第十四讲　多 态

潘建瑜@MATH.ECNU

继承/派生

面向对象的特点

抽象、封装、继承、多态

继承与派生

- 提高了代码的可重用性，有利于软件开发和维护
- 派生过程：吸收父类成员、改造父类成员、添加新成员
- 构造函数，访问权限，屏蔽规则，类型兼容规则，虚继承

多态

- 允许使用相同的接口来处理不同类型的对象，使程序更加灵活和可扩展
- 函数重载、运算符重载、虚函数、模板

多态 (polymorphism)

多 态

多态是指同样的消息被不同类型的对象接收时会导致不同的行为，即接口的多种不同的实现方式。比如调用具有相同函数名的函数，但实现不同的功能。

- 消息：对类的成员函数的调用
- 不同行为：不同的实现（功能），即调用不同函数

多态的实现：

—— 函数重载/运算符重载，**虚函数/纯虚函数，模板**

多态是面向对象程序设计的关键技术之一，若不支持多态，则不能称之为面向对象的语言。

目录页

Contents

1 虚函数 (Virtual)

为什么虚函数：

用派生类对象替代父类的对象后，只能发挥父类的功能。比如父类有成员函数 show，派生类也有 show，则用派生类对象替代父类对象后，只能发挥父类的 show 功能。如果仍然要发挥派生类的 show 功能呢?

→ 将其声明为虚函数

虚函数的声明

```
virtual 数据类型名 函数名(参数列表);
```

```
class Person   // 父类
{
    ... ...
    virtual void show()  // 虚函数
    {
       cout << "Name: " << name << endl;
       cout << "Age: " << age << endl;
    }
    ... ...
};
```

```
class Student : public Person   // 派生类
{
    ... ...
    virtual void show() // 虚函数
    {
       Person::show();
       cout << "Stuid: " << stuid << endl;
    }
    ... ...
};
```

举例：虚函数

ex14_virtual_fun01.cpp

```
int main()
{
    Person p1("Gao Dai", 18);
    Student stu1("Xi Jiajia", 18, 20150108);
    Person * p; // 父类指针

    p = &p1;
    p->show();  // 父类指针 p 指向父类对象，调用的是父类的 show()
    cout << endl;

    p = &stu1;
    p->show();  // 父类指针 p 指向派生类对象，调用的是派生类的 show()
}
```

- 虚函数提供了一种 通过父类访问派生类功能 的机制
- 虚函数只能借助于指针或引用才能达到多态的效果，否则无法实现多态

虚函数

推迟联编/动态联编：

一个函数的调用并不是在编译时刻被确定的，而是在程序运行时才被确定。由于编写代码的时候并不能确定被调用的是父类的成员函数还是派生类的成员函数，所以称为“虚”函数。

ex14_virtual_fun02.cpp

```
void fun(Person * p)
{
    p->show();    // Person:show() or Student:show()?
}                 // 同一段代码，可以产生不同效果 → 多态
```

虚函数

派生类中虚函数的关键字 virtual 可以省略，这意味着：只要在父类中声明的虚函数，则所有派生类（含派生类的派生类）中的同名函数（形参也相同）都是虚函数

```
class Person
{
    ... ...
    virtual void show() { ... ...  }
    ... ...
};

class Student : public Person
{
    ... ...
    void show() { ... ...  }  // 可以不带 virtual
    ... ...
};
```

ex14_virtual_fun03.cpp

派生类的派生类

```
class A // 父类
{ public:
    virtual void show()
    { cout << "x = " << x << endl; }
    ... ...
};
class B : public A   // 派生类
{ public:
    void show()
    { cout << "y = " << y << endl; }
    ... ...
};
class C : public B   // 派生类的派生类
{ public:
    void show()
    { cout << "z = " << z << endl; }
    ... ...
};
```

ex14_virtual_fun04.cpp

```
int main()
{   A z1(1);
    B z2(1,2);
    C z3(1,2,3);

    A * p1;
    p1 = &z1; p1->show(); // 1
    p1 = &z2; p1->show(); // 2
    p1 = &z3; p1->show(); // 3

    B * p2;
    p2 = &z1; p2->show(); // ERROR
    p2 = &z2; p2->show(); // 2
    p2 = &z3; p2->show(); // 3
}
```

