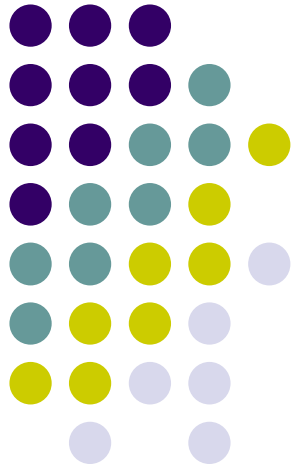
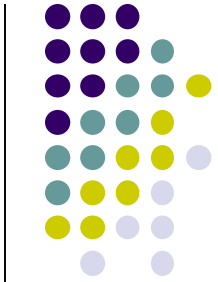
第六章 类和对象

C++程序设计





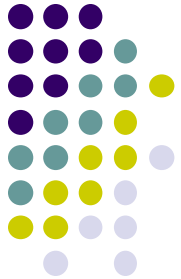
```
struct Student
{
        int   number;
        char name[20];
};
```

```
struct Student stu1,stu2,*ps,stu[30];
```

```
stu1.name;
```

```
stu2.number;
```

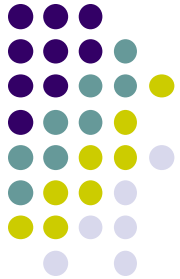
```
ps->name;
```



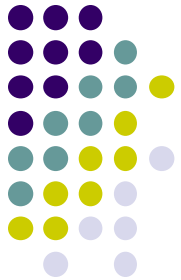
```
 class Student
{
private:
                int   number;
                char name[20];
public:
                int     GetNumber();
                char* GetName();
                void   SetNumber(int num);
                void   SetName(char* n);
};
Student stu;
```

类和对象

——对象的概念



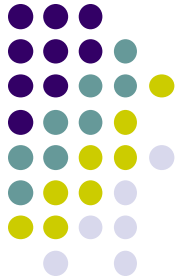
- 描述客观事物的实体，是类的实例，是类类型的变量
- 一个对象是由一组属性和对这组属性进行操作的一组服务构成的



```
class Student
{
private:
        int    number;       //4B
        char name[20];  //20B
public:
        int     GetNumber();
        char*GetName();
        void   SetNumber(int num);
        void   SetName(char* n);
};
Student stu;     //24B内存空间
```

类和对象

——类的实例

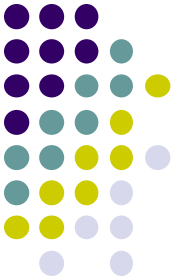


定义一个学生类，实现学生基本信息的存取

- 属性：
  - 学号
  - 姓名
- 操作：
  - 存入相关信息
  - 取出相关信息

类和对象

——文件结构

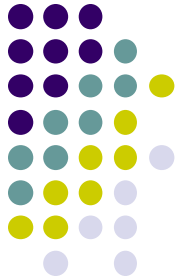


从结构上来看，类的定义与使用属于多文件结构

- class.h ： 类的外部接口（供其他程序引用）
- class.cpp ： 类的内部实现
- classApp.cpp ： 使用类的文件

各个.cpp的文件独立编译成.obj文件，并加入到项目中，连接程序（Bulid）会自动将它们连接成一个可执行文件.exe。

class原型声明文件Student.h



实现封装

数据类型1

```
class Stud
{
private:
        int   number;
        char name[20];
public:
        int     GetNumber();
        char*GetName();
        void   SetNumber(int num);
        void   SetName(char* n);
};
```

数据成员

数据类型2

成员函数

定义数据操作

class实现文件Student.cpp

```
#include “Student.h"
#include “string.h"
```

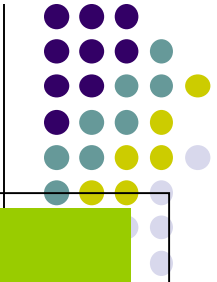
```
1. void Student::SetName(char* n)
2. { strcpy(name,n);  }

3. void Student::SetNumber(int num)
4. {   number=num;  }

5. char * Student::GetName()
6. {   return name；    }

7. int Student::GetNumber()
8. {   return number; }
```

