第五章 类和对象

5.1类的结构

- C++的类是对象的抽象
  - 数据成员。
  - 成员函数。

```
class 标识符
{
public:
    数据成员；
    成员函数；
private:
    数据成员；
    成员函数；
};
```

- C++类的说明

4.1类的结构（外部说明）

举例：

```
Class Rectangle
{
  public:
    Rectangle(int x,int y,int w,int h);
    void Move(int x,int y);
    void Size(int w,int h);
    void Where(int& x,int& y);
    int Area();
    void Draw();
  private:
    int X,Y,W,H;
};
```


5.1类的结构（外部说明）

- 访问控制描述符
  - public后声明的是公有数据成员和成员函数，可被任何程序单元引用。
  - private后声明的是私有数据成员和成员函数，仅可被类内部引用。
  - public和private出现的顺序和次数是任意的。

5.1类的结构（内部实现）

- 类成员函数的实现

```
返回类型  类名::成员函数名（参数声明）
{
  语句序列；
}
```

举例

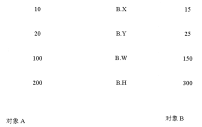
```
Void Rectangle::Move(int x,int y)
{ X=x;
  Y=y;
}
void Rectangle::Size(int w,int h)
{ W=w;
  H=h;
}
```


5.2类的实例化--对象的实现

- 一般语法：
  - 类标识符 对象标识符；
- 可以声明多个对象，对象之间代码共享，数据独立。
- 可使用对象名引用对象的公有成员函数。方法如下：
  - 对象名.成员函数名（参数列表）；

5.2类的实例化--对象的实现

- 成员函数所操作的数据成员是该类的某个对象的数据成员。



5.2类的实例化--对象的实现

举例

```
int main(){
    Rectangle A(10,20,100,200),B(15,25,150,240);
    A.Move(10,20);
    A.Size(100,200);
    B.Move(10,20);
    B.Size(100,200);
    return 0;
    }
```

**将一个标识符与一个程序实体相关联的过程称绑定。分静态和动态绑定两种。静态绑定在编译时进行，动态绑定在运行时进行。

5.3构造函数和析构函数

- 构造函数的作用是在对象被创建时使用特定的值构造对象（初始化）。
- 析构函数的作用是在对象被删除时做一些清理工作。
- 构造函数与类同名。
- 析构函数是类名前加~
- 两者无返回类型，析构函数无参数。

5.3构造函数和析构函数

举例

```
#include <iostream.h>
class Location{
public:
   Location(int x,int y);
   ~Location();
   int GetX();
   int GetY();
private:
   int X,Y;
};
```

```
Location::Location(int x,int y)
{
   X=x;   Y=y;}
Location::~Location()
{  cout<<"Destructor called."<<endl;}
int Location::GetX()
{return X;}
int Location::GetY()
{return Y;}
```



```
#include <iostream.h>
int main()
{
   Location A(10,20);     //构造函数被调用
   cout<<A.GetX()<<","<<A.GetY()<<endl;
   return 0;
   //析构函数被调用
   }
   输出结果：
   Constructor called
   10,20
   Destructor called.
```


5.4 成员函数的内联实现

- 函数体放在类体内。
- 函数体放在类体外，使用inline关键字。

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



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


5.5带缺省参数值的成员函数和成员函数重载

```
#include <iostream.h>
class Location{
public:
      Location(int x=0,int y=0);
      void Move(int x=5,int y=5);
      void ValueX(int x);
      int ValueX();
      void ValueY(int y);
      int ValueY();
private:
      void Set(int x,int y);
      int X,Y;
};
```



```
Location::Location(int x,int y)
{Set(x,y);}
void Location::Move(int x,int y)
{Set(x,y);}
void Location::ValueX(int val)
{X=val;}
int Location::ValueX()
{return X;}
void Location::ValueY(int val)
{Y=val;}
int Location::ValueY()
{return Y;}
void Location::Set(int x,int y)
{X=x;Y=y;}
```



```
int main()
{       Location A,B;       //Location A(0,0),B(0,0)
        A.Move();           //A.X和A.Y被设为5
        A.ValueX(15);     //A.X=15
       cout<<A.ValueX()<<A.ValueY()<<endl;
       B.Move(6,2);       //B.X=6,B.Y=2
       B.ValueY(4);
      cout<<B.ValueX()<<B.ValueY()<<endl;
      return 0;
}
```


