第6章：类的多态性与虚函数

计算机系 江志英 jiangzy@mail.buct.edu.cn

本 章 内 容

1. 多态性概述
2. 静态联编和动态联编
3. 虚 函 数
4. 纯虚函数和抽象类
5. 模 板 与 STL类库

面向对象程序设计的特征

抽象性（Abstraction）

封装性（Encapsulation）

继承性（Inheritance）

多态性（Polymorphism）

计算机系 江志英 jiangzy@mail.buct.edu.cn

面向对象程序设计的特征

A 抽象性

B 封装性

C 继承性

D 多态性

多态性概述

polymorphism，“many forms”：即多种形态

在自然语言中，多态性即是“一词多义”；

更准确地说，*多态性是指相同的动词作用到不同类型的对象上*，

例如：

驾驶摩托车

驾驶汽车

驾驶飞机

驾驶轮船

驾驶宇宙飞船

什么是多态性？（OOP）

当不同对象接受到相同的消息产生不同的动作，这种性质称为多态性。

通俗地说，多态性是指用一个名字定义不同的函数，这些函数执行不同但又类似的操作，

即用同样的接口访问功能不同的函数，从而实现“一个接口，多种方法”。

多态性的例子

在C语言中，由于不支持多态，求绝对值的动作要求三个不同的函数名字：

abs()，labs()，fabs()

分别用来求整数，长整数、浮点数的绝对值。

在C++语言中，由于支持多态，求绝对值的动作可以只用一个函数名：

abs()

函数重载 和 **运算符重载** 实现了类的一种多态性

面向对象程序设计的精华

通过一个简单的接口对不同的实现进行概念上的简化。

```
class walkman{
    virtual void on()=0;
    virtual void off()=0;
    virtual void play()=0;
    virtual void record()=0;
    virtual void stop()=0;
};
```

提供给用户相同的接口

Cassette player, CD player, MD player, MP3 player;

应用多态性的好处

多态应用于OOP的目的是允许用一个名字来指定动作的一般类（即逻辑上相似的动作）。

从而带来以下的好处：

提高了处理问题的抽象级别；

降低了程序设计时的复杂性；

（程序员只需记住一个接口，而不是好几个）

计算机系 江志英 jiangzy@mail.buct.edu.cn

C++实现的多态性

编译时多态性：

函数重载

运算符重载

模板

运行时多态性：

- 借助虚函数来获得

在C++中，多态性的实现与联编（Binding）这一概念有关。

以下那种方式不是编译时多态的方式

A 运算符重载

B 函数重载

C 模板

D 虚函数

联 编（Binding）

什么叫联编（Binding）？

一个源程序需要经过编译、连接，才能成为可执行代码。上述过程中需要将一个函数调用链接上相应的函数代码，这一过程称为**联编**。

联编的目的是要建立函数调用与函数体之间的联系，即：

将一个函数调用连接到一函数的**入口地址**。

C++支持两种联编

静态联编：在程序被编译时进行联编；（早联编）

程序执行快，但灵活性较小。

动态联编：编译时无法确定要调用的函数，在程序运行时联编。（晚联编，滞后联编）

灵活性高，程序执行慢。

动态联编实际上是进行动态识别，是C++实现**运行时多态性**的关键因素。

计算机系 江志英 jiangzy@mail.buct.edu.cn

静态联编和动态联编的描述，正确的是

A 静态联编是在运行时进行联编

B 静态联编更加灵活

C 动态联编运行效率更高

D 动态联编是实现运行时多态的关键因素

虚 函 数

virtual float area( ){ return -1;}

virtual 关键字的作用：指示C++编译器对该函数的调用进行动态联编。

计算机系 江志英 jiangzy@mail.buct.edu.cn


```
#include <iostream.h>
class shape{
public:
        float  area( ){ return -1;}
};
class circle :public shape{
        float radius;
public:
        void circle(float r){ radius = r; }
        float  area( ){ return 3.14159*radius*radius; }
};
```

```
void main( ) {
        shape obj, *ptr;
        circle c(3.6);
        ptr=&obj;
        cout<<ptr->area( )<<endl;

        ptr=&c;
        cout<<ptr->area( )<<endl;
}
```

基类 shape 的指针

求圆形的面积

输出的结果分析

输出的结果为:

—1

—1

WHY？？？？？

原因在于这里只是用了静态联编。

前后两次ptr->area()调用都链接到下面的函数实现:

float area(){ return -1;}

```
#include <iostream.h>
class shape{
public:
        virtual  float area( ){ return -1;}
};
class circle : public shape{
        float radius;
public:
        void circle(float r){ radius = r; }
        float area( ){ return 3.14159*radius*radius; }
};
```

被关键字*virtual*说明的函数称为虚函数

计算机系 江志英 jiangzy@mail.buct.edu.cn

```
void main( ) {
        shape obj, *ptr;
        circle c(3.6);
        ptr=&obj;
        cout<<ptr->area( )<<endl;
        ptr=&c;
        cout<<ptr->area( )<<endl;
}
```

基类 shape 的指针

C++的编译器对虚函数调用采取动态联编方式

输出结果:

```
-1
40.715
```

使用虚函数后的输出结果

虚函数的定义

冠以关键字 virtual 的成员函数称为虚拟函数，简称虚函数。

说明虚函数的方法如下：

virtual <类型说明符> <函数名>(<参数表>)

虚函数是成员函数，而且是非static的成员函数。

包含虚函数的类被称为多态类。

虚函数的使用分析

```
class shape{
public:
        virtual float area( ) { };
};
class triangle : public shape{
        float H,W;
public:
        triangle(float h,float w){H=h;W=w;}
        float area( ) {return H*W*0.5;}
};
```

在派生类中，虚函数被重新定义以实现不同的操作。 这种方式称为函数超越（overriding），又称为函数覆盖。

```
class rectangle : public shape{
        float H,W;
public:
        rectangle(float h,float w){ H=h; W=w; }
        float area( ){return H*W;}
};
```

```
void main( ) {
        shape *s;
        triangle  tri(3,4);
        rectangle rect(3,4);
       s=&tri;
       cout<< "The area of triangle: " <<s->area()<<endl;
       s=&rect;
       cout<<  "The area of rectangle: "  <<s->area()<<endl;
}
```

请注意：在这里以指针s所指的对象来确定执行虚函数area（）的哪个版本。

“确定”是在运行时执行的，这构成了运行时的多态性

虚函数是动态联编的基础

virtual 关键字的作用：指示C++编译器对该函数的调用进行动态联编。

尽管可以用对象名和点算符的方式调用虚函数，即向对象发送消息：

tri.area()　或者　rect.area()

这时是静态联编方式。

只有当访问虚函数是通过基类指针s时才可获得运行时的多态性。

关于虚函数描述不正确的是

A 虚函数是用Virtual关键字修饰的

B 对虚函数的调用都是动态联编

C Virtual关键字提示编译器该函数进行动态联编

D 只有当访问虚函数是通过基类指针s时才可获得运行时的多态性

虚函数与重载函数的比较

1. 从形式上说，重载函数要求函数有相同的函数名称，并有不同的参数序列；

而虚函数要求这三项(函数名称、返回值和参数序列）完全相同，即具有完全相同的函数原型。

2. 重载函数可以是成员函数和非成员函数；

而虚函数只能是成员函数。

虚函数与重载函数的比较（续）

3.对重载函数的调用是以所传递参数序列的差别（参数个数或类型的不同）作为调用不同函数的依据；

虚拟函数是根据对象的不同去调用不同类的虚拟函数。

- 虚函数在运行时表现出多态功能，正是C++的精髓之一；而重载函数不具备这一功能。

关于虚函数与重载函数，描述正确的是

A 虚函数只能是成员函数

B 重载函数要求变量的个数必须不相等

C 重载函数与虚函数都可以进行动态联编

D 重载函数可以通过返回值类型进行区别

虚函数小结

如果某类中的一个成员函数被说明为虚函数，这就意味着该成员函数在派生类中可能有不同的实现。

为了实现运行时的多态性，调用虚函数应该通过第一次定义该虚函数的基类对象指针

只有通过指针或引用标识对象来操作虚函数时，才对虚函数采取动态联编方式。

如果采用一般类型的标识对象来操作虚函数，则将采用静态联编方式调用虚函数。

计算机系 江志英 jiangzy@mail.buct.edu.cn

虚函数小结（续）

构造函数不能是虚拟的，但析构函数可以是虚拟的。

设计C++程序时，一般在基类中定义处理某一问题的接口和数据元素，而在派生类中定义具体的处理方法。

通常都将基类中处理问题的接口设计成虚函数，然后利用基类对象指针调用虚函数，从而达到单一接口，多种功能的目的。

纯虚函数和抽象类

纯虚函数是指在基类中声明但是没有定义的虚函数，而且设置函数值等于零。

virtual type func_name(parameter list) =0;