只要父类声明了虚函数，则所有派生类（含派生类的派生类）中的**相同函数**都是虚函数。

几点注记

- 只需在声明处加关键字 virtual，函数定义（在外部定义）时不能加
- 如果派生类中没有对虚函数重新定义，则仍使用父类的虚函数
- 构造函数不能是虚函数，但析构函数可以（有时析构函数必须声明为虚函数）

虚函数构成多态的条件

- 必须存在继承关系
- 继承关系中必须有虚函数，且形成严格的屏蔽关系，即父类与派生类中存在具有相同函数原型（函数名和形参都要相同）的虚函数
- 只能通过父类的指针或引用来调用虚函数

ex14_virtual_fun05.cpp

虚析构函数

虚析构函数主要用于防止内存泄漏，通常是在派生类中含有指针成员时才会用。

ex14_virtual_destructor.cpp

```
class A // 父类
{ public:
    A() { x = new int(0); }
    virtual ~A() { delete x; }
  protected:
    int * x;
};
class B : public A   // 派生类
{ public:
    B() { y = new int(0); }
    ~B() { delete y; }
  private:
    int * y;
};
```

```
int main()
{
    A z1;
    B z2;

    A * p1 = new A();
    delete p1;

    B * p2 = new B();
    delete p2;

    A * p3 = new B();
    delete p3; // 若父类不是虚析构函数，则内存泄漏
}
```

2 纯虚函数与抽象类

- 纯虚函数
- 抽象类

纯虚函数与抽象类

纯虚函数

- 纯虚函数没有函数体，主要用来规范派生类的行为，实际上就是所谓的“接口”，它告诉使用者：我的派生类都会有这个函数，但具体实现由各个派生类定义。
- 如果类中有纯虚函数，则表示：我是一个抽象类，不能实例化！

抽象类：带有纯虚函数的类称为抽象类

- 抽象类是特殊的类，是为了抽象和设计的目的而建立的，一般处于继承层次结构的较上层
- 抽象类不能声明对象，只能派生。若纯虚函数在派生类中没有定义，则派生类还是抽象类
- 抽象类的主要作用是将相关功能作为接口组织在一个继承层次结构中，为派生类提供一个公共的根，具体实现由派生类完成。

纯虚函数的声明

```
virtual 类型说明符 函数名(...)=0;   // =0 表示一个虚函数为纯虚函数
```

有时定义纯虚函数的目的就是让父类不可实例化（声明对象）。事实上，有些父类只是用来派生，如动物，用他来创建对象没有任何实际意义。

举例：纯虚函数

ex14_virtual_fun_pure.cpp

```
class Person  // 父类，包含纯虚函数，是抽象类
{
    ... ...
    virtual void show()=0;  // 纯虚函数
    ... ...
};
class Student : public Person    // 派生类
{
    ... ...
    void show() { ... ...  }  // 实例化
    ... ...
};
class Teacher : public Person     // 派生类
{
    ... ...
    void show() { ... ...  }  // 实例化
    ... ...
};
```

3 模板 (Template)

为什么模板：

- 设计更具通用性的函数和类，使得它们可以作用于不同类型的数据，使得程序更加简洁和高效。
- C++ 模板包含：模板函数和模板类

- 模板是 C++ 最强大的特性之一
- 模板提供了在函数中**将数据类型作为参数**的功能

模板函数的声明和定义

```
template <typename T>  // 模板头
T fun(T x, T y)
{ 函数体; }
```

- typename 是关键字（也可以用 class），T 是形参，代表 类型参数
- 模板头和函数是一个整体，为了代码更加清晰，模板头通常独占一行
- 调用模板函数时，类型实参可以是基本数据类型，也可以是自定义的类
- 类型参数可以一个，也可以多个

```
template <typename T1, typename T2, typename T3>
T3 fun(T1 x, T2 y)
{ 函数体; }
```

模板函数：例一

ex14_template_01.cpp

```
template <typename T1>  // 模板头
T1 myfun(T1 x, T1 y)
{
    . . . . . // 函数体
}

int main()
{
    int a=2, b=3, c;
    float e=2.2, f=2.3, g;
    c = myfun<int>(a,b);        // 通过尖括号传递类型参数
    g = myfun<float>(e,f);

    return 0;
}
```

模板函数：例二

```
template <typename T1, typename T2, typename T3>
T3 myfun(T1 & x, T2 & y)
{
    . . . . . // 函数体
}

int main()
{
    int a=2;
    double e=3.14;
    float f;
    f = myfun<int,double,float>(a,e);

    return 0;
}
```

