基于 Linux 的 C++

第九讲　类与对象

提　纲

程序抽象与面向对象

- 类与对象的概念与意义

类类型

对　象

类与对象的成员

继　承

多　态

程序抽象与面向对象

抽象数据类型：设计能够存储二维平面上点的抽象数据类型

```
/*  点库接口“point.h”*/

struct POINT;
typedef struct POINT * PPOINT;

PPOINT PtCreate( int x, int y );
void PtDestroy( PPOINT point );
void PtGetValue( POINT point, int * x, int * y );
void PtSetValue( PPOINT point, int x, int y );
bool PtCompare( PPOINT point1, PPOINT point2 );
char * PtTransformIntoString( PPOINT point );
void PtPrint( PPOINT point );
```

程序抽象与面向对象

```
/*  点库实现“point.cpp”*/

#include <cstdio>
#include <cstring>
#include <iostream>
#include "point.h"

using namespace std;

static char* DuplicateString( const char* s );
struct POINT{  int x, y;  };

PPOINT PtCreate( int x, int y )
{
  PPOINT t = new POINT;
  t->x = x,    t->y = y;
  return t;
}
```

程序抽象与面向对象

```
void PtDestroy( PPOINT point )
{
  if( point ){  delete  point;  }
}

void PtGetValue( PPOINT point, int * x, int * y )
{
  if( point )
  {
    if( x )    *x = point->x;
    if( y )    *y = point->y;
  }
}

void PtSetValue( PPOINT point, int x, int y )
{
  if( point ){  point->x = x,    point->y = y;  }
}
```

程序抽象与面向对象

```
bool PtCompare( PPOINT point1, PPOINT point2 )
{
  if( !point1 || !point2 )
  {
    cout << "PtCompare: Parameter(s) illegal." << endl;
    exit( 1 );
  }
  return point1->x == point2->x && point1->y == point2->y;
}

void PtPrint( PPOINT point )
{
  if( point )
    printf( "(%d,%d)", point->x, point->y );
  else
    printf( "NULL" );
}
```

程序抽象与面向对象

```
char * PtTransformIntoString( PPOINT point )
{
  char buf[BUFSIZ];
  if( point )
  {
    sprintf( buf, "(%d,%d)", point->x, point->y );
    return DuplicateString( buf );
  }
  else  return "NULL";
}

char* DuplicateString( const char* s )
{
  unsigned int n = strlen(s);
  char* t = new char[n+1];
  for( int i=0; i<n; i++)    t[i] = s[i];
  t[n] = '\0';
  return t;
}
```

类与对象的概念与意义

类的概念与意义

- 属性与行为的辩证统一

程序抽象

- 数据封装、信息隐藏
- 如果没有类的概念，无法定义非指针量，且控制性不佳

对象的概念与意义

- 量
- 对象行为的主动性

点库接口

```
// 想象的代码，非C++标准实现
struct POINT
{
  int x, y;    // 公开量，不符合数据信息隐藏规则

  Create( int x, int y );
  void Destroy();
  void GetValue( int * x, int * y );
  void SetValue( int x, int y );
  bool Compare( const POINT* point );
  char* TransformIntoString();
  void Print();
};
```

类类型

类的声明与定义

示　例

- 点类库
- 圆类库

关于类声明与定义的说明

类的声明与定义

类声明：仅声明类的存在，没有提供细节

- 关键字：class
- 示例：class A;

类定义

- 一般定义格式
- 类成员：数据与函数
- 三个保留字顺序任意
- public：其后成员公开
- protected：其后成员有限公开
- private：其后成员私有，仅本对象可直接访问

```
class A
{
public:
  成员类型    成员名称;
protected:
  成员类型    成员名称;
private:
  成员类型    成员名称;
};
```

点类库接口

```
/*  点类库接口“point.h”*/

class Point
{
public:
  Point( int x, int y );
  ~Point();
  void GetValue( int * x, int * y );
  void SetValue( int x, int y );
  bool Compare( const Point & point );
  char* TransformIntoString();
  void Print();
private:
  int x, y;
};
```

