Lecture Notes on C++ Multi-Paradigm Programming

Bachelor of Software Engineering, Spring 2014

Wan Hai

whwanhai@163.com

13512768378

Software School, Sun Yat-sen University, GZ

What is polymorphism

- **Polymorphism refers to the ability to associate multiple meanings to one character or one identifier especially to one function name.**

c = a*b;   int* p;   *p=1;

Polymorphism Example: function overloading

```
int abs(int x)   // 整数类型数据的绝对值函数
{    cout << "Using integer version of abs().\n";
     return (x >= 0 ? x : -x);
}
double abs(double x)          // 浮点类型数据的绝对值函数
{    cout << "Using floating-point version of abs().\n";
     return (x >= 0.0 ? x : -x);
}
long abs(long x)              // 长整数类型数据的绝对值函数
{    cout << "Using long integer version of abs().\n";
     return (x >= 0 ? x : -x);
}
int main()
{    cout << abs(-5) << "\n";          // 调用abs()的整数版本
     cout << abs(-5L) << "\n";         // 调用abs()的长整数版本
     cout << abs(3.14) << "\n";        // 调用abs()的浮点版本
     return 0;
}                                      //程序abs
```

Two types of polymorphism

- **Compile-time polymorphism(编译时多态性)：association done in compile time, including**

    （1）function overloading

    （2）operator overloading

- **Run-time polymorphism(运行时多态性)：association done during run time. Implemented by dynamic biding(inheritance plus virtual function).**

binding（绑定）

- Binding is the process of associating a function call and a function definition.
- Two types of biding
  - Static biding（Early biding）
    - Done during compile-time
    - Applied to non-virtual function.
  - Dynamic biding（late biding）
    - Done during run-time.
    - Applied to virtual function.

virtual function(虚函数)

- **A virtual function is a member function with reserved word ‘virtual ’of a class.**
- “一旦为虚，永远为虚”：**The virtual function of a base will always be virtual in its derived classes.**
- 主要作用：与继承相结合以实现运行时多态性。在公有继承层次中的一个或多个派生类中对虚函数进行重定义，然后通过指向基类的指针（或引用）调用虚函数来实现实现运行时多态性。
- **Polymorphism class: class that contains virtual function(s).**

class Time Specification

```
class  Time                    //  SPECIFICATION FILE  ( time.h )
{

public :

    void   Set ( int  hours , int  minutes , int  seconds ) ;
    void   Increment ( ) ;
    void   Write ( )  const ;
    Time ( int  initHrs, int  initMins,  int  initSecs ) ;
    Time ( ) ;

private :

    int    hrs ;
    int    mins ;
    int    secs ;
} ;
```

`class ExtTime` Specification

```
// SPECIFICATION   FILE                              ( exttime.h)
#include   "time.h"
enum  ZoneType {EST, CST, MST, PST, EDT, CDT, MDT, PDT } ;

class  ExtTime  :  public  Time    // Time is the base class
{
public :
     ExtTime ( int  initHrs ,  int  initMins ,  int  initSecs ,
               ZoneType    initZone ) ;     // constructor
     ExtTime ( ) ;                             // default constructor
     void Set ( int  hours,  int  minutes,  int   seconds ,
               ZoneType   timeZone ) ;
     void Write ( )  const ;

private :
        ZoneType  zone ;        //  added data member
} ;
```

Example: Print()

```
void  Print  ( /* in */  Time   someTime )
{
          cout  <<  “Time is  “ ;
          someTime.Write (  ) ;
          cout  <<  endl ;
}
```

CLIENT CODE

```
Time         startTime ( 8, 30, 0 ) ;
ExtTime    endTime (10, 45, 0, CST) ;

Print ( startTime ) ;
Print ( endTime ) ;
```

OUTPUT

```
Time is  08:30:00
Time is  10:45:00
```

What is the problem?

- **In function Print, static biding is applied to someTime.Write().**
- **That is, during compile time, the compiler has already associated the function call**

  someTime.Write()

  with

  Time::Write()

  according to the type of someTime, which is Time.

