    void print()
    {
        cout << "x: " << x << ", y: " << y << endl;
    }
};
```

针对上述定义，则下列形式都是合法的：

- (1) point pt; //默认构造函数
- (2) point \*ptr = NULL; //空指针
- (3) point \*ptr = &pt; //取某个对象的地址
- (4) point \*ptr = new point(1, 2.2); //动态分配内存并初始化
- (5) point \*ptr = new point[5]; //动态分配 5 个对象的数组空间

使用指针对象：ptr->print(); (\*ptr).print();都是合法的。

## 2.10.3 对象的大小(sizeof)

对象占据一定大小的内存空间。总的来说，对象在内存中是以结构形式（只包括非 static 数据成员）存储在数据段或堆中，**类对象的大小（sizeof）一般是类中所有非 static 成员的大小之和**。在程序编译期间，就已经为 static 变量在静态存储区域分配了内存空间，并且这块内存存在程序的整个运行期间都存在。**而类中的成员函数存在于代码段中，不管多少个对象都只有一个副本。**

对象的大小同样遵循前边中介绍的 3 条准则，但有一些特殊之处需要强调：

- (1) C++将类中的引用成员当成“指针”来维护，**占据 4 个内存字节**。



(2) 如果类中有虚函数(后面课程将会介绍)时,虚析构造函数除外,还会额外分配一个指针用来指向虚函数表(vtable),因此,这个时候对象的大小还要加 4。

(3) 指针成员和引用成员属于“最宽基本数据类型”的考虑范畴。

//成员变量占据不同的内存区域;

//成员函数共用同一内存区域(代码段)

//所有类的对象公用类的成员函数,

//依靠 this 指针来区别是哪个类的对象调用成员函数

如何计算 class 数据类型的大小:

- 1) 成员里的数据类型相对于 class 首地址的偏移量必须为本数据类型(基本数据类型)的整数倍
- 2) 整个 class 数据类型的大小必须为成员里的最大基本数据类型的整数倍
- 3) Static 静态数据类型不计算
- 4) 函数不计算

例程如下:

```
#include <iostream>
using namespace std;

class cex
{
private:
    int a;           //int型, 在一般系统上占据4个内存字节      4
    char b;          //char型, 占1个内存字节                1 + 3 (3浪费)
    float c;         //单精度浮点型, 占4个内存字节            4 + 4 (4浪费)
    double d;        //double型, 占8个内存字节                8
    short e[5];      //short型数组, 每个元素占2个内存字节      8 + 2
    char & f;         //引用, 当成指针维护                    2 + 4 (2浪费)
    double & g;       //引用, 当成指针维护                    4 + 4 (后4浪费)
    static int h;    //static成员, 公共内存, 不影响单个对象的大小 0
public:
    cex():f(b), g(d) //构造函数, 引用成员必须在初始化表中初始化
    {
    }
    void print()      //成员函数的定义, 普通成员函数不影响对象大小
    {
        cout << "Hello" << endl;
    }
};

int cex::h = 0;      //static成员的初始化
int main()
{
    cex c;
    cout << "sizeof(cex): " << sizeof(cex) << endl; //输出类对象的大小 sizeof(cex) = 48
    return 0;
}
```

执行结果是：

```
sizeof(cex): 48
Press any key to continue
```

#### 2.10.4 this 指针

前面提到，一个类的所有对象共用成员函数代码段，不管有多少个对象，每个成员函数在内存中只有一个版本，那编译器是如何知道是哪个对象在执行操作呢，答案就是“this 指针”。

this 指针是隐含在成员函数内的一种指针，称为**指向本对象的指针**，可以采用诸如“this->数据成员”的方式来存取类数据成员。

举例来说：

```
class Ex
{
private:
    int x;
    int y;
public:
    void Set()
    {
        this->x=1;
        this->y=2;
    }
};
```

#### 2.10.5 对象数组

对象数组和标准类型数组的使用方法并没有什么不同，也有声明、初始化和使用 3 个步骤。

(1) 对象数组的声明：类名 数组名[对象个数];

**这种格式会自动调用无参或所有参数都有缺省值的构造函数**，类定义要符合该要求，否则编译报错。

(2) 对象数组的初始化：可以在声明时初始化。

对于 point(int x, int y) {} 这种没有缺省参数值的构造函数：