第六章 作用域、生存期和可见性

6.1作用域及可见性

- **一个标识符的有效区域。**
- **分为：函数原型作用域、块作用域、类作用域、文件作用域。**

函数原型作用域

- 函数原型中所作的参数声明在该作用域，始于左括号，结束于右括号。

```
double Area(double radius);
```

```
double Area(double);
```


块作用域

- 块作用域由一对花括号定义。（局部作用域）

```
void fun(int a)
{
   int b(a);
   cin>>b;
   if(b>0)
   {
      int c;
      …
   } //c 的作用域结束
} //a,b 的作用域结束
```


块作用域

```
if(int i=f())
    i=2*i;
else
    i=100;   //i 的作用域结束
cout<<i;   //错误
switch(int i=f())
{
    case 1:
    …
}       // i 的作用域结束
cout<<i;   //错误
```


类作用域

文件作用域

可见性

- 标识符是否可以引用。
- 在嵌套作用域中，内层作用域的标识符优先于外层作用域的标识符。
- C++中，标识符必须先声明后引用，在同一作用域中不能声明同样的标识符。

可见性举例

```
#include <iostream.h>
int I;
int main()
{
    I=5;
    {  int I;
       I=7;
       cout<<"I="<<I<<endl;}
    cout<<"I="<<I<<endl;
    return 0;
}
```


6.2对象的生存期

- 静态生存期
  - 与程序的运行期相同。这种对象的存储单元相对位置在整个程序的运行期间不变
  - 文件作用域或用static指定。
  - 若未显式指定初始值，为零。
- 动态生存期
  - 在函数内部（块作用域）声明的对象是动态生存期。
  - 若未显式指定初始值，为任意值。

对象的生存期举例

```
#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;
}
```

6.3静态成员函数和静态数据成员

126

数据共享的方法

- 数据存储在局部对象中，通过参数传递机制实现函数间的数据共享。
- 数据存储在全局对象中，通过全局对象进行数据交换。
- 封装在类中
- 全局对象存在的问题
  - 可见性无限制
  - 不能保证操作的合法性
- 解决上述问题的方法是类及其封装技术，相应提出静态数据成员和静态成员函数。

127

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;
}
```

128

举例

```
class A {
public:
    static void f( A a);
private:
    int x;
};
void A::f(A a)
{
    cout<<x; //错误，静态成员函数的实现中不能直接引用类中声明的非静态成员。
    cout<<a.x;
}
}
```

129

举例

```
class goods
{
public:
    goods(int w);
    ~goods();
    int Weight();
    static int TotalWeight();
private:
    int weight;
    static int totalWeight;
};
```

130

举例

```
goods::goods(int w){
    weight=w;
    totalWeight += w;}
goods::~goods(){
    totalWeight -= w ;}
int goods::Weight(){
    return weight;}
int goods::TotalWeight(){
    return totalWeight;
}
```

131

举例

```
#include <iostream.h>
int goods::totalWeight=0;
int main(){
    int w;
    cin>>w;
    goods g(w);
    cin>>w;
    goods g2(w);
    cout<<goods::TotalWeight()<<endl;
    return 0;
}
```

132

第七章 对象与指针

133

7.1指针

- 指针：对象的地址
- 指针对象：存放指针的对象
- 声明指针对象的语法：
  - T *标识符;
  - int *pointer;
- 一个指针对象被赋值为一个对象的地址后，称指针指向对象。

134

为指针对象引入的两种操作

- **& ——取地址操作**
- *** ——目标操作**

135

```
#include <iostream.h>
void swap(double* first,double* second);
int main()
{
    double a(12),b(22);
    swap(&a,&b);
    cout<<"a="<<a<<",b="<<b<<endl;
    return 0;
}
void swap(double * first,double* second)
{
    double temp;
    temp=*first;
    *first=*second;
    *second=temp;
}
```

136

```
#if !defined(_LOCATION_H)
#define   LOCATION_H
#include <iostream.h>
class Location{
public:
   Location();
   Location(int x,int y);
   ~Location();
   void Move(int x,int y);
   int GetX(){return X;}
   int GetY(){return Y;}
private:
   int X,Y;
};
#endif   //_LOCATION_H
```

137

```
Location::Location()
{  X=Y=0;
   cout<<"Default constructor called."<<endl;
}
Location::Location(int x,int y)
{  X=x;   Y=y;
   cout<<"constructor called."<<X<<","<<Y<<endl;}
Location::~Location()
{  cout<<"Destructor called."<<X<<","<<Y<<endl;}
void Location::Move(int x,int y)
{
   X=x;
   Y=y;
}
```

138

