第一章 类和对象

教学要求、重点与难点

- 教学要求
  - 1、掌握类定义；
  - 2 、掌握对象的定义与使用
  - 3 、掌握构造函数、析构函数和拷贝函数；
  - 4、理解友元、类的作用域
  - 5、掌握对象的生存期；

- 重点与难点
  - 1. 类的定义、对象的定义与使用；
  - 2. 利用类和对象进行程序设计。

内　　容

1　类的定义
2　对象的定义
3　对象的初始化
4　成员函数的特性
5　静态成员
6　友元
7　类的作用域
8　局部类和嵌套类
9　对象的生存期
本章小结
本章作业

回顾：面向过程的设计方法

- 重点:
  - 如何实现细节过程，将数据与函数分开。
- 形式：
  - 主模块+若干个子模块（main()+子函数）。
- 特点：
  - 自顶向下，逐步求精——功能分解。
- 缺点：
  - 效率低，程序的可重用性差。

面向对象的方法

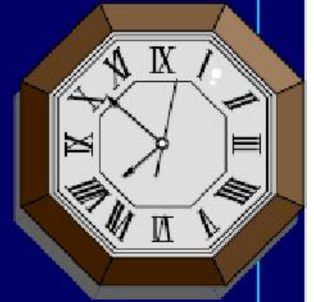
- 目的：
  - 实现软件设计的产业化。
- 观点：
  - 自然界是由实体（对象）所组成。
- 程序设计方法：
  - 使用面向对象的观点来描述模仿并处理现实问题。
- 要求：
  - 高度概括、分类和抽象。

OOP的基本特点：抽象

抽象是对具体对象（问题）进行概括，抽出这一类对象的公共性质并加以描述的过程。

- 先注意问题的本质及描述，其次是实现过程或细节。
- 数据抽象：描述某类对象的属性或状态（对象相互区别的物理量）。
- 代码抽象：描述某类对象的共有的行为特征或具有的功能。
- 抽象的实现：通过类的定义。

抽象实例——钟表



- 数据抽象：

  int Hour， int Minute,
    int Second

- 代码抽象：

  SetTime()，ShowTime()

抽象实例——钟表类

```
class  Watch
{
        public:
            void SetTime(int NewH,  int NewM,
                                    int NewS);
              void ShowTime();
        private:
            int Hour,Minute,Second;
};
```

抽象实例——人

- 数据抽象：

  char *name,char *sex,int age,int id

- 代码抽象：

  生物属性角度：

  GetCloth(), Eat(), Step(),…

  社会属性角度：

  Work(), Promote() ,…

封 装

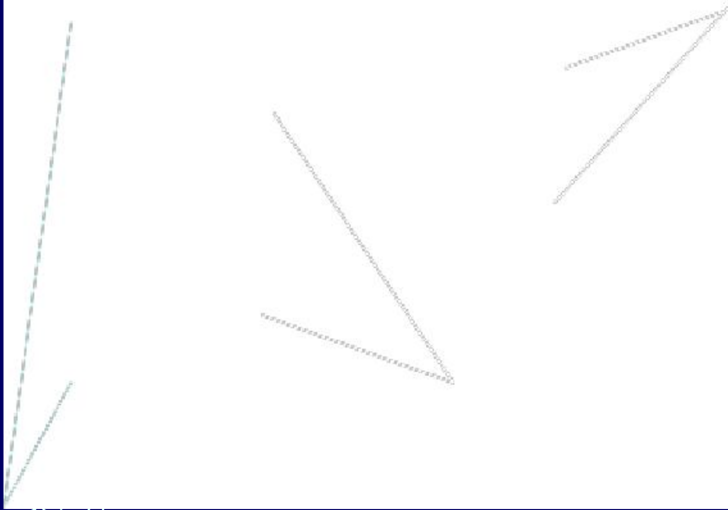
将抽象出的数据成员、代码成员相结合，将它们视为一个整体。

- 目的是增加强安全性和简化编程，使用者不必了解具体的实现细节，而只需要通过外部接口，以特定的访问权限，来使用类的成员。
- 实现封装：类定义中的{}

封 装

- 实例：

class Watch



外部接口

NewH,int NewM,
t NewS);
;
Second;

限

继承与派生

继承与派生是C++中支持层次分类的一种机制，允许程序员在保持原有类特性的基础上，进行更具体的类定义。

实现：定义派生类——*第7章*

多 态 性

- 多态：同一名称，不同的功能实现方式。
- 目的：达到行为标识统一，减少程序中标识符的个数。
- 实现：重载函数和虚函数——*第8章*

§1.1 类的定义

一、什么是类

- 类是具有相同属性和行为的一组对象的集合，它为属于该类的全部对象提供了统一的抽象描述，其内部包括属性和行为两个主要部分。
- 利用类可以实现数据的封装、隐藏、继承与派生。
- 利用类易于编写大型复杂程序，其模块化程度比C中采用函数更高。

§1.1 类的定义

二、类的定义格式

类是一种用户自定义类型，定义形式：

```
class 类名称
{
        public:
                公有成员（外部接口）
        private:
                私有成员
        protected:
                保护型成员
}
```

- 公有类型成员

在关键字public后面声明，它们是类与外部的接口，任何外部函数都可以访问公有类型数据和函数。

- 私有类型成员

在关键字private后面声明，只允许本类中的函数访问，而类外部的任何函数都不能访问。

如果紧跟在类名称的后面声明私有成员，则关键字private可以省略。

- 保护类型

protected与private类似，其差别表现在继承与派生时对派生类的影响不同，第7章讲。

• 类的成员
class Watch
{
public:
void SetTi
NewM,
nt NewS);
void ShowTime( );
private:
int Hour, Minute, Second;
};
成员函数
成员数据或
数据成员

```
void Watch :: SetTime(int NewH, int NewM,
                                          int
   NewS)
{
       Hour=NewH;
       Minute=NewM;
       Second=NewS;
}
void Watch :: ShowTime（ ）
{
       cout<<Hour<<":"<<Minute<<":"<<Second;
}
```