通过将虚函数声明为纯虚函数可以强制在派生类中重新定义虚函数。

如果没有在派生类中重新定义，编译器将会报告错误。

```
class shape{
public:
        virtual float area( )=0;
        //A PURE VIRTUAL FUNCTION
};
class triangle : public shape{
        float H,W;
public:
        triangle(float h,float w){H=h;W=w;}
        float area( ){return H*W*0.5;}
};
```

纯虚函数

抽象类

1. 抽象类：包含有纯虚函数的类称为抽象类。

2. 不能说明抽象类的对象，但能说明指向抽象类的指针，一个抽象类只能作为基类来派生其他的类。

3. 抽象类的指针用于指向该抽象类的派生类的对象。

抽象类的例子

```
void main( ) {
    shape *s;
    triangle  tri(3,4);
    rectangle rect(3,4);
    s=&tri;
    cout<< “The area of triangle: ” <<s->area()<<endl;
    s=&rect;
    cout<< “The area of rectangle: ” <<s->area()<<endl;
}
```

Shape是抽象类，只能定义指向该类对象的指针

计算机系 江志英 jiangzy@mail.buct.edu.cn

虚析构函数

- 何时需要虚析构函数?
  - 当可能通过基类指针删除派生类对象时

例 分析该程序的输出结果，并回答提出的问题。

```
#include <iostream.h>
class A {
public:
      virtual ~A()
      {  cout<<"A::~A() called.\n";  }
};
class B : public A {
    public:
      B(int i)
      {  buffer=new char[i];  }
      ~B( ) {
        delete [] buffer;
        cout<<"B::~B() called.\n";
      }
private:
      char *buffer;
};
```

```
void fun(A *a) {
    delete a;
 }
 void main( ) {
    B *b=new B(5);
    fun(b);
 }
```


计算机系 江志英 jiangzy@mail.buct.edu.cn

关于虚函数与抽象类的说法正确的是：

A 只能通过基类指针调用虚函数

B 可以通过抽象类对象访问虚函数

C 抽象类至少包含一个虚函数

D 纯虚函数的返回值是0

模板

- 目标
  - 代码重用
    - 函数模板
    - 类模板
  - 标准模板库（STL）

模板(template)的概念

C++中的模板提供了重用源代码的方法，
C++的库是基于模板的技术

两种类型的模板

- 类模板
- 函数模板

如果没有函数模板…

通过函数重载实现不用类型的操作

```
int max(int a, int b)
{
  return a > b ? a : b;
}
float max(float a, float b)
{
  return a > b ? a : b;
}
……
```

函数模板——意义

函数模板定义了参数化的非成员函数，使程序能用不同类型的参数调用相同的函数。

由编译器决定该使用哪一种类型，并且从模板中生成相应的代码。

适合对各种数据类型，执行完全相同的操作

根据函数调用中提供的参数，编译器自动实例化不同的对象代码函数

计算机系 江志英 jiangzy@mail.buct.edu.cn

函数模板的定义

关键字: template

声明函数模板的方法如下:

```
template <class T>
T <函数名称>(T <参数名>, … )
{
   <函数操作>
}
```

函数模板——举例

```
#include <iostream.h>

template <class T>
T max( T a, T b) {
    return a > b ? a : b;
}

void main( ) {
    cout << "max(20, 30) = " << max(20, 30) << endl;
    cout << "max('t', 'v') = " << max('t', 'v') << endl;
    cout << "max(10.1, 15.2) = " << max(10.1, 15.2) << endl;
}
```

返回两个元素的最大值

计算机系 江志英 jiangzy@mail.buct.edu.cn

函数模板——原理

编译器自动生成

```
int max(int a, int b)
{
     return a > b ? a : b;
}
float max(float a, float b)
{
    return a > b ? a : b;
}
char max(char a, char b)
{
    return a > b ? a : b;
}
```

函数模板的重载

函数模板和普通函数一样，之间也可以重载，而且函数模板也可以与普通函数之间构成重载关系

```
template<class T>
T Func(T t) {
    return t;
}
template<class T>
int Func(int i,T t) {
    return i*t;
}
int Func(int  i) { //如果有具体的函数则调用具体的函数
return 2*i;
}

void main( ) {
    cout<<Func(3);
}
```