```
Int main()
{
   Location A(5,10);
   Location* ptr;
   Ptr=&A;
   Int x=ptr->GetX();
   Cout<<x<<endl;
   Return 0;
   }
   (*ptr).GetX()======ptr->GetX();
```

139

7.2数组和指针

- 数组：一组类型相同的对象
- 声明语法：
  - 类型　标识符[元素个数]；
- 数组元素可以用下标引用
  - 数组名[index];
- 数组元素也可以用指针引用

140

数组和指针

- 通过指针引用数组元素
  - 定义指针对象
  - 数组的地址赋值给指针对象。
- 指针引用数组元素的方法
  - *(pointer + n)
  - pointer[n]

141

对象数组

- 没有为数组指定显式初始值时，数组元素使用缺省值初始化，当定义对象数组时，类中必须定义一个缺省的构造函数。

```
Int main()
{   cout<<Entering main…<<endl;
    Location A[2];
    For(int I=0;I<2;I++_
        A[I].Move(I+10I+10);
    Cout<<"exiting main…"<<endl;
    Return 0;
}
```

142

举例

```
Entering Main…
Default constructor called
Default constructor called
Exiting main
Destructor called
Destructor called
```

143

7.3引用

- 是标识对象的一种机制（是对象的一个别名）
- 一般用作参数类型、函数返回类型。
- 一般语法如下：
  - 基本类型& 标识符
- 一个引用不是独立的对象，只是对另一个对象的引用。
  - 基本类型& 标识符=对象；


```
int main()
{
    int a[]={1,3,5,7,9};
    int i;
    cin>>i;
    int& ref=a[i];
    ref=55;
    cout<<ref<<endl;
    return 0;
}
```



```
int& f(int index,int a[])
{   int& r=a[index];
    return r;
}
int main()
{   int a[]={1,3,5,7,9};
    f(2,a)=55;
    cout<<f(2,a);
    return 0;
}

int& f(int index,int a[])
```


栈对象与堆对象

- 在程序运行中根据需要在堆内存中创建的对象，使用完成后随时被删除。
- 创建语法如下：
  - T* ptr=new T(初始值列表);
- 删除语法如下：
  - delete ptr;
- 数组对象的创建
  - int* ptr=new int[10];
- 数组对象的删除 delete[] ptr;


```
#include <iostream.h>
#include <except.h>
#include "location.h"
int main()
{   int *ptr;
    try{
        ptr=new int[100];
    }
    catch(xalloc & e){
        cout<<"out of mem"<<endl;
        return -1;
    }
    for(int i=0;i<100;i++)
```



```
class Set{
public:
    Set(int sz=100);
    ~Set();
    void Copy(Set* s);
private:
    int *elems,size,PC;};
Set::Set(int sz)
{elems=new int[size=sz];
 PC=0;}
Set::~Set()
{delete[] elems;}
void Set::Copy(Set* s)
{PC=s->PC;delete[] elems;elems=new int[size=s->size];
 for(int I=0;I<PC;I++)
```



```
#include <iostream.h>
#include "location.h"
int main()
{cout<<"Step one:"<<endl;
 Location* ptr1(new Location);
 Delete ptr1;
 Cout<<"Step two:"<<endl;
 Ptr1=new Location(1,2);
 Delete ptr1;
 Return 0;
}
Step one:
Default constructor called.
Destructor called.1,2
Step two:
Constructor called.1,2
Destructor called.1,2
```


7.4指针数组

- 数组元素的类型是指针，这样的数组称指针数组。
- 语法：
  - T* 标识符[元素个数];

多维数组与指针

- 多维数组可以解释为其元素是多维数组的一维数组。

```
void f(int (*p)[3],int n);
int main()
{   int a[2][3]={{1,2,3},{4,5,6}};
    f(a,2);
    return 0;
}
void f(int (*p)[3],int n)
{   int i;
    int   **pp = *p+1;
    for(i=0;i<3;i++)
        cout<<pp[I]<<endl;
    int (*q)[3]=p+1;
    int **pq = *q+1;
    for(I=0;I<3;I++)
        cout<<pq[I]<<endl;
    for(I=0;I<n;I++)
        for(int j=0;j<3;j++)
```

7.5this指针

- 一个成员函数被调用时，被自动传递一个隐含的参数，该参数是一个指向正被该函数操作的对象的指针，在程序中可以用this引用它，因此称其为this指针。