- 可以有多个类型参数
- 返回数据的类型也可以用模板

ex14_template_02.cpp

模板函数：例三

ex14_template_03.cpp

```
class Rational
{ ... ... };

template <typename T>
T Add(T & x, T & y)
{  return x+y;  }

int main()
{
    Rational a(4,5), b(7,3), c;
    c = Add<Rational>(a,b);
    cout<<"a+b=";
    c.Display();
}
```

- 类型参数也可以是类名

模板函数：例四

ex14_template_04.cpp

```
template <typename T>
T Div(T x, T y)
{  return x/y; }

int main()
{
    int a=3, b=2;
    double e=3.0, f=2.0;
    cout << "自动判断数据类型：a/b=" << Div(a,b) << endl;
    cout << "自动判断数据类型：e/f=" << Div(e,f) << endl;
    cout << "明确指定数据类型：a/b=" << Div<double>(a,b) << endl;

    return 0;
}
```

- 在调用模板函数时，类型实参可以省略，编译器会根据传入的数据自动推断其数据类型（包括基本数据类型和用户自定义的类）

模板类

- C++ 除了支持模板函数，还支持模板类。
- 模板函数的类型参数可用于声明和定义函数，而模板类的类型参数则可以用在类的声明和实现中（数据成员和函数成员）。
- 模板类的目的同样是将数据类型参数化。

模板类的声明

```
template <typename T1, typename T2, ... ...>
class 类名
{ 类的声明 };
```

模板类成员函数的定义

```
template <typename T1, typename T2, ... ...>
类型标识符 类名<T1,T2,...>::函数名(形参列表)
{
    ... ...  // 函数体中可以直接使用类型参数
};
```

```
template <typename T>
class Point
{ public:
    Point(T x, T y); // 构造函数声明
  private:
    T x, y;
}
```

```
template <typename T>
Point<T>::Point(T x, T y) // 构造函数的定义
{
    this->x = x;
    this->y = y;
}
```

- 在类外部定义成员函数时，仍然需要带上模板头，类名后面也要带上类型参数，但不用加 typename

模板类举例

```
template <typename T>
class Point
{
  public:
    Point() { x = 0; y = 0; };
    Point(T x, T y);
    T Dist( Point<T> & A);
    Point<T> Move(T xshift, T yshift);
    void Disp();

  private:
    T x, y;
};
```

ex14_template_class_01.cpp

```
template <typename T>
Point<T>::Point(T x, T y)
{ this->x = x; this->y = y; }

template <typename T>
T Point<T>::Dist( Point<T> & A)
{  T xx = x - A.x, yy = y - A.y;
   return sqrt(xx*xx + yy*yy);
}

template <typename T>
Point<T> Point<T>::Move(T xshift, T yshift)
{  return Point(x+xshift, y+yshift);  }

template <typename T>
void Point<T>::Disp()
{ cout << "(" << x << "," << y << ")"; }
```

几点说明

- 模板类的成员函数通常是模板函数
- 若成员函数是模板函数且在外部定义，则需加模板前缀
- 用模板类创建对象时，必须明确指明数据类型，编译器不能根据给定的数据判断其数据类型
- 模板类的类型参数可以带缺省值，但模板函数不行

由于此时的类是 类名<T>，因此在所有需要声明该类对象的地方都要写 类名<T> (包括声明该类的对象, 函数返回值是该类的对象，函数形参是该类的对象等)。但构造函数和析构函数不需要加类型参数。另外，在类内部也可以省略类型参数。

举例：类型参数带缺省值

```
template <typename T=int> // 类型参数带缺省值
class Point
{
  public:
    Point(T a, T b);
    T Getx() { return x; }
    T Gety() { return y; }

  private:
    T x, y;
};
```

```
template <typename T>
Point<T>::Point(T a, T b)
{ x=a; y=b; }

int main()
{
    Point<> A(4.2,5.3); // 使用缺省值
    cout << "A.x=" << A.Getx() << endl;

    Point<float> B(4.2,5.3);
    cout << "B.x=" << B.Getx() << endl;

    return 0;
}
```

ex14_template_class_02.cpp

举例：多个类型参数

```
template <typename T1, typename T2>
class Point
{
  public:
    Point(T1 a, T1 b) {x=a; y=b; };
    T2 dist(const Point &);

  private:
    T1 x, y;
};
```

ex14_template_class_03.cpp

```
template <typename T1, typename T2>
T2 Point<T1,T2>::dist(const Point & P)
{
    T2 dx = x - P.x;
    T2 dy = y - P.y;
    return sqrt(dx*dx + dy*dy);
}

int main()
{
    Point<int, float> A(2.2,3), B(5.5,7);
    cout << "||A-B||=" << A.dist(B) << endl;

    return 0;
}
```