- **数据成员（data members）**
- 与一般的变量定义相同，但需要将它放在类的定义体中。

- **成员函数（member functions）**
- 在类中说明原形，在类外定义函数体实现，并在函数名前使用类名加以限定。也可以直接在类中定义函数体，形成内联成员函数。
- 允许定义重载函数和带缺省形参值的函数。
- 在类外定义函数体时需使用作用域运算符::，用它来标识某个成员函数是属于哪个类的。
- 格式如下：

  <类名>::<函数名>(<参数表>)

三、定义类时应注意事项

1、不可以对类中的数据成员进行初始化

下面的说明是错误的：

```
class Location {
    private:
    int X=0; Y=0;  //错
    .....
}
```

2、在类中说明的任何成员不能使用**extern**、**auto**和**register**关键字进行修饰

3、一般地，在一个类体内，先说明**public**成员，然后再说明**private**成员。不过**public**和**private**出现的次序和次数可以是任意的。每遇到**private**或**public**关键字，就改变其后的成员的访问权限为该关键字所规定的访问权限。

三、定义类时应注意事项

4、类中的数据成员的类型可以是基本类型和构造类型，也可以是其它类，即变量和对象。自身类的对象不能作为该类成员，但自身类的指针或引用是可以的。

5、当一个类的对象作为这个类的成员时，若另一个类的定义在后，需要提前说明。例如：

```
class N;     //提前说明类N
class M
{   public: .....
    private:
        N   n;//n是类N的对象
};
class N   //类N的定义
{
     .....
};
```

§1.2 对象的定义

类的对象是该类的某一特定实体，即类类型的变量。

一、对象的定义格式

<类名> < 对象名表>;

例：

```
Watch myWatch,ArrayWatch[20];
Watch *pWatch, &rWatch=myWatch;
```

§1.2 对象的定义

二、对象成员的表示方法

- 类中成员互访
  - 直接使用成员名
- 类外访问
  - <对象名>.<成员名>

    <对象名>.<成员名(参数表)>

其中<成员名>和<成员名(参数表)>为public 属性的成员

例：

```
myWatch.SetTime(8,30,30);
myWatch.ShowTime();
```

§1.2 对象的定义

二、对象成员的表示方法

- 类中成员互访
  - 直接使用成员名
- 指向对象的指针的成员表示

<(*对象指针名)>.<成员名>
<(*对象指针名)>.<成员名(参数表)>

或者

<对象指针名> -> <成员名>
<对象指针名>-> <成员名(参数表)>

§1.2 对象的定义

```
//例题：类的应用
    #include<iostream.h>
    class  Watch //类的定义
    {
      public:
          void SetTime(int NewH, int NewM, int NewS);
          void ShowTime(  );
      private:
          int Hour, Minute, Second;
    };
```

```
void Watch :: SetTime(int NewH, int NewM, int NewS)
{
        Hour=NewH;
        Minute=NewM;
        Second=NewS;
}
void Watch :: ShowTime（ ）
{
        cout<<Hour<<":"<<Minute<<":"<<Second;
}
void main(void)
{    Watch  myWatch;
      myWatch.SetTime(8,30,30);
      myWatch.ShowTime( );
}
```

- 例：定义一个CDog类

```
class CDog
{
    private:
        char    m_strName[20];
        int     m_nAge;
    public:
        void    Register(const char *Name, int Age);
        void    GetName(char *Name);
        int     GetAge();
        void    Speak();
};
```

- 例：CDog类成员函数的定义

```
#include <string.h>
void CDog:: Register(const char *name, int age)
{
    strcpy(m_strName,name);
    m_nAge = age;
}
void CDog::GetName(char *name)
{
    strcpy(name, m_strName);
}
int CDog:: GetAge()
{
    return m_nAge;
}
void CDog:: Speak()
{
    cout<<"Arf!Arf!"<<endl;
}
```

- 例：Dog对象的使用

```
#include <iostream>
using namespace std;
#include "dog.h"
void main()
{
    char name[20];
    int age;
    CDog dog1;
    cout<<"Enter a dog’ s name,age:";
    cin>>name>>age;
    dog1.Register(name,age);
    dog1.GetName(name);
    cout<<"Dog's name: "<<name<<endl;
    cout<<"Dog's age:  "<<dog1.GetAge()<<endl;
    cout<<"Dog speak:  ";
    dog1.Speak();
    cout<<endl;
}
```

§1.3 对象的初始化

一、构造函数和析构函数

- 构造函数是类的一个特殊成员函数，其函数名与类名相同，其功能是在创建对象时，使用给定的值来将对象初始化。
- 析构函数用于撤销一个对象，完成释放对象成员所占存储空间的工作。

构造函数（Constructor）

格式：

<类名>::<类名> (<参数表>);

- 注意事项：
  - 构造函数必须与类名相同
  - 构造函数没有返回值
  - 其功能是对对象进行初始化，一般由一系列赋值语句构成
  - 在创建对象时由系统自动调用，程序中不能直接调用
  - 构造函数可以重载

析构函数（Destructor）

- 格式为：

  <类名>::~<类名>();

- 注意事项：
  - 函数名与类名相同，只是在前边加“~”符号
  - 析构函数不得返回任何值
  - 析构函数不得带有任何参数
  - 其功能是释放对象成员所占用的存储
  - 在下面两种情况下由系统自动调用析构函数

    定义对象的函数执行结束

    使用delete删除由new运算符创建的对象

例：使用构造函数的CDog类

```
class CDog
{
  private:
        char    m_strName[20];
        int     m_nAge;
  public:
        CDog(const char *name, int age);
        void    GetName(char *Name);
        int     GetAge();
        void    Speak();
  };
```

CDog的构造函数

```
#include <string.h>
CDog::CDog(const char *name, int age)
{
    strcpy(m_strName, name);
    m_nAge = age;
}
void CDog::GetName(char *name)
{
    strcpy(name, m_strName);
}
int CDog:: GetAge()
{
    return m_nAge;
}
void CDog:: Speak()
{
    cout<<"Arf!Arf!"<<endl;
}
```