```
Class Location{
public:
    Location(int xx,int yy){X=xx;Y=yy;}
    Void Assign(Location& p);
    Int GetX(){return X;}
    Int GetY(){return Y;}
Private:
    Int X,Y;
};
void Location::Assign(Location& p)
{   if(this!=&p)
        X=p.X;
        Y=p.Y;
}
```


7.6聚合类类型

- 用内部包含子对象的类实现复杂的类
- 子对象的初始化


```
#if !defined(_PART_H)
#define _PART_H
class Part{
public:
    Part();
    Part(int i);
    ~Part();
    void Print();
private:
    int val;
};
```

```
#include "part.h"
Part:Part()
{   val=0;
    cout<<"Default constructor of Part"<<endl;
}
Part::Part(int i)
{   val=i;
    cout<<"Constructor of Part"<<val<<endl;
}
Part::~Part()
{   cout<<"Destructor of Part"<<val<<endl;
}
void Part::Print()
```



```
#if !defined(_WHOLE_H)
#define _WHOLE_H
#include "part.h"
class Whole{
public:
    Whole();
    Whole(int I,int j,int k);
    ~Whole();
    void Print();
private:
    Part one;
    Part two;
    Int date;
};
```

```
#include "whole.h"
Whole::Whole()
{   date=0;
    cout<<"Default constructor of Whole"<<endl;
}
Whole::Whole(int I,int j,int k):two(I),one(j),date(
{   cout<<"Constructor of Whole"<<endl;
}
Whole::~Whole()
{   cout<<"Destructor of Whole"<<endl;
}
void Whole::Print()
```



```
#include "whole.h"
int main()
{ Whole anObject(5,6,10); //Whole anObject;
  AnObject.Print();
  return 0;}
Constructor of Part5
Constructor of Part6
Constructor of Whole
6
5
10
Destructor of Whole
Destructor of Part5
Destructor of Part6
```


第八章 运算符重载

- **重载的运算符是具有特殊名字的函数：其名字由operator和运算符共同定义；**
- **重载运算符也包含：返回类型、参数列表和函数体；**
- **重载运算符可作为成员函数或友元函数；**
- **作为成员函数时，其第一个参数为当前对象this；**
- **具体语法如下：**
  返回值类型 operator运算符名称（参数列表）

- **当使用运算符来表达对对象的某个操作时，可以将运算符函数声明为类的成员函数或类的友员函数。**
- **运算符的参数个数、优先级、结合性和使用语法是由C++规定的，不能被改变。**
- **重载运算符的形参不能具有缺省值。**
- **重载运算符的返回值类型任意，但最好不是void 。**

运算符重载规定

- **不建议重载的运算符：**
  - **&& || 逗号运算符**
  - **取地址运算符**

成员函数 or 友元函数

- **赋值、下标、调用、成员访问箭头必须为成员函数；**
- **改变对象状态的运算符一般为成员函数；**
- **具有对称性的运算符如算术、相等性、关系和位运算符，一般为友元函数；**

8.1重载为类的成员函数

```
#include <iostream.h>
class complex{
public:
    complex(double r=0,double i=0);
    complex operator+(const complex& c);
    complex operator -(const complex& c);
    complex operator –();
    void print()const;
private:
    double real,imag;
};
```


重载为类的成员函数

```
Complex::complex(double r,double i)
{real=r;imag=i;}
complex complex::operator + (const complex& c)
{   double r=real+c.real;
    double i=imag+c.imag;
    return complex(r,i);
}
complex complex::operator – (const complex& c)
{   double r=real-c.real;
    double i=imag-c.imag;
    return complex(r,i);
}
```

```
complex complex::operator + (const complex& c)
{   double r=real+c.real;
    double i=imag+c.imag;
    return complex(r,i);
}
complex complex::operator – (const complex& c)
{
    double r=real-c.real;
    double i=imag-c.imag;
    return complex(r,i);
}
```


重载为类的成员函数

```
complex complex::operator –()
{
    return complex(-real,-imag);
}
void complex::print()const
{
    cout<<"(real<<","<<imag<<")"<<endl;
}
```


重载为类的成员函数

```
int main()
{
    complex c1(2.5,3.7),c2(4.2,6.5);
    complexc ;
    c=c1-c2;        //c1.operator –(c2)
    c.print();
    c=c1+c2;        //c1.operator +(c2)
    c.print();
    c=-c1;          //c1.operator –()
    c.print();
    return 0;
}
```


8.2重载为类的友元函数

```
#include <iostream.h>
class complex{
public:
    complex(double r=0,double i=0);
    friend complex operator+(const complex& c1,const complex& c2);
    friend complex operator -(const complex& c1,const complex& c2);
    friend complex operator –(const complex& c);
    void print()const;
private:
    double real,imag;
};
```


重载为类的友元函数