Static Binding

- **is the compile-time determination of which function to call for a particular object based on the type of the formal parameter.**

- **when *pass-by-value* is used, static binding occurs.**

Slicing(切割问题)

- 用值传递的方式把派生类的对象传递给父类对象时，仅传递它们相同的数据成员，派生类比父类多出来的数据成员被切割掉。
- 使用引用形参，可避免切割问题，因为传递的是实参的内存地址。
- 代码改为。。。

但输出还是没有达到目的

```
void  Print  ( /* in */  Time&  someTime )
{
          cout  <<  “Time is  “ ;
          someTime.Write (  ) ;
          cout  <<  endl ;
}
```

CLIENT CODE

```
Time        startTime ( 8, 30, 0 ) ;
ExtTime     endTime (10, 45, 0, CST) ;

Print ( startTime ) ;
Print ( endTime ) ;
```

OUTPUT

```
Time is  08:30:00
Time is  10:45:00
```

Dynamic Binding

- **Is the run-time determination of which function to call for a particular object of a descendant class based on the type of the argument.**

- **Declaring a member function to be virtual instructs the compiler to generate code that guarantees dynamic binding.**

Virtual Member Function

```
class  Time                    //  SPECIFICATION FILE  ( time.h )
{

public :

    void   Set ( int  hours , int  minutes , int  seconds ) ;
    void   Increment ( ) ;
    virtual void   Write ( )  const ;
    Time ( int  initHrs, int  initMins,  int  initSecs ) ;
    Time ( ) ;

private :

    int    hrs ;
    int    mins ;
    int    secs ;
} ;
```

达到目的

```
void  Print  ( /* in */  Time&  someTime )
{
     cout  <<  “Time is  “ ;
     someTime.Write (  ) ;
     cout  <<  endl ;
}
```

CLIENT CODE

```
Time          startTime ( 8, 30, 0 ) ;
ExtTime     endTime (10, 45, 0, CST) ;
Print ( startTime ) ;
Print ( endTime ) ;
```

OUTPUT

```
Time is  08:30:00
Time is  10:45:00 CST
```

形参实参数据类型决定？？

```
void Fun( classA& param )
{
    param.MemberFunc();
}


Fun( classB argument);
```

1.若MemberFunc不是虚函数，
形参类型
决定调用哪个函数（静态绑定）
2.若MemberFunc是虚函数，
实参类型
决定调用哪个函数（动态绑定）

例：静态绑定

```
class BASE {
public:
    void who( ) { cout<<"BASE\n";}
};

class FIRST_D:public BASE {
public:
    // 继承成员的重定义
    void who( ) { cout<<"The First Derivation\n";}
};

class SECOND_D:public BASE {
public:
    void who( ) { cout<<"The Second Derivation\n";}
};
```

例：静态绑定

```
void main( )
{
    BASE b_obj;
    FIRST_D f_obj;
    SECOND_D s_obj;
    BASE *p;    // 定义指向基类的指针
    p= &b_obj; p->who();
    p= &f_obj;  p->who();    // 根据赋值兼容性规则
    p= &s_obj; p->who();     // 基类指针可指向派生类对象
}
```

输出结果：
BASE
BASE
BASE

- 不管p指向什么对象，通过p三次调用的都是基类的who函数。
- 原因：调用普通成员函数采用静态绑定方式。通过指针（或引用）调用普通成员函数，仅仅与指针（或引用）的基类型有关，而与该指针当前所指向（或引用当前所关联）的对象无关。

例：动态绑定

将基类**BASE**修改为：

```
class BASE {
public:
     virtual void who( )
    {   cout<<"BASE\n";  }
};
```

输出结果：
BASE
The First Derivation
The Second Derivation

- 则函数调用**p->who()**进行动态绑定：实际调用哪个**who**函数依赖于运行时**p**所指向的对象

动态绑定的另一实现方式：使用引用形参

```
//功能：演示通过基类引用实现动态绑定
//类的定义同上例（略）
void print_identity( BASE& me )
{
    me.who();   //通过基类引用调用虚函数
}

void main( )
{
    BASE b_obj;
    FIRST_D f_obj;
    SECOND_D s_obj;
    print_identity(b_obj);
    print_identity(f_obj);
    print_identity(s_obj);
}
```

输出结果：

BASE

The First Derivation

The Second Derivation

关于虚函数的说明

- 用虚函数实现动态绑定的关键：必须用基类指针（或基类引用）来访问虚函数。
- 若一函数是类中的虚函数，则称该函数具有虚特性。
- 在派生类中重定义从基类中继承过来的虚函数(函数原型保持不变)，该重定义的函数在该派生类中仍是虚函数。
- 函数重载，虚特性丢失。
- 当一个派生类没有重新定义虚函数时，则使用其基类定义的虚函数版本。

虚函数例子

```
class BASE {
public:
        virtual void f1( ) { cout<<"BASE::f1()"<<endl; }
        virtual void f2( ) { cout<<"BASE::f2()"<<endl;  }
        virtual void f3( ) { cout<<"BASE::f3()"<<endl;  }
        void f ( ) { cout<<"BASE::f()"<<endl;  }
};

class DERIVED:public BASE {
public:
    void f1( ) { cout<<"DERIVED::f1()"<<endl;  }
    //虚函数的重定义,f1在该类中还是虚函数
    void f2( int ) { cout<<"DERIVED::f2()"<<endl; }
    //f2是函数重载，虚特性丢失
    void f ( ) { cout<<"DERIVED::f()"<<endl; }  // 普通函数的重定义
};
```