构造函数举例

```
class Watch
{
   public:
     Watch(int NewH, int NewM, int NewS);//构造函数
    void SetTime(int NewH, int NewM, int NewS);
    void ShowTime（）;
  private:
    int Hour,Minute,Second;
};
```

```
//构造函数的实现：
Watch :: Watch(int NewH, int NewM, int NewS)
{
    Hour=NewH;
    Minute=NewM;
    Second=NewS;
}
//建立对象时构造函数的作用：
void main（ ）
{
    Watch c (0,0,0); //隐含调用构造函数，将初始值作为实参。
    c.ShowTime（ ）;
}
```

构造函数和析构函数举例

```
class Point
{
   public:
       Point(int xx=0,int yy=0){X=xx; Y=yy;}
       Point(Point&  p);
       ~Point（）;
       int GetX（） {return X;}
       int GetY（） {return Y;}
   private:
       int  X,Y;
};
```

析构函数（Destructor）

```
Point::Point (Point&  p)
{
    X=p.X;
    Y=p.Y;
    cout<<"拷贝构造函数被调用"<<endl;
}
Point::~Point（ ）
{
     cout<< “析构函数被调用"<<endl;
}
```

析构函数（Destructor）

```
Point::Point (Point&  p)
{
    X=p.X;
    Y=p.Y;
    cout<<"拷贝构造函数被调用"<<endl;
}
```

二、缺省构造函数和缺省析构函数

1、缺省构造函数

在类中若没有定义构造函数则编译器自动生成一个 不带参数的缺省构造函数，其格式如下：

<类名>::<类名>(){ }

用缺省构造函数对对象初始化时，则将对象的所有数据成员都初始化为零或空。

2、缺省析构函数

在类中若没有定义析构函数则编译器自动生成一个 缺省析构函数，其格式如下：

<类名>:: ~ <类名>(){ }

缺省析构函数是一个空函数。

三、拷贝初始化构造函数

- 拷贝构造函数是一种特殊的构造函数
- 其功能是用一个已知的对象来初始化一个被创建的同类的对象。
- 其形参为本类的对象引用。其格式如下：

```
class 类名
{  public :
       类名（形参）；//构造函数
       类名（类名 &对象名）；  //拷贝构造函数
          ...
};
类名:: 类名（类名 &对象名） //拷贝构造函数的实现
{    函数体    }
```

```
class Point
{
  public:
    Point(int xx=0,int yy=0){X=xx; Y=yy;}
    Point(Point&  p);
    int GetX（） {return X;}
    int GetY（） {return Y;}
  private:
    int  X,Y;
};
Point::Point (Point&  p)
{
    X=p.X;
    Y=p.Y;
    cout<<"拷贝构造函数被调用"<<endl;
}
```

拷贝初始化构造函数举例

- ①当用类的一个对象去初始化该类的另一个对象时，系统自动调用它实现拷贝赋值。

```
void main(void)
{   Point A(1,2);
    Point B(A); //用对象A初始化对象B，拷贝构造函数被调用
    cout<<B.GetX（ ）<<endl;
}
```

拷贝初始化构造函数举例

- ②若函数的形参是类的对象，调用函数时，实参赋值给形参，系统自动调用拷贝构造函数。例如：

```
void fun1(Point p)
{
    cout<<p.GetX（ ）<<endl;
} //调用析构函数
void main（ ）
{   Point A(1,2);
    fun1(A); //函数的形参为类的对象，当调用函数时，拷贝构造函数被调用
}
```

拷贝初始化构造函数举例

- ③当函数的返回值是类的对象时，系统自动调用拷贝构造函数。例如：

```
Point  fun2（ ）
{     Point A(1,2);
      return A;  //函数的返回值是类的对象，函数执行完成返回调用者时，调用拷贝构造函数
}
void main（ ）
{
      Point B;
      B=fun2（ ）;
}
```

三、拷贝初始化构造函数

- 如果程序员没有为类声明拷贝初始化构造函数，则编译器自己生成一个拷贝构造函数。
- 这个构造函数执行的功能是：用作为初始值的对象的每个数据成员的值，初始化将要建立的对象的对应数据成员

拷贝初始化构造函数

- 深拷贝
- 浅拷贝

```
class arraylist
{
    int *data
     int size;
}
```

```
class arraylist
{
        int data[20];
        int size;
}
```

例：顺序表类定义

```
class arrList
{                                   // 顺序表，向量
private:                            // 线性表的取值类型和取值空间
    int *aList ;                    // 私有变量，存储顺序表的实例
    int maxSize;                    // 私有变量，顺序表实例的最大长度
    int curLen;                     // 私有变量，顺序表实例的当前长度
    int position;                   // 私有变量，当前处理位置
public:                             // 顺序表的运算集
    arrList(const int size)     // 创建一个新的顺序表，参数为表实例的最大长度
    arrList(const arrList tList) ; // 拷贝构造函数
    ~arrList() ;                    // 析构函数，用于消除该表实例
    void clear() ;
    int length();                               // 返回此顺序表的当前实际长度
    bool append(const int value);                   // 在表尾添加一个元素value,表的长度增1
    bool insert(const int p, const  int value);          // 在位置p上插入一个元素value，表的长度增1
    bool delete1(const int p);                  // 删除位置p上的元素，表的长度减 1
    bool setValue(const int p, const  int value);        // 用value修改位置p的元素值
    bool getValue(const int p, int & value);             // 把位置p的元素值返回到变量value中
    bool getPos(int & p, const int value);             // 查找值为value的元素，并返回第1次出现的位
      置
    void output();
    void sort();
    void output();
};
```

```
arrList::arrList(const int size)
   {
        maxSize = size;   aList = new int [maxSize];  curLen = position = 0;
   }


   arrList ::~arrList()
   {
      delete [] aList;
      cout<<"析构函数"<<endl;
   }


   void  arrList::clear()
   {
      delete [] aList; curLen = position = 0;
      aList = new int [maxSize];
   }
```

```
arrList::arrList(const arrList  tList)
    {
          maxSize = tList. maxSize;
          aList = new int [maxSize];
          curLen= tList. curLen;
         position= tList. position ;
         for(int i=0;i< curLen; i++)
           aList[i]= tList. aList[i];
}
```