圆类库接口

设计表示二维平面上圆的类类型

```
/*  圆类库接口“circle.h”*/

class Circle
{
public:
  void GetOrigin( double * x, double * y );
  void SetOrigin( double x, double y );
  double GetRadius();
  void SetRadius( double r );
  double GetPerimeter();
  double GetArea();
private:
  double r, x, y;
};
```

圆类库实现

```
/*  圆类库实现“circle.cpp”*/

#include "circle.h"

const double pi = 3.141592653589793;

void Circle::GetOrigin( double * x, double * y )
{
  *x = this->x;
  *y = this->y;
}

void Circle::SetOrigin( double x, double y )
{
  this->x = x;
  this->y = y;
}
```

this指针：指向当前对象的指针，由系统自动定义

圆类库实现

```
double Circle::GetRadius()
{
  return r;
}

void Circle::SetRadius( double r )
{
  this->r = r;
}

double Circle::GetPerimeter()
{
  return 2 * pi * r;
}

double Circle::GetArea()
{
  return pi * r * r;
}
```

类类型的声明

仅限于函数原型使用类类型的声明

不能用于定义类的数据成员

示　例

```
class B;
class A
{
public:
  void func( B b );  // 正确
private:
  B b;  // 错误
};
class B{ … };
```

对　象

对象的定义与使用

对象的构造

对象的析构

对象数组

对象的定义与使用

对象的定义

- 像结构体一样定义和使用对象及其公开的成员
- 私有成员不可在对象外部直接访问

对象示例

```
/* 源文件“main.cpp” */
int main()
{
  Circle circle;
  circle.SetOrigin( 0.0, 0.0 );
  circle.SetRadius( 1.0 );
  cout << "Perimeter: " << circle.GetPerimeter() << endl;
  cout << "Area: " << circle.GetArea() << endl;
  return 0;
};
```

对象的构造

对象构造的意义

- 构造就是初始化，在定义对象时初始化其数据成员

对象构造的技术手段：使用构造函数

- 与类类型同名，没有返回值类型（包括void类型）
- 构造函数允许重载
- 构造函数可以带缺省参数，但是不建议
- 至少公开一个构造函数
- 只能由系统在创建对象时自动调用，程序其他部分不能直接调用

圆类库接口

设计表示二维平面上圆的类类型

```
/*  圆类库接口“circle.h”*/

class Circle
{
public:
  Circle();
  //Circle( double r, double x = 0.0, double y = 0.0 );    // 缺省参数，不建议
  Circle( double r, double x, double y );
  ……
private:
  double r, x, y;
};
```

圆类库实现

```
/*  圆类库实现“circle.cpp”*/

Circle::Circle()
{
  r = 0.0,    x = 0.0,    y = 0.0;
}

Circle::Circle( double r, double x, double y )
{
  this->r = r,    this->x = x,    this->y = y;
}

/*  主程序“main.cpp”  */

int main()
{
  double r = 1.0, x = 0.0, y = 0.0;
  Circle circle( r, x, y );
}
```

缺省构造函数

类没有明确的构造函数

- 系统自动产生一个缺省构造函数，自动调用
- 缺省构造函数无参数，且函数体中没有任何语句
- 如果定义了任意一个构造函数，则不再生成缺省构造函数

缺省构造函数调用示例

- 正确示例：Circle circle;
- 错误示例：Circle circle();
- 在构造函数无参数时，不能使用函数形式构造对象。原因？

拷贝构造函数

拷贝构造函数用于构造已有对象的副本

拷贝构造函数单参数，为本类的常对象的引用

如未定义，系统自动产生一个缺省拷贝构造函数

缺省拷贝构造函数为位拷贝（浅拷贝），如需深拷贝（例如成员为指针），需自行定义

```
/* 圆类库接口“circle.h” */
class Circle
{
public:
  Circle( const Circle & that );
  ......
private:
  double r, x, y;
};
```

```
/* 圆类库实现“circle.cpp” */
Circle::Circle( const Circle & that )
{
  this->r = that.r;
  this->x = that.x;
  this->y = that.y;
};
```

构造函数的初始化列表

初始化列表的格式

```
class A
{
public:
  A( int a );
private:
  int a;
};

class B
{
public:
  B( int a, int b );
private:
  A a;
  int b;
};
```

```
A::A( int a ) : a(a)
{
}

B::B( int a, int b ) : a(a), b(b)
{
}
```