```
Complex::complex(double r,double i)
{
    real=r;imag=i;
}
complex complex::operator + (const complex& c1,const complex& c2)
{
    double r=c1.real+c2.real;
    double i=c1.imag+c2.imag;
    return complex(r,i);
}
complex complex::operator – (const complex& c1,const complex& c2)
{
    double r=c1.real-c2.real;
    double i=c1.imag-c2.imag;
    return complex(r,i);
}
```


重载为类的友元函数

```
complex complex::operator –(const complex& c)
{
    return complex(-c.real,-c.imag);
}
void complex::print()const
{
    cout<<"(real<<","<<imag<<")"<<endl;
}
```


重载为类的友元函数

```
int main()
{
    complex c1(2.5,3.7),c2(4.2,6.5);
    complexc ;
    c=c1-c2;     //operator – (c1,c2)
    c.print();
    c=c1+c2;    //operator + (c1,c2)
    c.print();
    c=-c1;       //operator – (c1)
    c.print();
    return 0;
}
```


8.3例子

两种方法的比较

- 下标运算符只能被重载为成员函数
- 函数调用运算符可以看作是下标运算符的扩展

第九章 对类的进一步讨论

- 类如何控制其对象的拷贝、赋值及销毁；
- 需要定义拷贝构造函数、赋值运算和析构函数；

拷贝构造函数

```
class CTest{
public:
    CTest();
    //拷贝构造函数，其第一个参数是自身的引用，且其他参数（若有的话），都有缺省值；
    CTest(const CTest& op);
    void func1(){cout<<"this "<<val<<endl;}
private:
    int val1,val2;
};
```

- 若没有定义，则编译器会提供一个合成拷贝构造函数，若类已经定义，则不再提供合成构造函数；
- 即使类定义了其他形式的构造函数（未定义拷贝构造函数），编译器也会提供一个拷贝构造函数；

拷贝构造函数

```
Ctest::CTest(const CTest& op)
{
    val1=op.val1;
    val2=op.val2;
}
```

- 使用=定义对象时；
- 将一个对象作为实参传递给值传递的形参；
- 从一个返回值为对象值的函数返回对象；
- 使用花括号列表初始化数组或聚合类成员；

拷贝构造函数

```
#include <iostream>
using namespace std;
class CTest{
public:
    CTest(int v1, int v2){val1=v1;val2=v2;}
    CTest(const CTest& op);//拷贝构造函数
    void func1(){cout<<"this "<<val1<<endl;}
private:
    int val1,val2;
};
//CTest::CTest()=default;
CTest::CTest(const CTest& op)
{
    val1=op.val1;
    val2=op.val2;
}
int main()
{
    CTest obj(1,2);
    CTest obj2(obj);//调用拷贝构造函数
    CTest obj3=obj2; //调用拷贝构造函数
    obj2.func1();
    return 0;
}
```


拷贝构造函数

```
#include <iostream>
using namespace std;
class CTest{
public:
    CTest(int n){
    if(n<0)
        n=16;
    m_nSize=n;m_pBuf=new int[n_nSize];}
    void Assign(int val){for(int n=0;n<m_nSize;n++)m_pBuf[n]=val;}
    CTest(const CTest& op);//拷贝构造函数
    void print(){for(int n=0;n<m_nSize;n++) cout<<m_pBuf[n]<<endl;}
private:
    int* m_pBuf, m_nSize;
};
CTest::CTest(const CTest& op)
{
    m_nSize=op.m_nSize;m_pBuf=op.m_pBuf;
}
int main()
{  CTest obj(4);
    CTest obj2(obj);
    CTest obj3=obj2;
    obj2.Assign(1);
    obj2.print();////
    obj3.print();////经拷贝构造后，obj2和obj3及obj的m_pBuf在内存中指向同一个地方
    return 0;}
```


赋值运算

```
class CTest{
public:
    CTest();
     //赋值运算，其参数是自身的引用
    Ctest& operator=(const CTest& op);
    void func1(){cout<<"this "<<val<<endl;}
private:
    int val1,val2;
};
```

- 若没有定义，则编译器会提供一个赋值运算，若类已经定义，则不再提供合成赋值运算；

赋值运算

```
Ctest& Ctest::operator=(const CTest& op)
{
    val1=op.val1;
    val2=op.val2;
    return *this;
}
```

- 使用=时调用该运算符；

第十章 模板

- 模板定义
- 模板实参推断
- 重载与模板
- 可变参数模板
- 模板特例化

10.1 模板定义