§1.4 成员函数的特性

一、内联函数和外联函数

内联函数：在类体内定义的函数；

外联函数：在类体外定义的函数；

但在外联函数头前面加上关键字inline就成为内联函数

- 为了提高运行时的效率，对于较简单的函数可以声明为内联形式。
- 内联函数体中不要有复杂结构（如循环语句和switch语句）。
- 在类中声明内联成员函数的方式：
  - 将函数体放在类的声明中。
  - 使用inline关键字。

内联成员函数举例(一)

```
class Location
{
 public:
    void Init(int initX,int initY)
    {
      X=initX;
      Y=initY;
    }
    int GetX() {return X;}
    int GetY() {return Y;}
 private:
    int X,Y;
};
```

有两种方法可存储对象。

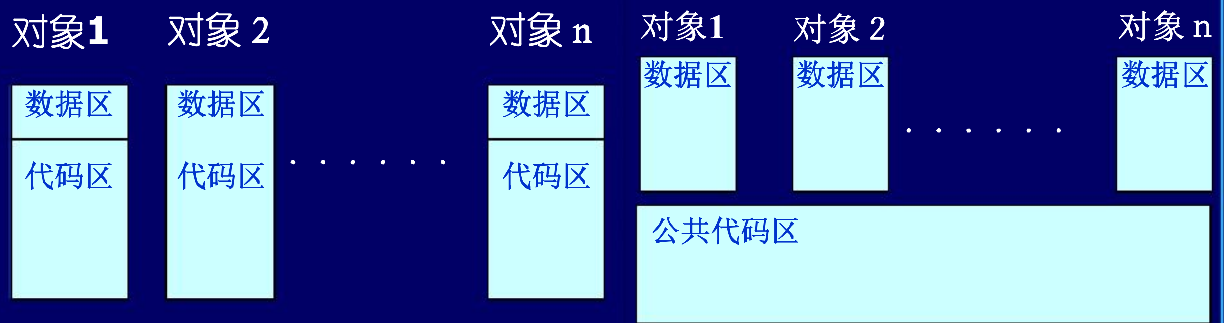


图1 各对象完全独立地安排内存的方案

图2 各对象的代码区共用的方案

图1对应的是在类说明中定义函数，
图2对应的是在类说明外部定义函数 。

内联成员函数举例(二)

```
class Location
{
 public:
    void Init(int initX,int initY);
    int GetX()；
    int GetY()；
 private:
    int X,Y;
};
```

```
inline void Point::
        Init(int initX,int initY)
{
   X=initX;
   Y=initY;
}

inline int Point::GetX（）
{
   return X;
}

inline int Point::GetY（）
{
   return Y;
}
```

注意事项

- 在类内的成员函数，如果函数体直接写在类内部，不必写inline， 自动为内联函数。
- 使用内联函数，要注意权衡效率、安全等各种因素。
- 内联函数使用不当，反而使代码增大的同时效率下降。
- 内联函数常用在简单的存取函数：内部变量定义为私有，定义几个相关的公有内联函数用于获得或改变私有部分。
- 赋值顺序：inline函数在同一个类内可以随便调用，而不必考虑哪个在前

§1.4 成员函数的特性

二、重载性

成员函数可以重载，构造函数可以重载；但析构函数不能重载。例如：

```
class M
{
 public:
    int add(int x, int y) {  return x+y; }
    double add(double x, double y) {  return x+y; }
 };
```

```
#include <iostream>
using namespace std;
void main(void)
{
      M a;
      int m, n;
      double x, y;
      cout<<"Enter two integer: ";
      cin>>m>>n;
      cout<<"integer ”  <<m <<'+’  <<n<<"=”  <<
    a.add(m,n)<<endl;
      cout<<"Enter two real number: ";
     cin>>x>>y;
     cout<<"real number "<<x<<'+'<<y<<"= "<<a.add(x,y)
    <<endl;
}
```

§1.4 成员函数的特性

三、设置参数的缺省值

成员函数可以被设置参数的缺省值。例如：

```
class Point
{
  public:
     Point(int xx=0,int yy=10){X=xx; Y=yy;}
     int GetX（）{return X;}
     int GetY（）{return Y;}
  private:
     int  X,Y;
};
```

```
#include <iostream>
using namespace std;
void main(void)
{  Point a，b(8),c(10,20);

   cout<<a.GetX（）+a. GetY( )<<endl;
   cout<<b.GetX（）+b. GetY( )<<endl;
   cout<<c.GetX（）+c. GetY( )<<endl;
}
```

运行结果：

10

18

40

§1.5 静态成员

静态成员的提出是为了解决数据共享的问题。

全局变量或对象可解决数据共享问题，但存在不安全性。

- 静态成员分类：静态数据成员和静态函数成员
- 使用类中的静态数据成员——解决访问权限控制问题。
- 使用静态成员函数——解决操作合法性控制问题。
- 静态成员只是为了程序抽象的需要而在类中所做的语法嵌入，不是对某个对象的抽象实现，所以也可以说静态成员不是对象的成员。
- 在类结构说明中以关键字static标明静态数据成员和静态成员函数，它们具有静态生存期

§1.5 静态成员

一、静态数据成员

- 用关键字static声明
- 静态数据成员在生成每一个类的对象时并不分配存储空间，而是该类的每个对象共享一个公共的存储空间，并且该类的所有对象都可以直接访问该存储空间。该类的所有对象维护该成员的同一个拷贝。
- 必须在类的实现部分定义和初始化，用(::)来指明所属的类。若没有进行初始化，则自动被初始化为 0。
- 其格式为：<数据类型类名>::<静态数据成员名>=<值>;