构造函数的初始化列表

初始化列表的目的与意义

- 在构造对象时，同步构造内部对象
- 部分成员（常量与引用）只能初始化，不能赋值
- 部分成员（类的对象）如果赋值，将导致两次构造
  - 在分配内存时，调用缺省构造函数构造，然后执行构造函数体内的赋值语句再次构造，效率不佳
  - 若类没有缺省构造函数，则会导致问题

注意事项

- 成员初始化按照成员定义顺序，而不是初始化列表顺序
- 必须保持初始化列表和成员定义的顺序一致性，但允许跳过部分成员；否则后续成员可能不会正确初始化

对象的析构

对象析构的意义

- 析构就是终止化，在对象生命期结束时清除它

对象析构的技术手段：使用析构函数

- 与类类型同名，前有“~”记号，无返回值类型（包括void类型），无参数
- 析构函数必须是公开的
- 可以由系统在销毁对象时自动调用，也可以由程序其他部分直接调用，但两者工作原理不同
- 每个类只能有一个析构函数
- 若未定义，系统会自动产生一个缺省析构函数，该函数无代码

圆类库接口

设计表示二维平面上圆的类类型

```
/*  圆类库接口“circle.h”*/

class Circle
{
public:
  Circle();
  Circle( double r, double x, double y );
  ……
 ~Circle();
private:
  double r, x, y;
};
```

对象的析构

定义析构函数的目的

- 用于释放对象中动态分配内存的目标数据对象

使用示例

```
class A{
public:
  A( int x );
  ~A();
private:
  int * p;
};
```

```
A::A( int x )
{
  p = new int;
  *p = x;
}
A::~A()
{
  delete p,  p = NULL;
}
```

对象数组

对象数组

- 像普通数组一样定义和使用

对象数组的初始化

- 当构造函数单参数时，像普通数组一样构造所有元素
- 当构造函数多参数时，使用下述方法构造

```
Circle circles[2] = { Circle(1.0, 0.0, 0.0), Circle(2.0, 1.0, 1.0) };
```

类与对象的成员

- 内联函数
- 常数据成员
- 常成员函数
- 静态数据成员
- 静态成员函数
- 静态常数据成员
- 友元函数与友元类

内联函数

目的：程序优化，展开函数代码而不是调用

内联函数使用的注意事项

- 在函数定义前添加inline关键字，仅在函数原型前使用此关键字无效
- 编译器必须能看见内联函数的代码才能在编译期展开，因而内联函数必须实现在头文件中
- 在类定义中给出函数体的成员函数自动成为内联函数
- 函数体代码量较大，或包含循环，不要使用内联
- 构造函数和析构函数有可能隐含附加操作，慎用内联
- 内联函数仅是建议，编译器会自主选择是否内联

内联函数

```
/*  圆类库接口“circle.h”*/

inline void Circle::GetOrigin( double * x, double * y )
{
  *x = this->x;
  *y = this->y;
}

inline void Circle::SetOrigin( double x, double y )
{
  this->x = x;
  this->y = y;
}
```

常数据成员

常数据成员：值在程序运行期间不可变

- 定义格式：const 类型 数据成员名称;
- 初始化：只能通过构造函数中的初始化列表进行

使用示例

```
class A
{
public:
  A( int a );
private:
  const int num;
};

A::A( int a ) : num(a) { ..... };
```

常成员函数

常成员函数：不能修改对象成员值的函数

- 定义格式：类型 成员函数名称(参数列表) const;
- 常成员函数不能调用类中非常成员函数
- 静态成员函数不能定义为常成员函数
- 如果对象为常量，则只能调用其常成员函数

使用示例

```
class Circle{
public:
  double GetArea() const;
  ......
};
double Circle::GetArea() const{ ...... }
```

静态数据成员

静态数据成员只有一份，由该类所有对象共享

- 声明格式：static 类型 静态数据成员名称;
- 仅声明，不在对象上分配空间
- 定义格式：类型 类名称::静态数据成员名称 = 初始值;
- 必须在外部初始化，初始化动作与访问控制无关