```
int Compare(const string& v1,const string& v2)
{
      if(v1<v2) return -1;
      if(v2<v1) return 1;
      return 0;
}
int Compare(const double& v1,const double& v2)
{
      if(v1<v2) return -1;
      if(v2>v1) return 1;
      return 0;
}
```


10.1.1 函数模板定义

针对上述情况，可以定义函数模板如下：

```
template<typename T>
int Compare(const T& v1,const T& v2)
{
    if(v1<v2) return -1;
    if(v2<v1) return 1;
    return 0;
}
```

- 关键字template,<参数列表>；
- 使用模板时，隐式或显式地指定模板实参，将其绑定到模板参数上；

实例化函数模板

- 调用函数模板时，编译器用函数实参来推断模板实参，如：

```
cout<<Compare(1,0)<<endl;
//推断模板参数为int，并绑定到T
vector<int> v1{1,2,3},v2{4,5,6};
cout<<Compare(v1,v2)<<endl;
//推断模板参数为vector<int>，并绑定到T
```

- 编译器用推断出的模板实参实例化一个特定版本的函数，类似如下效果：

```
int Compare(const int& v1,const int& v2)

int Compare(const vector<int>& v1,const vector<int>& v2)
```


类型参数和非类型参数

- 类型参数可以看作类型说明符，类型参数前必须以typename 关键字说明，如：

  template<typename T,class U>

  T和U均为类型参数，表示不同类型；
- 非类型参数表示一个值而不是类型，通过特定类型名定义，而不用typename，如：

  template<unsigned N,unsigned M>

  N和M表示2个数值（整型）；

非类型参数

```
template<unsigned N,unsigned M>
int Compare(const char (&p1)[N],const char (&p2)[M])
{
    return strcmp(p1,p2);
}
Compare("Hi","ABC");//N=3,M=4
```

- 非类型参数可以是整型、指针或引用
- 绑定到非类型整型参数的必须是常量表达式；
- 绑定到非类型指针或引用参数的实参必须是具有静态生存期的；
- 函数模板可以声明为inline，如下：

```
template<unsigned N,unsigned M> inline
int Compare(const char (&p1)[N],const char (&p2)[M])
{
   return strcmp(p1,p2);
}
```

- 要特别注意的是，只有当使用模板（实例化）时编译器才生成代码会导致代码错误检测的时机延后；

10.1.2 类模板定义

```
#include <iostream>

using namespace std;

template<typename T,unsigned SZ=16>
class CStack{
public:
    CStack();
    int size(){return m_size;}
    bool empty(){return &_top==m_bottom?true:false;}
    void push(T ele);
    T pop();
    void print(){while(!empty())
                 cout<<pop()<<endl;}
private:
    T data[SZ];
    int m_top;
    int m_bottom;
    int m_size;//stack size
};
```


10.1.2 类模板定义

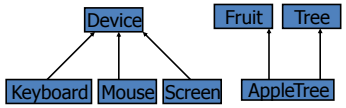
```
template<typename T,unsigned SZ>
void CStack<T,SZ>::push(T ele)
{
    if(m_top<m_size)
        data[m_top++]=ele;
}

template<typename T,unsigned SZ>
CStack<T,SZ>::CStack()
{
    m_size=SZ;
    m_top=m_bottom=0;
}
template<typename T,unsigned SZ>
T CStack<T,SZ>::pop()
{
    if(m_top>m_bottom)
        return data[--m_top];
    return 0;
}
int main()
{
    CStack<int,10> aStack;
    aStack.push(1);
    aStack.push(2);
    aStack.push(3);
    aStack.push(4);
    aStack.print();
    return 0;
}
```


第十章结束

- 结束

第十一章 继承与多态性

- 通过已有的类进行扩展产生新类的过程
- 产生的新类称派生类
- 产生派生类的类称基类或父类。
- 从一个基类派生称单基继承，从多个基类派生称多基继承。



11.1继承

单基继承的语法

class 派生类名：访问控制 基类名

{

数据成员和成员函数声明；

}

多基继承的语法

class 派生类名：访问控制 基类名1，访问控制 基类名2…

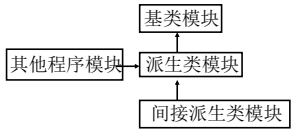
{

数据成员和成员函数声明；

}

访问控制

- 访问控制用于控制基类中声明的名字在多大的范围内能够被派生类的用户访问
- 访问控制有三种public,protected,private