【例】用静态数据成员计算由同一类建立的对象的数量。

```
class  Ctest
{
private:
      static int count;//注意私有
public:
      Ctest(){
      ++count;cout<<"对象数量="<<count<<'\n';}
~Ctest(){
      --count;cout<<"对象数量="<<count<<'\n'; }
};
int Ctest::count=0; //A行  对静态数据定义性说明
void main(void)
{
    Ctest a[3];
}
```

执行程序后输出：

对象数量=1　//a[0]构造函数产生
对象数量=2　//a[1]构造函数产生
对象数量=3　//a[2]构造函数产生
对象数量=2　//a[2]析构函数产生
对象数量=1　//a[1]析构函数产生
对象数量=0　//a[0]析构函数产生

静态数据成员虽然具有全局变量的一些特性，但受到访问权限的约束。建议静态成员说明为私有的，从而保证面向对象程序设计的封装性。如果说明为公有的，它会带来与全局变量同样的副作用。

§1.5 静态成员

```
#include <iostream>
using namespace std;
class Point
{public:
    Point(int xx=0, int yy=0) {X=xx; Y=yy; countP++; }
     Point(Point &p);
    int GetX( ) {return X;}
    int GetY( ) {return Y;}
    void GetC( ) {cout<<" Object id="<<countP<<endl;}
private:
    int X,Y;
    static int countP;
};
```

```
Point::Point(Point &p)
{   X=p.X;
    Y=p.Y;
    countP++;
}

int Point::countP=0; //给静态数据成员countP初始化
void main( )
{   Point A(4,5);
    cout<<"Point A,"<<A.GetX( )<<","<<A.GetY( );
    A.GetC( );
    Point B(A);
    cout<<"Point B,"<<B.GetX( )<<","<<B.GetY( );
    B.GetC( );
}
```

运行结果：

Point A, 4，5 Object id= 1

Point B, 4，5 Object id= 2

§1.5 静态成员

二、静态成员函数

- 类外代码可以使用类名和作用域操作符来调用静态成员函数。形式为：

  <类名>:: <静态成员函数名>(<参数表>)

- 静态成员函数只能引用属于该类的静态数据成员或静态成员函数。如果静态成员函数中要引用非静态成员时，可通过对象来引用。

静态成员函数举例

```
#include<iostream.h>
class Application
{ public:
    static void f（）;
    static void g（）;
  private:
    static int global;
};
int Application::global=0;
void Application::f（）
{  global=5;}
void Application::g（）
{  cout<<global<<endl;}
```

```
int main（）
{
 Application::f（）;
 Application::g（）;
 return 0;
}
```

运行结果：

5

静态成员函数举例

```
class A
{
    public:
        static void f(A a);
    private:
        int x;
};
void A::f(A a)
{
    cout<<x; //对x的引用是错误的。
              //x是非静态数据成员。
    cout<<a.x;  //正确
}
```

静态成员函数举例

```
class A
{
    public:
        static void f(A a);
    private:
        int x;
};
void A::f(A a)
{
    cout<<x; //对x的引用是错误的。
              //x是非静态数据成员。
    cout<<a.x;  //正确
}
```

静态成员函数举例

```
#include <iostream>
using namespace std;
class Point        //Point类声明
{public: //外部接口
    Point(int xx=0, int yy=0) {X=xx;Y=yy;countP++;}
    Point(Point &p);        //拷贝构造函数
    int GetX（）{return X;}
    int GetY（）{return Y;}
    static void GetC（）
      {cout<<" Object id="<<countP<<endl;}
private: //私有数据成员
    int X,Y;
    static int countP;
}
```

```
Point::Point(Point &p)
{   X=p.X;
    Y=p.Y;
    countP++;
}
int Point::countP=0;
void main（）          //主函数实现
{  Point A(4,5);       //声明对象A
    cout<<"Point A,"<<A.GetX（）<<","<<A.GetY（）;
   A.GetC（）;         //输出对象号，对象名引用
   Point B(A); //声明对象B
    cout<<"Point B,"<<B.GetX（）<<","<<B.GetY（）;
   Point::GetC（）; //输出对象号，类名引用
}
```

静态成员函数举例

- 模拟商店一种货物的购进和卖出情况，要求该种货物以重量为单位成箱买卖，每箱的重量不相同，并且要求记录目前库存总重量。
  - 用类描述此种货物，每箱货物作为这个类的一个对象；该类中应该包含描述对象重量的数据成员和买、卖货物的成员函数（利用构造函数和析构函数）及得到每箱重量的成员函数。
  - 因为要求记录目前库存总重量，在类中添加一个静态数据成员来记录，并添加一个静态成员函数来从类中得到这个记录结果，使得这个总重量只能访问不能任意修改。

§1.6 友 元

- 友元是C++提供的一种破坏数据封装和数据隐藏的机制。
- 通过将一个模块声明为另一个模块的友元，一个模块能够引用到另一个模块中本是被隐藏的信息。
- 可以使用友元函数和友元类。
- 为了确保数据的完整性，及数据封装与隐藏的原则，建议尽量不使用或少使用友元。

§1.6 友 元

一、友元函数

- 友元函数是在类声明中由关键字friend修饰说明的非成员函数，在它的函数体中能够通过对象名访问 private 和 protected成员
- 作用：增加灵活性，使程序员可以在封装和快速性方面做合理选择。
- 访问对象中的成员必须通过对象名。

例 使用友元函数计算两点距离

```
#include <iostream>
using namespace std;
#include <math.h>
class Point     //Point类声明
{public:        //外部接口
    Point(int xx=0, int yy=0) {X=xx;Y=yy;}
    int GetX（ ） {return X;}
    int GetY（ ） {return Y;}
    friend float fDist(Point &a, Point &b);
private:        //私有数据成员
    int X,Y;
};
```

例 使用友元函数计算两点距离

```
double Distance( Point& a, Point& b)
{
      double dx=a.X-b.X;
      double dy=a.Y-b.Y;
      return sqrt(dx*dx+dy*dy);
}
int main（ ）
{  Point p1(3.0, 5.0), p2(4.0, 6.0);
    double d=Distance(p1, p2);
    cout<<"The distance is "<<d<<endl;
    return 0;
}
```

