多态

Andrew Huang<bluedrum@163.com>

课程内容

- 多态的概念
- 虚函数
- 多态应用实例

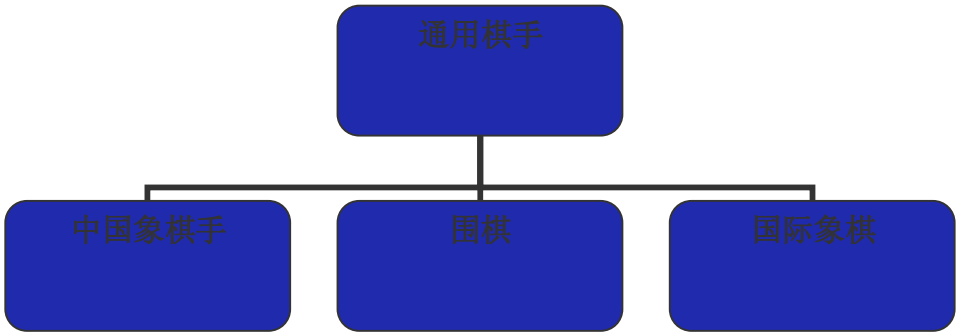
多态概念

多态的概念

- 多态（Polymorphism ）是面向对象（Object-Oriented ，OO ）思想¡°三大特征¡±之一，其余两个分别是封装（Encapsulation ）和继承（Inheritance ）
- 在程序设计领域，一个广泛认可的定义是¡°一种将不同的特殊行为和单个泛化记号相关联的能力¡±
- 通俗地讲，多态是用同一接口实现不同功能的机制
- 可以看成是继承相反的机制
  - 继承是派生类扩展基类的功能的从上而下的机制
  - 多态是基类调用子类的虚函数的从下而上的机制

多态的实例

- 棋手的基本活动是下棋,但是对不同类型的棋手,下棋的内容不一样.
- 按OOP的观念,棋手是基类,中国象棋手,围棋手和国际象棋手,是派生类.
- 下棋是所有棋手类的方法.由具体的派生类实现下棋的功能.
- 当抽象的通用棋手来进行操作时,它会调用具体棋手的实现方法来完成下棋功能.这就是多态概念



多态的实现

- 多态是一种机制、一种能力，而非某个关键字。它在类的继承中得以实现，在类的方法调用中得以体现 .
- C++中的多态有着更广泛的含义。分为两大类,静态多态和动态多态
  - 常见的通过**类继承和虚函数机制**生效于运行期的动态多态（dynamic polymorphism）
  - 模板也允许将不同的特殊行为和单个泛化记号相关联，由于这种关联处理于编译期而非运行期，因此被称为静态多态（static polymorphism）。
- 多态性给我们带来了好处：多态使得我们可以通过基类的引用或指针来指明一个对象

（包含其派生类的对象），当调用函数时可以自动判断调用的是哪个对象的函数。

静态联编（通过对象的类型区别）

- 静态联编就是一般的函数重写(override)，重新在派生类中定义基类的函数

```
class point{…
float area(){return 0.0;}
};
class circle:public point{…
float area( ){return 3.14159*r*r;}
};
void main( )
{   point p;
    p.area( );      调用 point 类的函数
    circle c;
    c.area( );      调用 circle 类的函数
}
```

虚函数

多态的基础¡ª¡ª 虚函数（virtual ）

- 虚函数：为实现某种功能而假设的函数，虚函数只能是类中的一个成员函数，不能是静态成员
- 格式：virtual 数据类型 函数名（参数）
  - 首先在基类中声明虚函数（需要具有动态多态性的基类）
  - 在派生类中某个成员函数的**参数个数**、**相应类型**和**返回类型**与基类**同名**的虚函数一样（即使没有virtual)，则该成员函数为虚函数
  - 不允许在派生类中定义与基类仅仅返回类型不同的函数
  - 派生类中定义与基类虚函数同名但参数不同的函数，则该函数不为虚函数。
  - 对虚函数保证在**通过一个基类类型的指针调用**一个虚函数时，系统对该调用进行动态联编。
  - 对虚函数在**通过一对象调用**一个虚函数时，系统对该调用进行静态联编。
  - 在调用中对虚函数使用成员名限定可强制对该函数的调用使用静态联编。

运行时的多态性

- 指定关键字virtual ，在运行时对函数动态联编，根据实际对象，调用该对象的成员函数

```
class circle : public point
{};
virtual float circle::area( ){return 0.0;}

void main( )
{   point *p;
    circle c;
    p=&c;
    p->area( );     结果调用 circle 类的 area
```