```
point pt[2]={point(1,2), point(3,4)}; // #1 正确
point pt[ ]={point(1,2), point(3,4)}; // #2 正确
point pt[5]={point(1,2), point(3,4)}; // #3 错误
point pt[5]; // #4 错误
```

语句#1 和#2 是正确的，但语句#3 错误，因为 pt 的后 3 个元素会自动调用无参的或者所有参数都有缺省值的构造函数，但这样的构造函数不存在。

例程：

```
#include <iostream>
using namespace std;

class point
{
private:
    int x;
    int y;
public:
    // point(int ix = 0, int iy = 0)
    point(int ix, int iy)
    {
        static int iCount = 0;
        iCount++;
        cout << iCount << " .构造函数被调用" << endl;
        x = ix;
        y = iy;
    }
};

int main()
{
    point pt1[2]; //错误：没有合适的构造函数
    point pt2[2] = {point(1,2), point(3,4)}; //正确
    point pt3[] = {point(1,2), point(3,4)}; //正确，自动确定数组的元素个数

    point pt4[5] = {point(1,2), point(3,4)}; //错误：后3个元素会自动调用无参的构造函数，但这样的构造函数不存在

    return 0; //解决方法：给上述构造函数的2个参数定义缺省值
}
```

## 2.10.6 对象链表

对象链表中，节点的初始化需要构造函数来完成，除此之外，对象链表和 C 语言中介绍的链表并无不同。

## 2.11 为对象动态分配内存

和把一个简单变量创建在动态存储区一样，可以用 new 和 delete 为对象分配动态存储区，在复制构造函数一节中已经介绍了为类内的指针成员分配动态内存的相关范例，本节主要讨论如何为对象和对象数组动态分配内存。

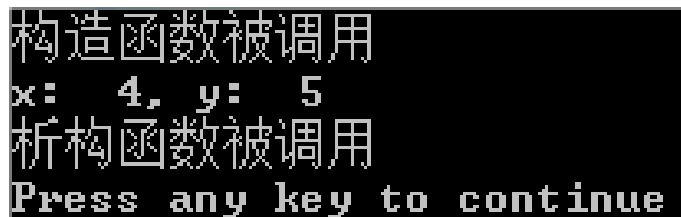
### 2.11.1 使用 new 和 delete 为单个对象分配/释放动态内存

```
#include <iostream>
using namespace std;

class point
{
private:                                //private数据成员列表
    int x;
    int y;
public:
    point(int xp=0,int yp=0)            //构造函数，带缺省参数值
    {
        x=xp;
        y=yp;
        cout<<"构造函数被调用"<<endl;
    }
    ~point()                            //析构函数
    {
        cout<<"析构函数被调用"<<endl;
    }
    void print()                        //成员函数，类内部实现
    {
        cout<<"x: "<<x<<"", y: "<<y<<endl;
    }
};

int main()
{
    point* p = new point(4,5);          //动态申请一块内存，存储point类对象，并将地址赋值给point型指针p
    p->print();                          //使用指针->调用成员函数
    delete p;                           //释放动态申请的内存，防止内存泄露
    p=NULL;                             //养成良好习惯，防止野指针
    return 0;
}
```

执行结果：



```
构造函数被调用
x: 4, y: 5
析构函数被调用
Press any key to continue
```

### 2.11.2 使用 new 和 delete[] 为对象数组分配/释放动态空间

使用 new 为对象数组分配动态空间时，他会调用对象的构造函数，因此，对象要么没有定义任何形式的构造函数（由编译器缺省提供），要么显式定义了一个（且只能有一个）所有参数都有缺省值的构造函数。

例程：

```

#include <iostream>
using namespace std;

class point
{
private:
    int x;
    int y;
public:
    point(int xp = 0, int yp = 0) //构造函数，带缺省参数值
    {
        x = xp;
        y = yp;
        cout << "构造函数被调用" << endl;
    }
    ~point() //析构函数
    {
        cout << "析构函数被调用" << endl;
    }
    void print() //成员函数，类内部实现
    {
        cout << "x: " << x << ", y: " << y << endl;
    }
    void Set(int xp, int yp) //成员函数，类内部实现，用来修改成员x和y
    {
        x = xp;
        y = yp;
    }
};

int main()
{
    point * p = new point[2]; //申请一块动态内存，连续存放两个point对象，将首地址赋值给point指针p
    p[0].print(); //可以将指针当成数组名，使用下标运算符访问对应对象，等价性
    p[1].Set(4, 5); //调用数据元素（对象）的成员函数Set
    p[1].print();
    delete[] p; //释放申请的动态内存
    p = NULL; //养成良好习惯，防止野指针
    return 0;
}

```

执行结果：

