# C++\_多态\_02\_通用性多态特定性多态

## 引言

多态从应用的角度可以分为两类：通用性多态和特定性多态。

通用性多态分为：参数多态(parametric)和包含多态(inclusion)；

特定性多态分为：重载多态(overloading)和强制多态(coercion)。

## 参数多态

类型参数化多态是指当一个函数(或类)对若干个类型参数操作时，这些类型具有某些公共的语义特性，可以用该函数(或类)来描述这些公共语义特性。C++中的模板是实现类型参数化多态的工具，分为类模板和函数模板。

参数多态与类模板相关联，它把功能相似仅数据类型(或类类型)不同的函数或类类设计为通用的函数模板或类模板，从而实现了相同的函数或类的成员函数对多种数据类型的数据处理。泛型编程：编写与类型无关的通用代码，是代码复用的一种手段，模板是泛型编程的基础。

### 类模板

一个类模板可以表示一组类。下面是一个通用栈类,它可以存放整数、字符或类对象。

# include<stdlib.h>

template <class T> class stack // stack 类模板

{

T \* v;

T \* p;

int sz ;

public :

stack(int s) {v = p = new T[ sz = s];}

～stack( ) {delete [ ] v;}

void push( T a) { &p++ = a;}

T pop( ) { return 3 22p;}

int size( ) const {return p2v;}

}

void main( )

{

stack <char> sch(20); // stack 类模板用char 实例化后创建对象

stack <int> si(20); // stack 类模板用int 实例化后创建对象

sch.push('a');

si.push( 10);

}

其中，stack类模板中带有一个类型参数T，表示栈中存放对象的类型，它不是实际类型。因此不能用类模板直接生成实例对象。通过对类模板的实例化(即给类模板的参数指定具体类型的过程)，类模板实例化后的结果是类而不是实例对象，因此可用类模板实例化后的结果类产生实例对象。在main( )函数中，从stack类模板实例化了两个模板类:stack<char>和stack<int>，然后又由这两个模板类分别实例化各自对象:sch和si。

### 函数模板

与类模板相似，可以定义操纵一组类型的函数。下面定义了一个求两对象间的最大值的函数模板。

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

该函数模板可以求int , char , float 指针或任何重载了> 运算符的类对象间的最大。函数模板也要实例化，实例化后就生成了具体的函数代码(即特定于参数的类型)，与类模板不同的是，它的实例化不需要用户显式进行，而是在函数调用时由编译器来处理。

例如:

int a , b;

char c , d;

int m1 = max( a , b) ; // 调用max( int a , int b) ;

int m2 = max( c , d) ; // 调用max( char c , char d) ;

事实上，函数模板表示了一组名字相同的函数，这些函数之间以及这些函数与其它同名函数之间是重载函数的关系。在使用函数模板时，应保证函数的参数与模板函数的参数正好相配，因为编译器不会给模板函数的参数提供任何形式的转换。

总之，模板描述了一组类或一组函数，避免了为各种不同的数据类型进行重复的编码工作。

## 包含多态

包含多态就是运行时多态。在C++中公有继承关系是一种包含关系。派生类直接公有继承基类定义中的属性或服务,如果一个程序段既能处理基类的对象也能处理基类的派生类的对象，该程序段称为多态程序段。

C++采用虚函数实现包含多态，同样的操作可用于一个类型及其子类型。一个函数一旦声明为虚函数，在编译阶段，编译器并不按照它的静态类型为它生成调用此函数的版本，而只为它生成虚函数表(表中存放与此函数同名、同参数、同返回值的虚函数的地址)。在程序运行时，再根据实际对象的类型，查虚函数表，找出相应版本的函数后，才能使用它。因此，这种不是在编译阶段而是在运行阶段动态确定应使用哪一个虚函数的方式叫动态束定。

要把一个函数声明为虚函数，只要在原函数的声明之前加上virtul关键字即可。利用虚函数进行动态束定，必须用指向基类的指针或引用来访问它，这是因为C++是类型系统，在编译阶段，C++的变量名或函数名就与相应的存储单元联系起来，使用名字也就是使用对应的存储单元。这有利于检查类型系统，并可产生高效代码。但这种类型的限制缺乏灵活性，不能在运行时改变名字的含义，然而多态要求在不同的上下文中，同一名字有多种含义，C++引入虚函数的目的是告诉编译器在运行时才能确定要调用哪一个虚函数。为了把变量名与相应的存储单元分开，它用指针来调用虚函数。这样，只要改变指针所存地址的值，也就间接地改变了调用对象。