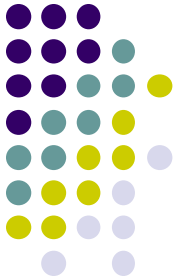
class使用文件studentApp.cpp



```
#include <iostream.h>
#include “Student.h"
```

```
void main()
{
        Student s1,s2;
        char *name_s1;
        int  number;
        int a1=1,a2=2;
        s1.SetName("Li");
        s1.SetNumber(a1);
        s2.SetName("zhang");
        s2.SetNumber(a2);
        name_s1=s1.GetName();
        number=s1.GetNumber();
        cout<<"name of s1 is : "<<name_s1<<endl;
        cout<<"number of s1 is : "<<number<<endl;
}
```

抽象实例——钟表(Clock)

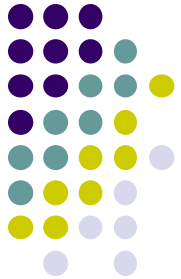


- 数据抽象：

  int Hour，int Minute，int Second
- 代码抽象：

  SetTime()，ShowTime()

抽象实例——钟表类



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

封装

- 实例：

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

外部接口

特定的访问权限

边界

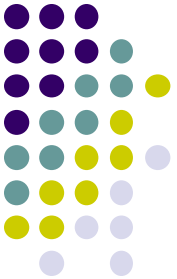
类的成员

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

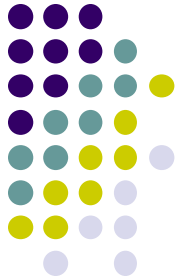
成员函数

成员数据

成员函数的实现



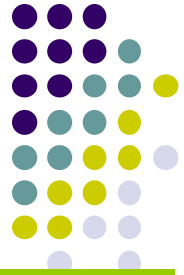
```
void Clock :: SetTime(int NewH, int NewM, int NewS)
{
    Hour=NewH;
    Minute=NewM;
    Second=NewS;
}
void Clock :: ShowTime()
{
    cout<<Hour<<":"<<Minute<<":"<<Second;
}
void main()
{     Clock  myClock;
      myClock.SetTime(8,30,30);
      myClock.ShowTime();
}
```



```
1.  class Student
2.  {
3.      int  number;   //没有限定符，默认为private
4.  public：
5.      void SetNumber(int num)
6.          {number=num;}
7.   };
8.   void main()
9.  {
10.    Student  s;
11.    s.number=12;          // 错误
12.    s. SetNumber(12);   //正确
13. }
```

类和对象

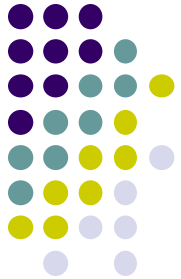
——数据成员



```
1.  class  Point
2.  {
3.  private:
4.       int x,y;
5.  public：
6.       Point( ){ }                    //默认构造函数
7.       Point( int a=0, int b=0){x=a;y=b;}//带参数构
8.                                          造函数
9.       Point(Point &a);          //拷贝构造函数
10.       ~Point( ){ }                //析构函数
11.   ……
12.  };    Point b1; Point b2(5,6); Point b3(b2);
```

类和对象

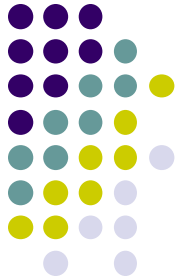
——构造函数



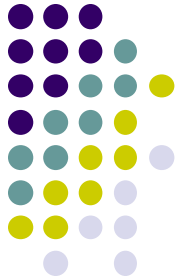
- 作用是在对象被创建时使用特定的值构造对象，或者说将对象初始化为一个特定的状态；
- 构造函数是与类同名的函数，没有返回值类型，可以有参数；
- 在对象创建时由系统自动调用；
- 如果程序中未声明，则系统自动产生出一个默认形式的构造函数；
- 允许为内联函数、重载函数、带默认形参值的函数。

类和对象

——析构函数



- 与类同名，没有返回值和参数。
- 完成对象被删除前的一些清理工作。
- 在对象的生存期结束的时刻系统自动调用它，然后再释放此对象所属的空间。
- 如果程序中未声明析构函数，编译器将自动产生一个默认的析构函数。