示　例

```
class A
{
private:
  static int count;
};

int A::count = 0;
```

静态成员函数

静态成员函数

- 在类而不是对象上调用
- 目的：访问类的静态数据成员，若要访问类的非静态数据成员，必须指定对象或者使用指向对象的指针

使用示例

```
class A
{
public:
  static int f();
  static int g( const A & a );
private:
  static int count;
  int num;
};
```

```
int A::count = 0;
int A::f()
{
  return count;
}
int A::g( const A & a )
{
  return a.num;
}
```

单子模式

只存在某类的单一共享对象

存在某种全局访问策略，以在需要时访问该对象

单子模式：无析构

```
class Singleton
{
public:  // 静态成员函数，对象不存在时构造，否则返回之，保证唯一性
  static Singleton * Get() {  if (!_s)  _s = new Singleton;  return _s;  }
  int GetData() { return ++a; }
private:  // 私有构造和析构函数，禁止在外部构造和析构本类的对象
  Singleton() { a = 0; }
  Singleton( const Singleton & that );  // 只声明不实现，禁止拷贝和赋值构造
  Singleton & operator=( const Singleton & that );
  ~Singleton();  // 只声明不实现，禁止析构
private:
  static Singleton * _s;  // 静态数据成员，指向本类唯一对象的指针
  int a;  // 验证数据
};

Singleton * Singleton::_s = NULL;  // 定义于源文件中

//  使用方法：以Singleton::Get()->GetData()方式直接访问
Singleton::Get()
```

单子模式：错误析构

```
class Singleton
{
public:  // 静态成员函数，对象不存在时构造，否则返回之，保证唯一性
  static Singleton * Get() {  if (!_s)  _s = new Singleton;  return _s;  }
  int GetData() { return ++a; }
private:  // 私有构造和析构函数，禁止在外部构造和析构本类的对象
  Singleton() { a = 0; }
  Singleton( const Singleton & that );  // 只声明不实现，禁止拷贝和赋值构造
  Singleton & operator=( const Singleton & that );
  //  错误析构函数，访问控制改为public也不行
  //  delete操作符本身需要调用析构函数
  //  且非静态函数不能释放静态指针成员，否则可能导致系统崩溃
  ~Singleton() {  if(Singleton::_s) {  delete Singleton::_s, Singleton::_s = NULL; }  }
private:
  static Singleton * _s;  // 静态数据成员，指向本类唯一对象的指针
  int a;  // 验证数据
};
```

单子模式：正确析构

```
class Singleton
{
public:  // 静态成员函数，对象不存在时构造，否则返回之，保证唯一性
  static Singleton * Get() {  if (!_s)  _s = new Singleton;  return _s;  }
  int GetData() { return ++a; }
private:  // 私有构造函数，禁止在外部构造本类的对象
  Singleton() { a = 0; }
  Singleton( const Singleton & that );  // 只声明不实现，禁止拷贝和赋值构造
  Singleton & operator=( const Singleton & that );
public:  // 在部分系统下使用private亦可，系统简单释放全部内存，并不调用它
  ~Singleton() { }  // 因此，如果函数非空（如需数据持久化），有可能导致问题
private:
  static Singleton * _s;  // 静态数据成员，指向本类唯一对象的指针
  // Destroyer类的唯一任务是删除单子
  class Destroyer{
  public:
    ~Destroyer() {  if(Singleton::_s) {  delete Singleton::_s, Singleton::_s = NULL;  } }
    };
  static Destroyer _d;  // 程序结束时，系统自动调用静态成员的析构函数
  int a;  // 验证数据
};
```

单子模式：正确析构

```
class Singleton
{
public:  // 静态成员函数，对象不存在时构造，否则返回之，保证唯一性
  static Singleton * Get() {  if (!_s)  _s = new Singleton;  return _s;  }
  // 不调用析构函数，Release调用时机由程序员确定
  static void Release() {  if(_s)  {  free(_s),  _s = NULL;  }  }
  int GetData() { return ++a; }
private:  // 私有构造和析构函数，禁止在外部构造和析构本类的对象
  Singleton() { a = 0; }
  Singleton( const Singleton & that );  // 只声明不实现，禁止拷贝和赋值构造
  Singleton & operator=( const Singleton & that );
  ~Singleton();
private:
  static Singleton * _s;  // 静态数据成员，指向本类唯一对象的指针
  int a;  // 验证数据
};
```