如果派生类中覆盖了基类对象的函数，根据赋值兼容，用基类类型的指针或引用指向派生类，就可以通过这个指针或引用来使用派生类的成员函数。如果这个函数是普通成员函数，通过基类指针访问到的只能是基类的成员成员。如果将其设置为虚函数，则可以使用基类类型的指针访问到指针正在指向的派生类的同名函数，这样通过基类类型的指针就可以使属于不同派生类的不同对象产生不同的行为，从而实现运行时过程的多态。

包含多态的及实现包含多态的构成条件：

1. 基类中必须包含有虚函数
2. 在派生类中必须对基类中的虚函数进行重写。
3. 必须通过基类的指针或者引用来调用虚函数。

#include < iostream. h >

class Point //定义基类Point

{

private:

float x, y;

public:

Point(){};

Point(float i, float j)

{

x = i;

y = j;

}

virtul float area() //声明为虚函数

{

return 0. 0;

}

};

const float Pi = 3. 141593;

class Circle : public Point //定义派生类Circle

{

private:

float radius;

public:

Circle(float r)

{

radius = r;

}

virtul float area()

{

return Pi 3 radius 3 radius;

}

};

void main()

{

Point\* pp; // 指向基类的指针pp

Circle c(5. 4321);

pp = &c;

cout << pp->area() << endl; // 调用虚函数, 执行派生类中定义的函数area()

}

class Person

{

public:

virtual void BuyTicket()

{

cout<<""全价买票""<<endl;

}

};

class Student:public Person

{

public:

virtual void BuyTicket() //虚函数的覆盖

{

cout<<""半价买票""<<endl;

}

};

void Func(Person & people)

{

people.BuyTicket();

}

void Test()

{

Person Mike;

Func(Mike); //调用Person里的BuyTicket

Student Tom;

Func(Tom); //调用Student里的BuyTicket

}

从上面例子可看出,利用虚函数可在基类和派生类中使用相同的函数名定义函数的不同实现,从而实现“一个接口，多种方式”. 一般说来,外部函数不能声明为虚函数,成员函数(除构造函数)都可以声明为虚函数。然而，在处理虚函数时,要动态确定应该选用哪一版本函数，故它比标准函数需要更多的存储空间。因此，虚函数仅用于处理派生类中一系列同名、同参数和同返回值的函数。

## 重载多态

重载多态是多态性中最简单的形式，同时也属于编译时多态，它分为函数重载和运算符重载。

1. 函数重载 函数名相同，参数类型或参数个数不同构成的多态，编译器根据参数列表的不同区别不同函数。
2. 2.运算符重载 运算符重载是对已存在运算符赋予多重含义，同一个运算符作用于不同的类型数据导致不同类型的行为，它的实质是函数重载。

### 函数重载

重载函数是指同一作用域内名字相同、但参数不同的函数.

例如:

#include <iostream.h>

int func( int x, int y)

{

return x < y ? y :x;

}

float func( float x, float y)

{

return x < y ? y :x;

}

double func( double x, double y){

return x < y ? y :x;

}

void main()

{

int n1 = 8 , n2 = 10 ;

cout < <“the max is :”< < func( n1 ,n2) < < endl ;

float m1 = 4. 3 , float m2 = 2. 6 ;

cout < <“the max is :”< < func( m1 ,m2) < < endl ;

double f1 = 2. 0 ,f2 = 4. 9 ;

cout < <“the max is :”< < func( f1 ,f2) < < endl ;

}

运行结果为:

the max is :10

the max is :4. 3

the max is :4. 9

上述3 个函数,函数名相同,函数参数不同,因为编译器是根据参数来识别重载函数,所以必须保证重载函数的参数有所不同,即两重载函数必须具有以下两种差别之一才能分辨.

(1) 函数的参数个数不同;

(2) 一个或多个参数的类型不同.

### 运算符重载

C++的基本类型(int, char, float 等以及它们的派生类型)既能描述数据的存储格式,又能描述施加在数据上的操作,这种操作用运算符来指定。在基本类型中运算符都按系统预定义好的方式来工作。

为了使用户定义的类型与基本类型一样，C++也允许用户定义类型使用运算符来表示操作。实质上，运算符可以看成是一种函数，即运算符函数，只是对于基本类型,函数都是编译器给定的,不能加以改动. 但对于类对象,用户却可以重新定义运算符函数，以便设置运算符在类对象中新的含义。因此，定义运算符在某类对象操作的做法即所谓的运算符重载。

运算符函数可以是类的成员函数，也可以是非成员函数，如果是非成员函数，一般将它声明为该类的友元。

## 强制多态

强制多态也称强制类型转换是指将一种类型的值转换为另一种类型的值进行语义操作，从而防止类型错误。类型转换可以是隐式的，在编译时完成;也可以是显式的，可在动态运行时完成。

1.C++定义了基本数据类型之间的转换原则,从低到高：

char short int unsigned long unsigned long float double long double

2.构造函数也是一种重要的显示类型转换，构造函数进行的类型转换只能将参数类型向类类型转换。如果要把类类型的数据转换为所指定的某种数据类型，就需要使用类型转换函数。

### 基本数据类型

基本数据类型之间的类型转换，值的是当两个操作对象类型不一致时,在算术操作之前级别低的自动转换成级别高的类型.