```
Class Location{
Public:
    Void InitL(int xx,int yy);
    Void Move(int xOff,int yOff);
    Int GetX(){return X;}
    Int GetY(){return Y;}
Private:
    Int X,Y;
};
void Location::InitL(int xx,int yy)
{  X=xx; Y=yy;}
void Location::Move(int xOff,int yOff)
{  X+=xOff;   Y+=yOff;}
```

```
Class Rectangle:public Location{
Public:
    Void InitR(int x,int y,int w,int h);
    Int GetW(){return W;}
    Int GetH(){return H;}
Private:
    Int W,H;
};
void Rectangle::InitR(int x,int y,int w,int h)
{
    InitL(x,y);
    W=w;
    H=h;
}
```


公有继承

```
#include <iostream.h>
int main()
{
   Rectangle rect;
   Rect.InitR(2,3,20,10);
   Rect..Move(3,2);
   Cout<<rect.GetX()<<","
       <<rect.GetY()<<","
     <<rect.GetW()<<","
     <<rect.GetH()<<endl;
   return 0;
}
```


私有继承

```
Class Rectangle:private Location{
Public:
    Void InitR(int x,int y,int w,int h);
    Void Move(int xOff,int yOff);
    Int GetX(){return Location::GetX();}
    Int GetY(){return Location::GetY();}
    Int GetW(){return W;}
    Int GetH(){return H;}
Private:
    Int W,H;
};
```

```
void Rectangle::InitR(int x,int y,int w,int h)
{
    InitL(x,y);
    W=w;
    H=h;
}
void Rectangle::Move(int xOff,int yOff)
{
    Location::Move(xOff,yOff);
}
```


在派生类中访问基类成员

```
Class V:public Rectangle{
Public:
    Void Function();
};
void V::Function()
{   Move(3,2);    }
int main()
{   Rectangle rect;
    Rect.Move(3,2);
}
```


保护的成员

- 对V模块是public
- 对H模块是private

```
Class A{
Protected:
    Int X;};
int main()
{  A a;
   a.X=5;  //错误}
class B:public A{
public:
    void Function();};
void B::Function()
{  X=5;}
```


多继承

- 可以将其视为单继承来考察，每个基类与派生类的关系可以当做单基继承。

11.2构造函数和析构函数

- 派生类的数据成员由属于基类的部分和属于自己的部分组成，属于基类的部分初始化应由基类的构造函数完成。
- 构造函数不被继承，派生类必须调用基类的构造函数来初始化属于基类的部分数据成员。
- 一般形式如下：
  - C::C(参数表0):C1(参数表1),C2(参数表2),...,Cn(参数表n)
  - {...}

构造函数和析构函数

- 如果冒号右边的某个基类的构造函数参数表为空，则可以省略该项。
- 构造对象时，先执行基类的构造函数，然后执行派生类的构造函数。析构时则相反。
- 多基继承时基类构造函数的执行顺序由继承时的顺序决定（从左向右），与初始化列表中的顺序无关。

构造函数和析构函数

```
#include <iostream.h>
class B{
public:
   B();
   B(int I);
   ~B();
   void Print() const;
private:
   int b;
};
```

```
B::B()
{  b=0;
   cout<<"B's default constructor called."<<endl;
B::B(int I)
{  b=I;
   cout<<"B's constructor called."<<endl;}
B::~B()
{  cout<<"B's destructor called."<<endl;}
void B::Print() const
{  cout<<b<<endl;}
```


构造函数和析构函数

```
class C:public B{
public:
   C();
   C(int I,int j);
   ~C();
   void Print() const;
private:
   int c;
};
```

```
C::C()
{  c=0;
   cout<<"C's default constructor called."<<endl;}
C::C(int I,int j):B(i)
{  c=j;
   cout<<"C's constructor called."<<endl;}
C::~C()
{  cout<<"C's destructor called."<<endl;}
void C::Print() const
{  B::Print();
   cout<<c<<endl;}
```


构造函数和析构函数

```
Int main()
{
   C obj(5,6);
   Obj.Print();
   Return 0;
}
```

```
B's constructor called.
C's constructor called.
5
6
C's destructor called.
B's destructor called.
```


11.3二义性

- 如果派生类的一个表达式引用基类中不止一个成员，则存在二义性。
- 解决二义性的办法是加类限定符。

二义性