§1.6 友 元

二、友元类

- 若一个类为另一个类的友元，则此类的所有成员都能访问对方类的私有成员。
- 声明语法：将友元类名在另一个类中使用friend修饰说明。

友元类举例

```
#include <iostream>
using namespace std;
class A
{   friend class B;
    public:
      void Display（）
       {cout<<x<<endl;}
    private:
      int x;
};
class B
{   public:
      void Set(int i);
      void Display（）;
    private:
      A a;
};
```

```
void B::Set(int i)
{
   a.x=i;
}
void B::Display（）
{
   a.Display（）;
}
void main（）
{  A x;      B y;
    y.set(5);
    x. Display（）;
    y. Display（）;
 }
```

§1.8 常类型

常类型是指使用类型修饰符const说明的类型，常类型的变量或对象的值是不能被更新的。因此，定义或说明常类型量时必须进行初始化。

- C++中在某些情况下使用const代替#define，const量性质类似变量，但是值不可以改变
- 例：

```
const  buf=100; (默认int型)
const  int  buf=100;
```

§1.8 常类型

- 使用const的好处
- 在编译时预先计算常量表达式的结果。
- 例：

  ```
  const  s=3*100+20;
  ```

  使用时直接使用结果320
- 还可以用在运行期间产生的值初始化一个新定义的变量，并使用 const限制，防止在后面的程序中改变它：

  ```
  const  char  c = cin.get();
  const  char  c2=c+ ‘a’;
  ```
- c和c2都是常量，不可改变：c= ‘2’ ;//错
- *常类型的变量或对象必须进行初始化，而且不能被更新。*

§1.8 常类型

一、一般常量和对象常量

1、一般常量

定义或说明格式：

<类型说明符> const <变量名>=<初始值>

或 const <类型说明符> <变量名>=<初始值>

例如，int const x=20;

const int x=20;

注：const的位置可前可后

§1.8 常类型

一、一般常量和对象常量

- 常数组：数组元素不能被更新。

<类型说明符> const <数组名>[大小]…

或 const <类型说明符> <数组名>[大小]…

例如，int const a[5]={1,2,3,4,5};

const int a[5]={1,2,3,4,5};

2、常对象

- 常对象：必须进行初始化, 不能被更新。

<类名> const <对象名>

或 const <类名> <对象名>

§1.8 常类型

一、一般常量和对象常量

```
class  A
{
    public:
        A(int i=0, int j=0){  a=i; b=j;}
        void  print( ) { cout<<a<< “,” <<b<<endl; }
    private:
        int  a,b;
};
A const a1(3,4);
const A a2(5,8);
```

§1.8 常类型

二、常指针和常引用

1、常指针

- 指向常量的指针
  - 形式： const <类型> *<指针名>= <指针值>;
  - 不能通过指针来改变所指对象的值，但指针本身可以改变，可以指向另外的对象。例：

```
const char *name1 = "John";
int n1=3;   int const n2=5;    const int *pn= &n1;
pn=&n2;  //正确，指针是变量
*pn=6;  //错误，内容是常量
```

§1.8 常类型

二、常指针和常引用

1、常指针

- 指针常量
  - 形式： <类型> * const <指针名>= <指针值>;
  - 指针本身的值不能被改变，但可通过指针来改变所指对象的值。例：

```
char *const name2 = "John";
int n1=3;   int const n2=5;   int  *const pn= &n1;
pn=&n2; //错误
*pn=6; //正确
```

§1.8 常类型

二、常指针和常引用

2、常引用

- 被引用的对象不能被更新。

  形式： <类型> const &<引用名>;

  例： int const &n

  n=6; //错误

在实际应用中，常指针和常引用往往用来作函数参数，这样的参数称为常参数。

在C++中，常指针和常引用常用于对象参数的传递，这样就不需要执行拷贝初始化构造函数，从而改善程序的运行效率。

常指针作函数参数的例子

// P_{186} [] 指向常量的指针做形参

```
#include<iostream.h>
const int N=6;
void print(const int *p,int n);
void main（ ）
{  int array[N];
    for(int i=0;i<N;i++)
        cin>>array[i];
    print(array,N);
}
```

```
void print(const int *p, int n)
{
    cout<<"{"<<*p;
    for(int i=1;i<n;i++)
       cout<<" ， "<<*(p+i);
    cout<<"}"<<endl;
}
```

运行：

1 2 3 4 5 6↵

{1, 2, 3, 4, 5, 6}

```
#include <iostream.h> // P187 常引用作函数参数
class  K
{
    public:
      K(int i){  k=i;}
      int  setk( ) const{ return k; }
    private:
       int  k;
};
int add(const K &g1, const K &g2)
{
     int  sum=g1.setk( )+g2.setk( );
     return sum;
}
```

```
void main()
{
    K k1(8), k2(17);

    int s=add(k1,k2);
    cout<<s<<endl;
}
```

运行结果：

25

```
#include<iostream.h> //常引用做形参例子
void display(const double& r);
int main()
{
    double d(9.5);
    display(d);
    return 0;
}
void display(const double& r)
//常引用做形参，在函数中不能更新 r所引用的对象。
{
     cout<<r<<endl;
}
```

§1.8 常类型

三、常成员函数

- 使用const关键字说明的函数。
- 只有常成员函数才有资格操作常量或常对象。
- 常成员函数不更新对象的数据成员。
- 常成员函数说明格式：

  <类型说明符> <函数名>（<参数表>）const;
- 这里，const是函数类型的一个组成部分，因此在实现部分也要带const关键字。
- const关键字可以被用于参与对重载函数的区分

常成员函数例题

```
#include<iostream.h>// 常成员函数

class R
{    public:
         R(int r1, int r2){R1=r1;R2=r2;}
         void print();
         void print() const;
       private:
         int R1,R2;
};
```