单子模式：静态数据

```
class Singleton
{
public:  // 静态成员函数中的静态变量，保证唯一性
  static Singleton & Get() {  static Singleton _s;  return _s;  }
  int GetData() { return ++a; }
private:  // 私有构造和析构函数，禁止在外部构造和析构本类的对象
  Singleton() { a = 0; }
  Singleton( const Singleton & that );
  Singleton & operator=( const Singleton & that );
  ~Singleton() { }
private:
  int a;  // 验证数据
};

//  本实现没有动态内存分配，因而无需销毁单子对象
//  使用方法：定义引用或以Singleton::Get().GetData()方式直接访问
Singleton & sing = Singleton::Get();
int n = sing.GetData();
```

静态常数据成员

静态常数据成员：值在程序运行期间不可变，且只有唯一副本

- 定义格式：static const 类型 数据成员名称;
- 初始化：只能在类的外部初始化

使用示例

```
class A
{
private:
  static const int count;
};

const int A::count = 10;
```

友元函数与友元类

友元：慎用！破坏类数据封装与信息隐藏

- 类的友元可以访问该类对象的私有与保护成员
- 友元可以是函数、其他类成员函数，也可以是类
- 定义格式：friend 函数或类声明;
- 两个类的友元关系不可逆，除非互为友元

使用示例

```
class Circle
{
  friend double Get_Radius();
  friend class Globe;  // 将Globe类所有成员函数声明为友元
private:
  double radius;
};
```

继　承

- 继承与派生
- 单继承
- 多继承
- 虚继承
- 派生类的构造函数与析构函数
- 类的赋值兼容性

继承与派生

继承的基本概念

- 类类型：描述分类的概念
- 继承：描述类之间的血缘（继承）关系
- 基类、派生类
- 父类、子类（不恰当的概念）

继承的意义

- 派生类拥有基类的全部属性与行为
- 派生类可以增加新的属性与行为

单继承

单继承的基本语法格式

- class 派生类名称： 派生类型保留字 基类名称 { … };

派生类型保留字

- public：基类的public、protected成员在派生类中保持，private成员在派生类中不可见（属于基类隐私）
- protected：基类的private成员在派生类中不可见，public、protected成员在派生类中变为protected成员
- private：基类的private成员在派生类中不可见，public、protected成员在派生类中变为private成员
- 设计类时若需要使用继承机制，建议将派生类需要频繁使用的基类数据成员设为protected的

单继承

```
// “Point.h”

#include <iostream>
using namespace std;

class Point
{
public:
  Point( int x = 0, int y = 0 ) : _x(x), _y(y) { }
  Point( const Point& pt ) : _x(pt._x), _y(pt._y) { }
  int GetX() const {  return _x;  }
  void SetX( int x ) {  _x = x;  }
  int GetY() const {  return _y;  }
  void SetY( int y ) {  _y = y;  }
  friend ostream& operator<<( ostream& os, const Point& pt );
protected:
  int _x, _y;
};
```

单继承

```
class Point3D: public Point
{
public:
  Point3D( int x = 0, int y = 0, int z = 0 ) : Point(x,y), _z(z) { }
  Point3D( const Point3D& pt3d ) : Point(pt3d._x, pt3d._y), _z(pt3d._z) { }
  int GetZ() const {  return _z;  }
  void SetZ( int z ) {  _z = z;  }
  friend ostream& operator<<( ostream& os, const Point3D& pt3d );
protected:
  int _z;
};
```

单继承

```
//  “point.cpp”
#include "point.h"

ostream& operator<<( ostream& os, const Point& pt )
{
  os << "( " << pt._x << ", " << pt._y << " )";
  return os;
}

ostream& operator<<( ostream& os, const Point3D& pt3d )
{
  os << "( " << pt3d._x << ", " << pt3d._y << ", " << pt3d._z << " )";
  return os;
}
```

函数覆盖与二义性

派生类成员函数名称与基类相同

```
class Point {  void Print();  };
class Point3D: public Point { void Print();  };
Point pt( 1, 2 );
Point3D pt3d( 1, 2, 3 );
```