对象创建　　分配对象空间

构造函数（数据空间初始化） Student(…){…}

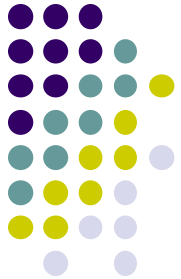
Student s(10,”zhang”);

使用

析构函数（主要是堆清理）~Student(){…}

对象销毁　　释放对象空间

构造函数举例

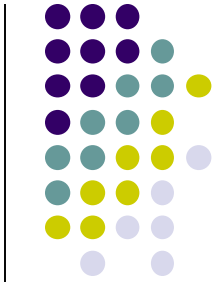


```
class Clock
{
public:
    Clock (int NewH, int NewM, int NewS);//构造函数
    void SetTime(int NewH, int NewM, int NewS);
    void ShowTime();
private:
    int Hour,Minute,Second;
};
```

构造函数的实现：

```
Clock::Clock(int NewH, int NewM, int NewS)
{
    Hour= NewH;
    Minute= NewM;
    Second= NewS;
}
```

```
void Clock :: SetTime(int NewH, int NewM, int NewS)
{
        Hour=NewH;
        Minute=NewM;
        Second=NewS;
}
```



构造函数的实现：

```
Clock::Clock(int NewH, int NewM, int NewS)
{
    Hour= NewH;
    Minute= NewM;
    Second= NewS;
}
```

建立对象时构造函数的作用：

```
int main()
{
    Clock  c (0,0,0); //隐含调用构造函数，将初始值作为实参。
    C.SetTime（8，30，55）；
    c.ShowTime();
}
```

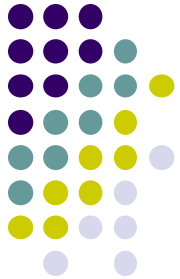
拷贝构造函数

拷贝构造函数是一种特殊的构造函数，其形参为本类的对象引用。

```
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    }
```

类和对象

——拷贝构造函数



- 形式：**Student(Student &a);**

  //类的对象引用作形参

- 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默认的拷贝构造函数。
- 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。

类和

1.

2.

3.

```
Point fun2()
{    Point A(1,2);
     return A; //调用拷贝构造函数
}
int main()
{    Point B;
     B=fun2();
}
{  cout<<p.GetX()<<endl;}
void main()
{
    Point A(1,2);
    fun1(A); //调用拷贝构造函数
}
```

关于类的拷贝构造函数，下面哪个叙述是正确的？（ A ）

A.如果成员函数的形参是类的对象，则该函数在调用时会自动调用该类的拷贝构造函数。

B. 如果函数的形参是类的对象的引用，则该函数在调用时会自动调用该类的拷贝构造函数。

C. 如果函数的形参是指向类的对象的指针，则该函数在调用时会自动调用该类的拷贝构造函数。

D.上面的说法都不对。

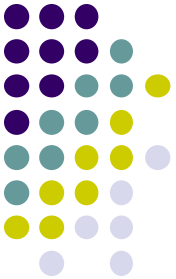
例6.3 分析下列程序的输出结果。

对象

```
#include<iostream.h>
class Point
{public: Point(int i,int j)
         {X=i;Y=j;}
         Point(Point &rp);
         ~Point()
         {cout<<"Destructor called.\n";}
         int Xcood()
         {return X;}
         int Ycood()
         {return Y;}
private: int X,Y;
};
Point::Point(Point &rp)
{        X=rp.X; Y=rp.Y;
         cout<<"Copy Constructor called.\n";
}
```

```
void main()
{Point p1(6,9);
 Point p2(p1);
 Point p3=p2,p4(0,0);
 p4=p1;
 cout<<"p3=("<<p3.Xcood()<<','<<p3.Ycood()<<")\n";
 cout<<"p4=("<<p4.Xcood()<<','<<p4.Ycood()<<")\n";
}
```

```
Copy Constructor called.
Copy Constructor called.
P3=(6,9)
P4=(6,9)
Destructor called.
Destructor called.
Destructor called.
Destructor called.
```