§1.8 常类型

```
void R::print()
{    cout<<R1<<":"<<R2<<endl;
}
void R::print() const
{    cout<<R1<<";"<<R2<<endl;
}
void main()
{   R a(5,4);
     a.print();  //调用void print()
     const R b(20,52);
     b.print();  //调用void print() const
}
```

运行结果：
5：4
20；52

§1.8 常类型

四、常数据成员

- 使用const关键字说明的数据成员。
- 通过成员初始化列表方式来生成构造函数对数据成员初始化。

常数据成员例题

```
#include<iostream.h>// 常数据成员
class  A
{
    public:
      A(int i);
      void  print( );
      const int &r;
    private:
       const int  a;
       static  const  int b;
};
const  int A::b=10;
```

```
A:: A(int i):a(i),r(a){   }//构造函数
void A:: print( )
{
   cout<<a<<":"<<b<<":"<<r<<endl;
}

void main()
{
    A a1(100),a2(0);
    a1.print();
    a2.print();
}
```

运行结果：100:10:100
          0:10:0

§1.9 子对象和堆对象

一、子对象

- 类中的成员数据是另一个类的对象。
- 可以在已有的抽象的基础上实现更复杂的抽象。通过对复杂对象进行分解、抽象，使我们能够将一个复杂对象理解为简单对象的组合。
- 分解得到复杂对象的部件对象，这些部件对象比它高层的复杂对象更容易理解和实现。然后由这些部件对象来“装配”复杂对象。

子对象例题

```
class Point
{   private:
          float x,y; //点的坐标
     public:
          Point(float h,float v); //构造函数
          float GetX(void); //取X坐标
          float GetY(void); //取Y坐标
          void Draw(void); //在(x,y)处画点
};
//...函数的实现略
```

子对象例题

```
class Line
{
    private:
        Point  p1,p2; //线段的两个端点
    public:
        Line(Point a,Point b); //构造函数
        Void Draw(void); //画出线段
};
//...函数的实现略
```

类聚集的构造函数设计

- 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。
- 定义形式：

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 }

类聚集的构造函数调用

- 构造函数调用顺序：先调用内嵌子对象的构造函数（按内嵌时的定义顺序，先定义者先构造）。然后调用本类的构造函数。（析构函数的调用顺序相反）
- 若调用缺省构造函数（即无形参的），则子对象的初始化也将调用相应的缺省构造函数。

类聚集的应用

```
class Part  //部件类
{
    public:
        Part();
        Part(int i);
        ~Part();
        void Print();
    private:
        int val;
};
```

```
class Whole
{
    public:
        Whole();
        Whole(int i,int j,int k);
        ~Whole();
        void Print();
    private:
        Part one;
        Part two;
        int date;
};
```

类聚集的应用

```
Whole::Whole()
{
  date=0;
}
Whole::Whole(int i,int j,int k):
        two(i),one(j),date(k)
{}//对普通数据成员的初始化也可以在聚集类构造函数后的成员初始化列表中进行

//...其它函数的实现略
```

类聚集的应用

```
#include <iostream.h>
#include <math.h>
class Point    //Point类定义
{
public:
    Point(int xx=0, int yy=0) {X=xx;Y=yy;}
    Point(Point &p);
    int GetX() {return X;}
    int GetY() {return Y;}
private:
    int X,Y;
};
```

```
Point::Point(Point &p) //拷贝构造函数的实现
{
    X=p.X;
    Y=p.Y;
    cout<<"Point拷贝构造函数被调用"<<endl;
}
//类的聚集
class Distance//Distance类的定义
{
public: //外部接口
    Distance(Point xp1, Point xp2);
    double GetDis(){return dist;}
private:          //私有数据成员
    Point p1,p2;          //Point类的对象p1,p2
    double dist;
};
```

```
Distance::Distance(Point xp1, Point xp2)
:p1(xp1),p2(xp2) //类的聚集的构造函数
{
    cout<<"Distance构造函数被调用"<<endl;
    double x=double(p1.GetX()-p2.GetX());
    double y=double(p1.GetY()-p2.GetY());
    dist=sqrt(x*x+y*y);
}
//主函数
void main()
{
    Point myp1(1,1),myp2(4,5);   //定义Point类的对象
    Distance myd(myp1,myp2);   //定义Distance类的对象
    cout<<"The distance is: ";
    cout<<myd.GetDis()<<endl;
}
```

§1.9 子对象和堆对象

二、堆对象

- 所谓堆对象是指在程序运行过程中间根据需要随时可以建立或删除的对象。
- 这种对象被创建在内存的空闲存储单元内，这些单元称为堆。故为得名为堆对象。
- 创建和删除运算符为：
- new //创建运算符，相当于C语言中的malloc( )
- delete//删除运算符，相当于C语言中的free( )

§1.9 子对象和堆对象

二、堆对象

- **1、运算符new用法**
- 功能：用于动态创建对象或动态变量(动态申请内存)
- 格式： new <类型说明符>(<初始值列表>)
- 执行结果：成功：类型的指针，指向新分配的内存；
- 失败：0（NULL，空指针）
- 创建数组类型对象(变量)格式：
- new <类型说明符>[<算术表达式>]
- 注意：创建数组不能初始化

§1.9 子对象和堆对象

二、堆对象

- **2、运算符delete用法**
- 功能：删除动态创建的对象或动态变量 (动态释放内存), 释放指针所指向的内存。指针必须是new操作的返回值。
- 格式： delete <指针名>//删除动态对象或动态变量
  delete[] <指针名> >//删除动态数组

例 动态创建对象举例

```
#include<iostream.h>
class Point
{ public:
    Point（ ）
    { X=Y=0; cout<<"Default Constructor called.\n";}
    Point(int xx,int yy);
    {   X=xx; Y=yy; cout<< "Constructor called.\n";  }
    ~Point（ ）  {   cout<<"Destructor called.\n";    }
    void  print( ) {   cout<<X<<","<<Y<<endl;   }
    int GetX（ ）  {return X;}
    int GetY（ ）  {return Y;}
    void Move(int x,int y)      {  X=x;  Y=y;   }
  private:
      int  X,Y;
};
```

例动态创建对象举例

```
int main（）
{   cout<<"Step One:"<<endl;
    Point *Ptr1=new Point;
    Ptr1->print ( );
    delete  Ptr1;
    cout<<"Step Two:"<<endl;
    Ptr1=new Point(1,2);
    Ptr1->print ( );
    delete Ptr1;
    return 0;
}
```

运行结果：

Step One:
Default Constructor called.
0,0
Destructor called.
Step Two:
Constructor called.
1,2
Destructor called.