上述规则不适用于赋值操作. 当赋值运算符右端的类型与左端的类型不同时,右端的值要转换成左端类型,然后将转换后的值赋值给左端.

类型转换可以使用下面3种强制类型转换表达式,从而可以改变编译器所使用的规则,可以按程序员自己的意愿进行所需的类型转换.

(1) static-cast < T > (E);

(2) T (E); 类型说明符(变量名)

(3) (T) E; (类型说明符)变量名

其中,E表示一个运算表达式,T表示一个类型表达式,第三种表达式是C语言中所使用的风格,在C++中,建议不要使用,应选择使用第一种形式。

例如:设对象f的类型为double，且其值为5.26，则表达式static-cast< int >(f)的值为5，类型为int。

### 自定义数据类型

自定义数据类型的转换，主要指类类型与其它数据类型之间的转换。

(1)在C++中,把其它数据类型转换成类对象是通过转换构造函数来完成的，前提是此类的转换构造函数只带1个非缺省参数。

(2)把类对象转换成其它数据类型是通过类型转换函数来完成的。类型转换函数又称为类型强制转换成员函数或转换运算符函数，他是类中的一个非静态成员函数。需要注意的是类型转换函数不能有返回值，不带任何参数，也不可以将类型转换函数定义为友元函数。

类型转换函数定义格式如下：

class<类型说明符1>

{

public：

operator<类型说明符2>();

…

}

例1:

class integer

{

public:

integer(int a) // 转换构造函数,把int a 转换为类对象

{

i = a;

printf("constructure :%d\r\n", i);

}

operator int() // 类型转换函数,把类对象转换为整型数

{

printf("operator int :%d\r\n", i);

return i;

}

private:

int i;

};

//上例可以在integer类对象与整型数之间相互转换。

int main()

{

printf("----------------begain------------------\n");

integer i1(10), i2(20); // 打印：constructure :10 constructure :20

// 使用转换运算符函数,将类对象i1转换为int后,再进行赋值

// i1 integer-->int 打印：operator int :10

int a = i1;

// 使用转换构造函数,将int a转换为integer类对象后赋给i1;

// a int-->integer 打印：constructure :10

i1 = a;

// 由于没有重载\*运算符,所以首先把i1通过转换运算符函数转换为int后

// 与2进行整数乘法运算,然后与整数10进行整数加法运算,

// 最后使用转换构造函数把最终结果转换为integer类对象后赋给i2.

// (1) i1 integer-->int 打印：operator int :10

// (2) (10 + i1 \* 2) int-->integer i2 = 10 + i1 \* 2 打印：constructure :30

i2 = 10 + i1 \* 2;

printf("-----------------end-------------------\n");

return EXIT\_SUCCESS;

}

例2:

#include<iostream.h>

class Rational

{

public:

Rational(int d,int n) //构造函数

{

den=d;

num=n;

}

operator double(); //类型强制转换成员函数的声明

private:

int den; //分数的分子

int num; //分数的分母

};

Rational::operator double() //类型强制转换成员函数的定义

{

return double(den)/double(num); //返回double类型的分数值

}

void main()

{

Rational r(5,8);

double d=4.7;

d+=r； //隐式调用类型转换成员函数，也可以使用显式调用：d+=double(r);

cout<<d<<endl; //输出结果为5.325

}

借助用户定义的类型转换,可以在多种不同类型对象之间进行混合运算,然而强制类型使类型检查复杂化,尤其在允许重载的情况下,可能会产生二义性。因此,在程序设计中要注意避免由于强制带来的二义性。

## 总结

以上就是今天要讲的内容，后续会有更多内容。

## 参考资料

版权声明：本文参考了其他资料和CSDN博主的文章，遵循CC 4.0 BY-SA版权协议，现附上原文出处链接及本声明。

1. <https://blog.csdn.net/as480133937/article/details/123740365>
2. <https://blog.csdn.net/weibo1230123/article/details/80210097>
3. LwIP应用开发实战指南：基于STM32