对类的构造函数和析构函数描述正确的是
（C）

A. 构造函数和析构函数都不能重载。

B. 即使一个类存在其他构造函数，系统也会创建默认构造函数。

C. 在默认的拷贝构造函数中，会将另一个对象的数据成员逐个拷贝。

D. 如果构造函数的某些参数有默认值就不必再写默认构造函数了。

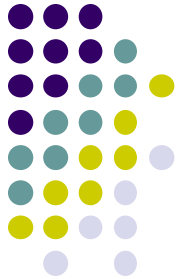
```
#include <iostream.h>
#include <string.h>
class string
{
    char *str;
public:
    string(char *s)
     {
          str = new char[strlen(s) + 1];
          strcpy(str,s);
          cout<<"A"<<strlen(str);
      }
    void print()
    {        cout<<str<<",";  }
     ~string()
   {    cout<<"#"; delete str; }
    string(string &obj)
    {
          str = new char[strlen(obj.str) + 1];
          strcpy(str,obj.str);
          cout<<"B"<<strlen(str);
     }
};
```

```
void main()
{
        string str1 = "hi";
        string str2("world");
        string str3 = str1;
        cout << endl;
        str1.print();
        str2.print();
        str3.print();
        cout << "!" <<endl;
}
```

A2A5B2

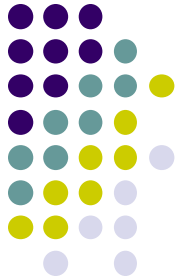
hi,world,hi,!

###



类和对象

——函数成员的定义



1. 在类体内声明成员函数，在类体外给出定义

```
class Point {
 private:
    int X,Y;
 public:
    void SetPosition(int a,int b);
 }
```

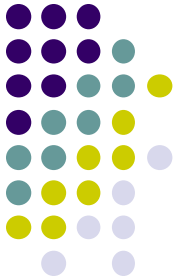
类体内

```
  void Point :: SetPosition(int a,int b)
{
        X=a;
        Y=b;
  }
```

类体外

类和对象

——函数成员的定义



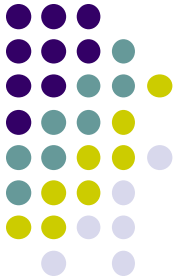
2.类体内定义与实现

```
class Point {
        private:
              int X,Y;
        public:
              void SetPosition( int a,int b);
              // void SetPosition( int a,int b){ X=a;Y=b;}
} ;
```

```
inline void Point :: SetPosition （ int a, int b ）
{
      X=a;
      Y=b;
}
```

类和对象

——inline内联函数



- 在类体内部直接给出函数的实现或类外加**inline**关键字的函数实现
- 编译时在调用处用函数体进行替换,节省了参数传递、控制转移等开销。
- 注意：
  - 内联函数体内不能有循环语句和**switch**语句。
  - 内联函数的声明必须出现在内联函数第一次被调用之前。

关于函数的说法，下面哪种叙述是正确的（B ）

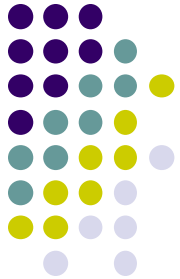
A. 函数名相同的函数可以重载，但函数内部的代码必须相同或大部分相同。

B. 内联函数执行起来速度比一般函数快，其原因是省去了函数调用时的寻址和返回的时间。

C. 当函数形参有默认值时，实参只能采用这个默认值。

D. 函数参数的引用调用是传值调用的另一种形式，其参数传递方式相同。

对象的生存期



- 对象从产生到结束的这段时间就是它的生存期
- 在对象生存期内，对象将保持它的值，直到被更新为止。

1. 静态生存期：这种生存期与程序的运行期相同
2. 动态生存期：始于声明点，终于作用域结束点

例

```
#include<iostream.h>
void fun();
int main()
{   fun();
    fun();
}
void fun()
{   static int a=1;
    int i=5;
    a++;
    i++;
    cout<<"i="<<i<<",a="<<a<<endl;
}
```