模板类

- 模板增强了 C++ 语言的灵活性，虽然不是 C++ 的首创，但是却在 C++ 中大放异彩
- C++ 模板有着复杂的语法，我们这里只是做了简单介绍
- C++ 模板非常重要，整个标准库几乎都是使用模板来开发的，STL（Standard Template Library，标准模板库） 更是经典之作

第十四讲上机作业

1、设计一个名为 **Integer** 的类，这个类包括：

- 一个保护型的 `int` 数据成员：`x`
- 一个不带形参的构造函数，设置 `x=0`
- 一个带形参的构造函数：`Integer(int x)`
- 纯虚成员函数 `Display()`

设计一个名为 **Rational** 的派生类，代表有理数，以公有方式继承 **Integer**，这个类包括：

- 一个私有型的 `int` 数据成员：`y`，代表分母，继承的 `x` 代表分子
- 一个不带形参的构造函数，设置 `y=1`
- 一个带形参的构造函数：`Rational(int x, int y)`
- 成员函数 `Display()`，以 `x/y` 形式输出有理数，如 `2/3`

设计一个名为 **Complex** 的派生类，代表整型复数，以公有方式继承 **Integer** ，这个类包括：

- 一个私有型的 `int` 数据成员：`y`，代表虚部，继承的 `x` 代表实部
- 一个不带形参的构造函数，设置 `y=0`
- 一个带形参的构造函数：`Complex(int x, int y)`
- 成员函数 `Display()`，输出复数，如 `4-3i`

实现上面三个类，并在主函数中测试：

创建有理数 `x=9/19` 和复数 `z=3-8i`，并通过 **Integer** 类的指针在屏幕上输出 `x` 和 `z`。（程序取名为 **hw14_01.cpp**）

第十四讲上机作业

2、设计一个名为 **Point2D** 的类，表示平面坐标下的一个点，这个类包括：

- 两个 `double` 型数据成员：`x, y`，分别表示横坐标和纵坐标
- 一个带形参的构造函数：`Point2D(double x, double y)`
- 成员函数 `double dist(const Point2D &)`，返回当前点与给定点的距离

设计一个名为 **Point3D** 的类，表示三维空间的一个点，这个类包括：

- 三个 `double` 型数据成员： `x, y, z`，代表点的坐标
- 一个带形参的构造函数：`Point3D(double x, double y, double z)`
- 成员函数 `double dist(const Point3D &)`，返回当前点与给定点的距离

定义一个模板函数 `double mydist(T1 & x, T1 & y)`，使得他既可以计算两个 **Point2D** 对象之间的距离，也可以计算两个 **Point3D** 对象之间的距离。

实现上面两个类和模板函数，并在主函数中测试：
创建 `A1(1.2,3.4), A2(5.6,7.8)` 和 `B1(1.2,3.4,5.6), B2(9.8,7.6,5.4)`，并调用模板函数 `mydist` 输出 `A1, A2` 之间的距离和 `B1, B2` 之间的距离。

（程序取名为 **hw14_02.cpp**）

第十四讲上机作业

3、用模板类实现 **Complex** 类，这个类包括：

- 两个数据成员：`real` 和 `imag`，分别表示实部和虚部
- 一个不带形参的构造函数，用于创建复数 `0`，即实部和虚部都为 `0`
- 一个带形参的构造函数 `Complex(T real, T imag)`，用于创建指定实部和虚部的复数
- 成员函数 `double Magnitude()`，计算复数的模长
- 以成员函数方式重载加法运算 `Complex operator+ (const Complex &)`
- 成员函数 `void Display()`，在屏幕上输出一个复数，如 `2+3i`, `4-5i`

实现这个类，要求在类外部定义所有成员函数（构造函数除外），并在主函数中测试这个类：

(1) 定义两个单精度型复数 `z1=2.1+5.3i` 和 `z2=1.9-2.3i`，计算 `z1+z2` 及其模长；

(2) 定义两个整型复数 `a1=2+5i` 和 `a2=1-2i`，计算 `a1+a2` 及其模长。

（程序取名为 **hw14_03.cpp**）