类模板——意义

类模板描述能够管理其他数据类型的通用的数据类型

常用于建立通用的包容器类，如堆栈，链表和队列，对于所有的包容器来说，其维护方式是相同的

类模板

```
template <class T1,class T2...class Tn>
class 类模板名 {
  //类模板定义
}
```

template是C++关键字，表示是对模板进行定义。template后的<>里是模板的参数，参数可以有一个或多个，每个参数用class关键字修饰，并用逗号隔开。

类模板——使用

为类模板中的类型参数指定具体类型的过程叫做类模板的实例化，类模板实例化的结果是类，而不是对象。

```
typedef TStack<char> CStackChar;

typedef TStack<int>  CStackInt;
```

- 类模板的成员函数也可以在类模板的外面定义。如：

```
template<class T>
void TStack<T>::Push(T t) {
    *pLast++=a;
}
```

- 类模板成员函数也可以重载，如：

```
template<class T>
 T TStack<T>::Pop(int i) {
    return *(pLast-i);
}
```

类模板——举例

```
template<class T>
class Array {
public:
    int getlength() {return length;}
    T  operator[](int i) {return array[i];}  //运算符[]重载
   void setarray(T t,int i) { this->array[i]=t; }
    Array(int l) {    //构造函数
        length = l;
        array = new T[length];
    }
    ~Array () {delete [] array;}
private:
    int length;
    T *array;
};
```


```
void main( ) {
   Array<int> a(5);  //int代表T，类模板的实例化产生类，类
     的实例化产生对象
   //Array  a(5);  会出错
   cout<<a.getlength()<<endl;

  a.setarray(3,2);
   cout<<a.operator[](2)<<endl;
   cout<<a[2]<<endl;
}
```

模板小结

所谓模板的思想就是想做一个通用的东西出来。比如一个通用的函数，通用的类。

注意：应该是具有近似功能的函数或者类才能进一步抽象为类模板或者函数。

函数模板就是将近似功能的多个函数抽象为一个，从而提高代码的利用率。注意函数模板实例化出来的函数之间是一个重载的关系。

类模板就是将具有近似功能的多个类进一步抽象为一个类，从而提高代码的利用率。注意此时的类模板实例化后就是普通的类，对普通的类进行实例化后就是对象。**注意：类模板并不是抽象类。**

STL简介
Standard Template Library

标准模板库(STL)是一个C++编程库

- 是一个容器类模板库
- 算法库
- 利用STL能够容易的实现许多标准类型的容器

它的组件是高度参数化的

使C++程序员能够进行通用的程序设计

STL 组成

包括

- 容器类
- 算法
- 迭代器

标准模板库(STL)中

- 通用算法被实现为函数模板
- 容器被实现为类模板
- 迭代器是指向对象的指针

容器类

容纳其他对象的类

包括

- vector、list、deque、set、mutilset、map、multimap、hash set、hash multiset、hash map和hash multimap

其中每个类都是一个模板，能够被实例化容纳任意对象类型

算法

用于操纵容器中所保存的数据

有的算法与容器类是分离的

迭代器

将算法与容器分离的一种机制

允许程序顺序地遍历一个容器中的元素

有些迭代器（如istream和ostream迭代器）与容器无关

指针迭代器

```
#include <iostream.h>
#include <algorithm>
#define SIZE 100
int iarray[SIZE];
int main( ) {
        iarray[20] = 50;
        int* ip = find(iarray, iarray + SIZE, 50);
        if (ip == iarray + SIZE)
          cout << "50 not found in array" << endl;
        else
          cout << *ip << " found in array" << endl;
        return 0;
}
```

容器迭代器

```
#include <iostream.h>
#include <algorithm>
#include <vector>
vector<int> intVector(100);
void main(){
  intVector[20] = 50;
  vector<int>::iterator intIter =
     find(intVector.begin(), intVector.end(), 50);
  if (intIter != intVector.end())
     cout << "Vector contains value " << *intIter << endl;
  else
     cout << "Vector does not contain 50" << endl;
}
```

本章总结

主要内容

- 多态性的概念、两种联编、虚函数、纯虚函数、抽象类、虚析构函数、模板、STL类库

达到的目标

- 理解多态的概念，学会运用多态机制。