Client codes

```
int main( )
{
    DERIVED d;
    BASE *p = &d ;   // 基类指针p指向派生类对象

    p->f1( );      //调用DERIVED::f1( ); 动态绑定
    p->f2( );      //调用BASE::f2( ); 静态绑定
    p->f ( );      //调用BASE::f( ); 静态绑定
    ((DERIVED *)p)->f2(100); //调用DERIVED::f2( ); 静态绑定

    return 0;
}
```

abstract class (抽象类)

- **Pure virtual function(纯虚函数)：a virtual function declared but without definition within a base class. Each derived class of this base must redefine and implement this function.**
  - **format：**

    virtual 返回值类型 函数名(形参表) **=0 ;**

  - **example：**

    图形基类**FIGURE** 的 **get_area()** 函数为纯虚函数

- **Abstract class：a class that contains pure virtual function.**

Abstract class

- **Syntax of abstract class**
  - **Can only be used as a base class.**
  - **Can not declare objects of an abstract class.**
  - **Can not be used for parameter type or returned type of a function.**
  - **Can not be used in explicit conversion.**
  - **Can declare pointers or references of an abstract class.**

Figure.h

```
#ifndef FIGURE_H
#define FIGURE_H
const double PI = 3.14159;      // 圆周率常量

class FIGURE {
public:
        void set_size(double x, double y = 0);
        virtual double get_area() = 0;   // get_area()被声明为纯虚函数
protected:
        double x_size, y_size;
};

#endif
```

Figure.cpp

```
#include "Figure.h"

void FIGURE::set_size(double x, double y)
{
        x_size = x;
        y_size = y;
}
```

Trangle.h & .cpp

```
#ifndef TRIANGLE_H
#define TRIANGLE_H

#include "Figure.h"

class TRIANGLE: public FIGURE {
public:
        virtual double get_area();
};

#endif
```

```
#include "Triangle.h"

double TRIANGLE::get_area()
{
    return (x_size * y_size / 2);    // 三角形面积 = 底×高÷2
}
```

Retangle.h & .cpp

```
#ifndef RETANGLE_H
#define RETANGLE_H

#include "Figure.h"

class RECTANGLE: public FIGURE {
public:
		virtual double get_area();
};

#endif
```

```
#include "Retangle.h"

double RECTANGLE::get_area()
{
	return (x_size * y_size);	// 矩形面积 = 长×宽
}
```

Circle.h & .cpp

```
#ifndef CIRCLE_H
#define CIRCLE_H

#include "Figure.h"

class CIRCLE: public FIGURE {
public:
        virtual double get_area();
};

#endif
```

```
#include "Circle.h"

double CIRCLE::get_area()
{
    return (PI * x_size * x_size);    // 圆面积 = p×半径×半径
}
```

客户代码 例1

```
Area of triangle is 60
Area of rectangle is 120
Area of circle is 706.858
```

```
int main()
{
    TRIANGLE triangle;
    RECTANGLE rectangle;
    CIRCLE circle;

     // 处理三角形
     triangle.set_size(15, 8);         // 设置三角形的底和高
     cout << "Area of triangle is " << triangle.get_area() << "\n";

     // 处理矩形
     rectangle.set_size(15, 8);        // 设置矩形的长和宽
     cout << "Area of rectangle is " << rectangle.get_area() << "\n";

     // 处理圆
     circle.set_size(15);              // 设置圆的半径
     cout << "Area of circle is " << circle.get_area() << "\n";

    return 0;
}
```

客户代码 例2

```
int main()
{
    FIGURE* figure;

    TRIANGLE triangle;
    RECTANGLE rectangle;
    CIRCLE circle;

                    :
}
```

客户代码 例2

```
Area of triangle is 60
Area of rectangle is 120
Area of circle is 706.858
```

```
int main()
{                                   :
    // 处理三角形
     figure = &triangle;
     figure->set_size(15, 8);       // 设置三角形的底和高
     cout << "Area of triangle is " << figure->get_area() << "\n";

    // 处理矩形
     figure = &rectangle;
     figure->set_size(15, 8);           // 设置矩形的长和宽
     cout << "Area of rectangle is " << figure->get_area() << "\n";

    // 处理圆
     figure = &circle;
     figure->set_size(15);          // 设置圆的半径
     cout << "Area of circle is " << figure->get_area() << "\n";
                                    :
}
```

Client codes

```
FIGURE fun1(int);    X

void fun2(FIGURE a);    X

FIGURE & fun3(FIGURE &a);    ✓


int main()
{
      FIGURE x;    X

      FIGURE *p;    ✓

      return 0;
}
```