```

构造函数被调用
构造函数被调用
x: 0, y: 0
x: 4, y: 5
析构函数被调用
析构函数被调用
Press any key to continue

```

### 2.11.3 malloc 和 free 不能为对象动态申请内存

malloc/free 无法满足动态对象的要求，因为 malloc 和 free 无法像 new/delete 及 new/delete[] 那样自动调用对象的构造函数和析构函数。

## 2.12 本章小结

本章讲述了 C++ 语言中面向对象编程的基本概念和方法。

C++ 通过 `class` 关键字可以定义类, 类的成员包括数据成员和函数成员两种。关于类的使用, 大体分为类的定义、类的实现和类的对象的创建 3 个步骤, 其中, 类的定义指明了类的结构, 相当于“蓝图”, 而类的实现相当于“技术图纸”, 根据定义和实现便可以声明一个类的对象。

类中有几个特殊的成员函数, 构造函数、复制构造函数和析构函数。构造函数和复制构造函数用于为类对象开辟所需内存空间, 并初始化各成员变量的值, 而析构函数则是在撤销对象时, 释放其内存空间, 但需要注意的是, **用户通过 `new` 申请的动态内存并不会在对象撤销时被自动释放**, 所以, 应合理搭配 `new` 和 `delete`, 及时释放无用的动态内存。**构造函数不能由用户调用, 但析构函数可以显式调用**。复制构造函数的形参是本类对象的引用, 它是用一个对象来初始化另一个对象。如果编程者没有显式定义构造函数 (包括复制构造函数), C++ 编译器就隐式定义缺省的构造函数。

## 这两天所讲内容复习

- (1) 面向对象的基本概念
- (2) 类的概念（分层）、类的对象的关系、定义一个类
- (3) 类中成员函数的实现（类内和类外）
- (4) 对象的创建和撤销（构造函数和析构函数）、构造函数可以有参数、支持重载、允许按参数缺省方式调用，但是不可以显示方式调用；析构函数没有参数、不支持重载（没有参数），但是析构函数可以显示调用
- (5) 初始化表达式（注意初始化的顺序不是由初始化表中的顺序决定的）
- (6) 复制构造函数（注意缺省复制构造函数带来的问题）
- (7) 特殊成员的初始化（常量成员、引用成员、类对象成员、静态成员）
- (8) 特殊成员函数（静态成员函数、`const` 成员函数）、`const` 对象
- (9) 指向对象的指针
- (10) 对象的大小（内存对齐）
- (11) `this` 指针
- (12) 对象数组
- (13) 为对象动态分配内存（`new` 和 `delete`）（注意不能用 `malloc` 和 `free`）

## 作业

- 1、创建文件 `student.h`，并在文件中声明类
- 2、定义一个类，类中分别有 `private`、`protected`、`public` 类型的成员数据和成员函数，并在成员函数内部和类的外部测试各种成员数据和成员函数的可访问性。



## 答案

//创建文件 student.cpp, 并在文件中定义 SetInfo 和 Display 成员函数的实现。

```
#include <iostream>
```

```
using namespace std;
```

//1、创建文件 student.h, 并在文件中声明类

```
class CStudent
```

```
{
```

```
private:
```

```
    int m_iId;
```

```
    char m_szName[32];
```

```
public:
```

```
    //将学生 ID 和姓名赋值给成员变量
```

```
    void SetInfo(int iId, char * pszName);
```

```
    //打印出成员变量
```

```
    void Display();
```

```
};
```

//将学生 ID 和姓名赋值给成员变量

```
void CStudent::SetInfo(int iId, char * pszName)
```

```
{
```

```
    m_iId = iId;
```

```
    strcpy(m_szName, pszName);
```

```
}
```

//打印出成员变量

```
void CStudent::Display()
```

```
{
```

```
    cout << "the student named " << m_szName << "'s id is " << m_iId <<
```

```
endl;
```

```
}
```

```
//最后在 main.cpp 中使用该类

void main()
{
    //栈中实现
    CStudent stud;
    stud.SetInfo(3, "Tom");
    stud.Display();

    //堆中实现
    CStudent *pStud = new CStudent;
    pStud->SetInfo(0, "Bill");
    pStud->Display();
    (*pStud).Display();

    delete pStud;
    pStud = NULL;
}
```