调用示例

- pt.Print()：调用Point类的Print成员函数
- pt3d.Print()：调用Point3D类的Print成员函数
- Point类的Print成员函数在Point3D类中仍存在，但被新类中的同名函数覆盖
- 访问规则（解析）：pt3d.Point::Print()

多继承

多继承的基本语法格式

```
class 派生类名称: 派生类型保留字 基类名称1, 派生类型保留字 基类名称2, … { … };
```

多继承示例

```
class A { … };  class B { … };
class C: public A, protected B { … };
```

多继承可能导致的问题：派生类中可能包含多个基类副本，慎用！

```
class A { … };  class B: public A { … };
class C: public A, protected B { … };
```

函数覆盖与二义性

派生类成员函数名称与基类相同

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

调用示例

- c.f()：调用C类的成员函数
- c.A::f()：调用C类继承自A类的函数
- c.B::f()：调用C类继承自B类的函数

函数覆盖与二义性

派生类成员函数名称与基类相同

```
class A {  public: void f();  };
class B: public A {  public: void f();  };
class C: public A {  public: void f();  };
class D: public B, public C {  public: void f();  };
d d;
```

调用示例

- d.f()：调用D类的成员函数
- d.B::f()：调用D类继承自B类的函数
- d.C::f()：调用D类继承自C类的函数
- d.B::A::f()：调用D类通过B类分支继承自A类的函数

虚基类

虚拟继承的目的

- 取消多继承时派生类中公共基类的多个副本，只保留一份
- 格式：派生时使用关键字virtual

使用示例：D中只有A的一份副本

```
class A {  public: void f();  };
class B: virtual public A {  public: void f();  };
class C: virtual public A {  public: void f();  };
class D: public B, public C {  public: void f();  };
```

派生类的构造函数与析构函数

构造函数的执行顺序

- 调用基类的构造函数，调用顺序与基类在派生类中的继承顺序相同
- 调用派生类新增对象成员的构造函数，调用顺序与其在派生类中的定义顺序相同
- 调用派生类的构造函数

析构函数的执行顺序

- 调用派生类的析构函数
- 调用派生类新增对象成员的析构函数，调用顺序与其在派生类中的定义顺序相反
- 调用基类的析构函数，调用顺序与基类在派生类中的继承顺序相反

类的赋值兼容性

公有派生时，任何基类对象可以出现的位置都可以使用派生类对象代替

- 将派生类对象赋值给基类对象，仅赋值基类部分
- 用派生类对象初始化基类对象引用，仅操作基类部分
- 使指向基类的指针指向派生类对象，仅引领基类部分

保护派生与私有派生不可以直接赋值

- 尽量不要使用保护派生与私有派生

类的赋值兼容性

```
#include <iostream>
#include <string>
using namespace std;
class Base
{
public:
  Base(string s) : str_a(s) { }
  Base(const Base & that) { str_a = that.str_a; }
  void Print() const { cout << "In base: " << str_a << endl; }
protected:
    string str_a;
};
class Derived : public Base
{
public:
    Derived(string s1,string s2) : Base(s1), str_b(s2) { }  // 调用基类构造函数初始化
    void Print() const { cout << "In derived: " << str_a + " " + str_b << endl; }
protected:
    string str_b;
};
```

类的赋值兼容性

```
int main()
{
  Derived d1( "Hello", "World" );

  Base b1( d1 );   // 拷贝构造，派生类至基类，仅复制基类部分
  d1.Print();  // Hello World
  b1.Print();  // Hello

  Base & b2 = d1;  // 引用，不调用拷贝构造函数，仅访问基类部分
  d1.Print();
  b2.Print();

  Base * b3 = &d1;  // 指针，不调用拷贝构造函数，仅引领基类部分
  d1.Print();
  b3->Print();
  return 1;
}
```

多态性

多态性

- 目的：不同对象在接收到相同消息时做不同响应
- 现象：对应同样成员函数名称，执行不同函数体

多态性的实现

- 虚函数：使用virtual关键字声明成员函数
- 声明格式：virtual 函数返回值 函数名称(参数列表);

非虚函数示例

```
// 头文件
#include <iostream>
using namespace std;

class Account
{
public:
  Account( double d ) : _balance(d) { }
  double GetBalance() const;
  void PrintBalance() const;
private:
    double _balance;
};

inline double Account::GetBalance() const
{
  return _balance;
}
```