例 动态创建对象数组举例

```
#include<iostream.h>
class Point
{   //类的声明同，略
};
int main（ ）
{   Point *Ptr=new Point[2];    //创建对象数组
    Ptr[0].Move(5,10);      //通过指针访问数组元素的成员
    Ptr[1].Move(15,20);   //通过指针访问数组元素的成员
    Ptr[0].print ( );     Ptr[1].print ( );
    cout<<"Deleting..."<<endl;
    delete[ ] Ptr;              //删除整个对象数组
    return 0;
}
```

例 动态创建对象数组举例

运行结果：

Default Constructor called.

Default Constructor called.

5,10

15,20

Deleting...

Destructor called.

Destructor called.

例 动态创建多维数组

```
#include<iostream.h>
void main（ ）
{   float (*cp)[9][8];
    int i,j,k;
    cp = new float[8][9][8];
    for (i=0; i<8; i++)
          for (j=0; j<9; j++)
                for (k=0; k<8; k++)
                   *(*(*(cp+i)+j)+k)=i*100+j*10+k;
                              //通过指针访问数组元素
```

例 动态创建多维数组

```
for (i=0; i<8; i++)
    {     for (j=0; j<9; j++)
          {     for (k=0; k<8; k++)
                     //将指针cp作为数组名使用,
                     //通过数组名和下标访问数组元素
                    cout<<cp[i][j][k]<<"  ";
                   cout<<endl;
          }
          cout<<endl;
    }
}
```

动态存储分配函数

- void *malloc(size);

参数size：欲分配的字节数

返回值：　成功，则返回void型指针。
　　　　　失败，则返回空指针。

头文件：　<stdlib.h> 和 <malloc.h>

动态内存释放函数

- void free(void *memblock);

参数memblock：

指针，指向需释放的 内存。

返回值：无

头文件：<stdlib.h> 和 <malloc.h>

例 动态存储分配举例

```
#include <iostream.h>

struct date
{
   int month;
   int day;
   int year;
};
```

例 动态存储分配举例

```
int main（ ）
{
  int index, *point1, *point2;
  point1 = &index;
   *point1 = 77;
   point2 = new int;
   *point2 = 173;
   cout << "The values are " << index << "  "
            << *point1 << "  " << *point2 << "\n";
   delete point2;
   point1 = new int;
   point2 = point1;
   *point1 = 999;
   cout << "The values are " << index << "  "
            <<*point1 << "  " << *point2 << "\n";
   delete point1;
```

例　动态存储分配举例

```
float *float_point1, *float_point2 = new float;

  float_point1 = new float;
  *float_point2 = 3.14159;
  *float_point1 = 2.4 * (*float_point2);
  delete float_point2;
  delete float_point1;
```

例 动态存储分配举例

```
date *date_point;

   date_point = new date;  //动态分配结构体
   date_point->month = 10;
   date_point->day = 18;
   date_point->year = 1938;
   cout << date_point->month << "/" << date_point->day
    << "/"
                             <<date_point->year << "\n";
   delete date_point;  //释放结构体
```

例　动态存储分配举例

```
char *c_point;
  c_point = new char[37];   //动态分配数组
  delete [ ] c_point;       //释放数组
  c_point = new char[sizeof(date) + 133];
                            //动态分配数组
  delete [ ] c_point;   //释放数组
  return 0;
}
```

运行结果：

```
The values are 77  77  173
The values are 77  999  999
10/18/1938
```

§1.10 类型转换

类型转换包含自动隐含转换和强制类型转换两种。

一、类型的自动隐式转换

1、程序在执行算术运算时，低类型向高类型转换

例如：int k; k=88+’A’;

2、赋值运算：将右边结果自动转换为左边类型

例如：char ch; ch=8+’A’;

3、函数调用时：将实参值自动转换为形参类型

例如，double f(double d); double x=f(5);

4、函数返回值：自动将返回值转换为函数类型

- 例如，double f(double d){ float x; …… return x;}

§1.10 类型转换

二、构造函数具有类型转换功能

单个参数 的构造函数提供了数据类型转换的功能。例如，

```
class  A
{
    public:
        A( ){  m=0;}
        A(double i){  m=i;}
        void  print( ) { cout<<m<<endl; }
    private:
        double m;
};
```

§1.10 类型转换

```
void main()
{
    A a1(100),a2;
    a1.print();
    a2=200;
    a2.print();
}
```

执行结果为：

100

200

§1.10 类型转换

三、转换函数

- 转换函数又称类型强制转换成员函数，它是类中的一个非静态成员函数。
- 定义格式：

```
class  <类型名1>
{
    public:
        operator <类型名2 >( );
         ……
};
```

- 功能：将<类型1>转换为<类型名2>

转换函数例题

```
class  Rational >// 转换函数
{
    public:
        Rational(int d, int m){  den=d; num=m; }
        operator  double( )；
    private:
        int den, num;
};
Rational:: operator  double( )
{     return double (den)/ double(num);
}
```

转换函数例题

```
void main()
{
    Rational r(4,8);
    double  d=4.7;
    d+=r;
    cout<<d<<endl;
}
```

运行结果：

5.2

定义转换函数注意事项

1、转换函数是用户定义的成员函数，但它是非静态的。
2、转换函数不带任何参数
3、转换函数不能定义为友元函数。

本章小结

1、类的定义；
2、对象的定义、初始化；
3、成员函数的特性；
4、静态成员；
5、友元；
6、类的作用域

重点：

1、类、对象的定义与使用；
2、利用类和对象进行程序设计