```
class A{
public:
    void f();};
class B{
public:
    void f();
    void g();};
class C:public A,public B{
public:
    void g();
    void h();
};
```

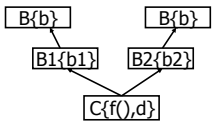
```
C obj;
obj.f();   //是 A 的 f()或 B 的 f()?
解决的办法是在类 C 中定义一个 f()成员函数
void C::f()
{
   A::f();
   或
   B::f();
}
```


二义性

- 如果一个派生类是从多个基类派生，而这些基类又有一个共同的基类，则在对该基类中声明的标识符进行访问时可能产生二义性。

```
class B{
public:
   int b;};
class B1:public B{
private:
   int b1;};
class B2:public B{
private:
   int b2;};
class C:public B1,public B2{
public:
   int f();
private:
   int d;};
```


二义性



二义性

```
C c;
c.b;   //错
c.B::b ;//错
c.B1::b;//对
c.B2::b; //对
int main()
{  C c;
   c.B1::b=5;
   c.B2::b=10;
   cout<<"path B1➔"<<c.B1::b<<endl;
   cout<<"path B2➔"<<c.B2::b<<endl;
   return 0;
}
```


虚基类

- 为避免一个基类在多条继承路径上造成在派生类中产生多个基类子对象，可以采用虚基类技术。
- 语法如下
  - class B1:virtual public B{}
  - class B2:virtual public B{}
  - class C:public B1,public B2{}

派生类的赋值和初始化

- 如未定义拷贝构造函数和赋值操作，则编译器提供。
- 如自定义，则应在实现中调用基类的拷贝构造函数和赋值操作。

11.4 虚函数与多态性

- **如果类型S是从类型T共有继承的，则称类型S是类型T的子类型。**
- **类型S的指针或引用可以适用于类型T的指针或引用的场合。**

动态绑定与虚函数

- **由于在基类和派生类中，相同的名字执行的操作可能不同，所以基类的成员函数在派生类中经常有不同的实现。**
- **用virtual关键字可以实现动态绑定。**
- **动态绑定只适用于虚函数的参数是指针或引用的情况。**

动态绑定与虚函数

```
class Shape{
public:
   Shape(double x,double y);
   Double Area();
private:
   double X,Y;};
Shape::Shape(double x,double y)
:X(x),Y(y){}
double Shape::Area()
{  return 0.0;}
```

```
class Rectangle:public Shape{
public:
   Rectangle(double x,double y,
             double w,double h);
   double Area() const;
private:
   double W,H;};
Rectangle::Rectangle(double x,double y,
   double w,double h):Shape(x,y),W(w),H(h
{}
double Rectangle::Area()const
{  return W*H;}
```


动态绑定与虚函数

```
#include <iostream.h>
void fun(const Shape& s)
{  cout<<s.Area()<<endl;}
int main(0
{
   Rectangle rect(2.0,5.0,10.0,20.0);
   fun(rect);
   return 0;
}
```


动态绑定与虚函数

```
class Shape{
public:
   Shape(double x,double y);
   Virtual double Area() const;
private:
   double X,Y;};
Shape::Shape(double x,double y)
:X(x),Y(y){}
double Shape::Area() const
{  return 0.0;}
```

```
class Rectangle:public Shape{
public:
   Rectangle(double x,double y,double w
                          ,double h);
   Virtual double Area() const;
private:
   double W,H;};
Rectangle::Rectangle(double x,double y,
   double w,double h):Shape(x,y),W(w),H(
{}
double Rectangle::Area()const
{  return W*H;}
```


动态绑定与虚函数

- **在一个成员函数内调用虚函数时，对该虚函数的调用进行动态绑定。**
- **派生类中虚函数必须满足下列条件：**
  - **与基类的函数有相同个数的参数**
  - **参数类型相同**
  - **返回类型或者与基类相同或者返回指针或引用，且返回的指针或引用的基类型是基类中对应函数所返回的指针或引用的基类型的子类型。**
- **在一个成员函数内调用虚函数时，对该虚函数的调用进行静态绑定。**

纯虚函数与抽象类

- 当定义基类时某个虚函数不能给出确定的实现时，可以定义其为纯虚函数。
- 语法如下
  - virtual 类型 函数名(参数列表)=0;
- 具有纯虚函数的类称为抽象类，
- 抽象类只能用来派生类而不能定义具体的对象。

纯虚函数与抽象类

```
class Shape{
public:
   virtual double Area() const=0;
};
class App{
public:
   double Compute(Shape* s[],int n) const;
};
double App::Area(Shape* s[],int n) const
{
   double sum=0;
   for (int I=0;I<n;I++)
      sum+=s[I]->Area();
   return sum;
}
```