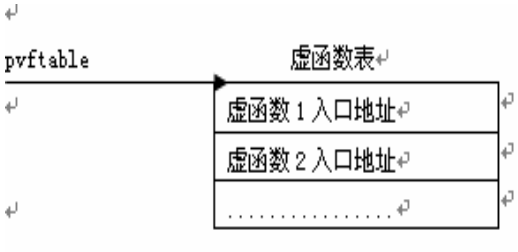
完整的实例

```
#include <iostream.h>
class Shapes
{
public:
        virtual void draw()             //基类中的函数
        {
                cout << "绘制基本形状\n";
        }
};
class Circle : public Shapes
{
private:
        int radius;
public:
        virtual void draw()             //在派生类中重新定义
        {
                cout << "绘制圆形\n";
        }
};
```

```
class Square : public Shapes
{
private:
        int length;
public:
        void draw()                     //在派生类中重新定义
        {
                cout << "绘制正方形\n";
        }
};
void main()
{
        Circle c;
        Square s;
        Shapes* ptr;
        ptr = &c;
        ptr->draw();
        ptr = &s;
        ptr->draw();
}
```

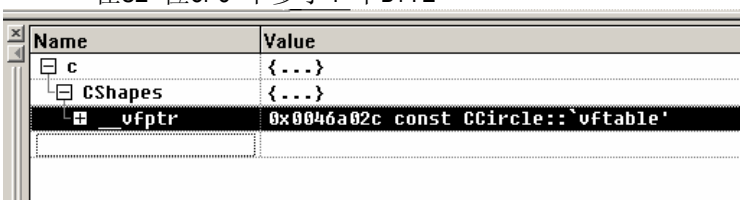
虚函数表

- 如果类中包含有虚成员函数，在用该类实例化对象时，对象的第一个成员将是一个指向虚函数表的指针(pvftable) 。虚函数表记录运行过程中实际应该调用的虚函数的入口地址
- 用VC 的Watch 可以查看pvftable 地址



类的sizeof

- 普通类的sizeof是所有数据成员经过字节对齐的总和
  - 不包括静态数据成员
- 带虚函数类的sizeof 在数据成员总和之上加上一个虚函数入口地址
  - 在32 位CPU 下多了4 个BYTE



纯虚函数

- 基类无法（或没有必要）提供虚函数的实现
- 将虚函数声明为纯虚函数

```
virtual void print() = 0;
```

- 派生类要创建对象，必须实现纯虚函数
- 不能创建含有纯虚函数的类的对象

抽象类

- 抽象类：至少包含有一个纯虚函数的类，只能作为基类来派生新类.本身不能创建实例
- 抽象类不能实例化对象，但是可以用抽象类的指针指向派生类对象，并调用派生类的虚函数的实际实现。
  - 在抽象类中也可定义普通成员函数或虚函数，虽然不能为抽象类声明对象，但可通过派生类对象来调用不是纯虚函数的函数。
  - 在成员函数内可以调用纯虚函数，但在构造函数和析构函数内不能调用

```
class CWorm
{
public:
        virtual void Draw()=0;
};
class CAnt:public CWorm
{
public:
        void Draw()
        {
                cout<<"CAnt::Draw()"<<endl;
        }
};
bool CreateInstance(void **pInterface)
{
        *pInterface=new CAnt;
        return true;
}
```

```
void main()
{
        CWorm *pWorm;
        pWorm=new CAnt;
        //或 CreateInstance((void **)&pWorm);
        pWorm->Draw();
        delete pWorm;
}
```

虚析构

- 调用析构函数是为了释放由构造函数分配的内存空间
- 如果基类的析构函数是非虚的，则不能用指向派生类的指针调用派生类的析构函数
- 需要虚析构函数

虚析构函数

- 只要基类的析构函数被说明为虚函数，则派生类的析构函数自动成为虚函数。而构造函数的调用意味着建立一个对象，这时必须确切的知道这个对象的类型，因此无意义
- 虚析构函数的说明：
  - virtual ~类名（）{…}

```
class A{
                public:
                  A( ){ }
                  virtual ~A( ){cout<< “Destructor A” <<endl;}      };
          class B:public A{
                public:
                B( ){ }
                ~B( ){cout<< “Destructor B” <<endl;} };
        void main()
        {     A *pa=new B;
                delete pa;
  }
```

虚析构/ 非虚析构的实例

```
#include <iostream.h>
class Alpha
{
private:
      char* alpha_ptr;
public:
      Alpha()                  //构造函数不能是虚函数
      {
            alpha_ptr = new char[5];
      }
      virtual ~Alpha()     //虚析构函数
      {
            delete[] alpha_ptr;
            cout << "Alpha 的析构函数" << endl;
      }
};
```

```
class Beta : public Alpha
{
private:
	char* ptrderived;
public:
	Beta()
	{
		ptrderived = new char[100];
	}
	~Beta()
	{
		delete[] ptrderived;
		cout << "Beta 的析构函数" << endl;
	}
};
void main()
{
	Alpha *ptr = new Beta;
	delete ptr;
}
```

多态设计实例

多态设计实例

- 异构结点链表
- 支持不同格式棋谱系统
  - 类工厂模式 ClassFactory

课堂练习

- 把单链表代码CLinkList 类增加查找功能，
  - Search(char * context);
  - 通过遍历所有结点CNode 的match 虚函数来实现
  - 每个CNode 结点的派生类必须实现match 方法，扩展CIntNode,CCharNode 的match 方法
- 新增一些其它CNode 派生类，并加入到测试中