运行结果：

i=6, a=2

i=6, a=3

i是动态生存期

a是静态生存期

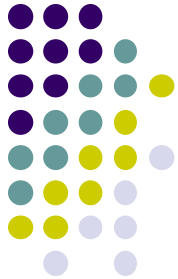
静态成员

- 静态数据成员
  - 在类体内使用关键字static说明的成员称为静态成员。静态成员包括静态数据成员和静态成员函数两种。
  - 静态成员的特点是它不是属于某对象的，而是属于整个类的，即所有对象的。
  - 用来保存流动变化的对象个数。

```
class A
{
public: A(int i);
private:
    int a;
    static int b;
    ……
};
int A::b = 10;
……
```

在对象产生之前就已经存在了！
具有文件生命期。

因此必须在对象之外初始化。



静态数据成员的特点

静态数据成员不是属于某个对象，而是属于整个类的。
静态数据成员具有共享性和唯一性

静态数据成员不随对象的创建而分配内存空间，它也不随对象被释放而撤销。只有在程序结束时才被系统释放。
静态数据成员具有文件生命期

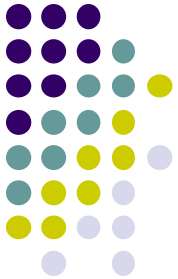
静态数据成员只能在类体外被初始化。
静态数据成员必须初始化。

例6.7 分析下列程序的输出结果。

```
#include<iostream.h>
class MY
{public: MY(int i,int j,int k);
         void PrintNumber();
         int GetSum(MY m);
private: int a,b,c;
         static int s;
};
int MY::s=0;
MY::MY(int i,int j,int k)
{        a=i;b=j;c=k;
         s=a+b+c;
}
void MY::PrintNumber()
{        cout<<a<<','<<b<<','<<c<<endl;
}
int MY::GetSum(MY m)
{        return MY::s;
}
```

```
void main()
{
        MY m1(2,3,4),m2(5,6,7);
        m2.PrintNumber();
        cout<<m1.GetSum(m1)<<',‘
        <<m2.GetSum(m2)<<endl;
}
```

5,6,7
18,18



定义静态成员函数的格式如下:

static 返回类型 静态成员函数名（参数表）;

与静态数据成员类似,调用公有静态成员函数的一般格式有如下几种:

类名::静态成员函数名(实参表)

对象. 静态成员函数名(实参表)

对象指针->静态成员函数名(实参表)

静态成员函数

静态成员函数的实现可放在类体内，也可以放在类体外。

在静态成员函数中可以直接引用其静态成员，而引用非静态成员时需用对象名引用。

```
class A
{
    public:
        static void f(A a);
    private:
        int x;
};
static void A::f(A a)
{
    cout<<x;
    cout<<a.x;
}
```

静态成员函数

静态成员函数的实现可放在类体内，也可以放在类体外。

在静态成员函数中可以直接引用其静态成员，而引用非静态成员时需用对象名引用。

```
class A
{
    public:
        static void f(A a);
    private:
        int x;
};
static void A::f(A a)
{
    cout<<x; //对x的引用是错误的
    cout<<a.x;  //正确
}
```

- 【例6.8】分析下列程序的输出结果，学会静态成员函数的用法。

```
#include <iostream.h>
#include <string.h>
class Student
{
public:
  Student(char name1[],int sco)
  {
   strcpy(name,name1);
   score=sco;
  }
  void total()
  { sum+=score;  count++;  }
  static double aver()
  { return (double) sum/count;   }
private:
  char name[20];
  int score;
  static int sum, count;
};
```

//静态函数成员可以直接访问静态数据成员

静态成员函数

静态数据成员

```
int Student::sum=0;
int Student::count=0;
void main()
{
   Student stu[5]
   ={Student("Ma",89),Student("Hu",90),Student("LU",9
   5),Student("Li",88),Student("Gao",75)};
   for(int i=0;i<5;i++)
       stu[i].total();
   cout<<"Average="<<Student::aver()<<endl;
}
```

静态数据成员初始化

对象数组

虽然有5个对象，但aver()是唯一的。

程序输出：

```
Average=87.4
```