非虚函数示例

```
class CheckingAccount : public Account
{
public:
   CheckingAccount(double d) : Account(d) { }
   void PrintBalance() const;
};

class SavingsAccount : public Account
{
public:
   SavingsAccount(double d) : Account(d) { }
   void PrintBalance() const;
};
```

非虚函数示例

```
// 源文件

void Account::PrintBalance() const
{
   cerr << "Error. Balance not available for base type." << endl;
}

void CheckingAccount::PrintBalance() const
{
  cout << "Checking account balance: " << GetBalance() << endl;
}

void SavingsAccount::PrintBalance() const
{
  cout << "Savings account balance: " << GetBalance() << endl;
}
```

非虚函数示例

```
int main()
{
   CheckingAccount * checking = new CheckingAccount( 100.00 ) ;
   SavingsAccount  * savings  = new SavingsAccount( 1000.00 );

   Account * account = checking;
   account->PrintBalance();

   account = savings;
   account->PrintBalance();

  delete checking;
  delete savings;
  return 0;
}
```

虚函数示例

```
// 头文件
#include <iostream>
using namespace std;

class Account
{
public:
  Account( double d ) : _balance(d) { }
  double GetBalance() const;
  virtual void PrintBalance() const;
private:
    double _balance;
};

inline double Account::GetBalance() const
{
  return _balance;
}
```

虚函数示例

```
class CheckingAccount : public Account
{
public:
   CheckingAccount(double d) : Account(d) { }
   virtual void PrintBalance() const;
};

class SavingsAccount : public Account
{
public:
   SavingsAccount(double d) : Account(d) { }
   virtual void PrintBalance() const;
};
```

虚函数示例

```
int main()
{
   CheckingAccount * checking = new CheckingAccount( 100.00 ) ;
   SavingsAccount  * savings  = new SavingsAccount( 1000.00 );

   Account * account = checking;
   account->PrintBalance();

   account = savings;
   account->PrintBalance();

  delete checking;
  delete savings;
  return 0;
}
```

多态性

纯虚函数

- 充当占位函数，没有任何实现
- 派生类负责实现其具体功能
- 声明格式：virtual void f(int x) = 0;

抽象类

- 带有纯虚函数的类
- 作为类继承层次的上层

虚析构函数

- 保持多态性需要虚析构函数，以保证能够正确释放对象

抽象类与虚函数示例

```
//  “Point.h”

#include <iostream>
using namespace std;

class Point
{
public:
  Point() { }
  virtual void Print() const = 0;
//  virtual ~Point();
};
```

抽象类与虚函数示例

```
class Point2D : virtual public Point
{
public:
  Point2D( int x = 0, int y = 0 ) : _x(x), _y(y) { }
  Point2D( const Point2D& pt2d ) : _x(pt2d._x), _y(pt2d._y) { }
  int GetX() const {  return _x;  }
  void SetX( int x ) {  _x = x;  }
  int GetY() const {  return _y;  }
  void SetY( int y ) {  _y = y;  }
  virtual void Print() const;
protected:
  int _x, _y;
};
```

抽象类与虚函数示例

```
class Point3D: virtual public Point2D
{
public:
  Point3D( int x = 0, int y = 0, int z = 0 ) : Point2D(x,y), _z(z) { }
  Point3D( const Point3D& pt3d ) : Point2D(pt3d._x, pt3d._y), _z(pt3d._z) {  }
  int GetZ() const {  return _z;  }
  void SetZ( int z ) {  _z = z;  }
  virtual void Print() const;
protected:
  int _z;
};
```

抽象类与虚函数示例

```
//  “point.cpp”
void Point2D::Print()
{
  cout << "( " << _x << ", " << _y << " )";
}
void Point3D::Print()
{
  cout << "( " << _x << ", " << _y << ", " << _z << " )";
}

//  “main.cpp”
int main()
{
   Point * pt1 = new Point2D( 1, 2 );
   Point * pt2 = new Point3D( 1, 2, 3 );
   pt1->Print();
   pt2->Print();
}
```

编程实践

9.1 使用面向对象架构实现动态数组库。

9.2 使用面向对象架构实现抽象链表库。