类的多态性有两种体现：

1．在运行时体现，通过虚函数来体现

2．在编译时体现，通过范型编程体现，一般为类模板

虚函数：

C++中函数调用默认不使用动态绑定技术，要触发动态绑定技术要满足两个条件：

1. 只有指定为虚函数的成员函数才能进行动态绑定。成员函数默认为非虚函数，需要用virtual关键字定义。
2. 必须通过基类类型的引用（如果直接将派生类的对象赋给基类的对象，则不会触发动态绑定）或指针进行动态绑定函数调用。（如果将派生类指针强行指向基类的对象，不会触发动态绑定技术，虚函数不会有效，只会调用基类的函数。）

从派生类到基类的转换：

因为每个派生类对象都包含基类部分，所以可以将基类的引用绑定到派生类对象上（绑定到了派生类的基类部分），也可以用基类指针指向派生类对象。

例:程序class\_test14

// shape.h

#ifndef SHAPE\_H

#define SHAPE\_H

class Shape // 基类

{

public:

Shape(double x,double y);

virtual double area() const;//定义了一个const常量虚函数

//const常量成员函数相当于一个“只读”函数，即不对对象作修改

// virtual不能出现类外部函数实现中

virtual Shape& f();

protected:

double xCoord;

double yCoord;

};

#endif

shape.cpp

#include "shape.h"

#include <iostream>

using namespace std;

Shape::Shape(double x,double y):xCoord(x), yCoord(y)

{

}

double Shape::area() const //const常量成员函数，在函数原型声明和函数定义处都必须加const限定

{

cout << "调用Shape类的area函数" << endl;

return 0.0;

}

Shape& Shape::f()

{

cout << "调用Shape类的f()函数" << endl;

xCoord = 0;

return \*this;

}

circle.h

#ifndef CIRCLE\_H

#define CIRCLE\_H

#include "shape.h"//一定要添加基类的头文件

class Circle:public Shape

{

public:

Circle(double x,double y,double r);

virtual double area() const;

Circle& f(); // 尽管在返回值上与基类Shape中f()的返回值不同，但f()仍然是虚函数

protected:

double radius;

};

#endif

circle.cpp

#include "circle.h"

#include <iostream>

using namespace std;

Circle::Circle(double x,double y,double r)

:Shape(x,y) //注意：Shape类没有默认的构造函数，所有需要用冒号表达式来调用其构造函数

{

radius = r;

}

double Circle::area() const

{

cout << "调用Circle类的area函数" << endl;

return 3.14 \* radius \* radius;

}

Circle& Circle::f()

{

cout << "调用Circle类的f()函数" << endl;

radius = 2.0;

return \*this;

}

rectangle.h

#ifndef RECTANGLE\_H

#define RECTANGLE\_H

#include "shape.h"//一定要添加基类的头文件

class Rectangle:public Shape

{

public:

Rectangle(double x, double y, double x1, double y1);

virtual double area() const;

protected:

double x2Coord;

double y2Coord;

};

#endif

rectangle.cpp

#include "rectangle.h"

#include <iostream>

#include <math.h>

using namespace std;

Rectangle::Rectangle(double x, double y, double x1, double y1) : Shape(x,y)

{

x2Coord = x1;

y2Coord = y1;

}

double Rectangle::area() const

{

cout << "调用Rectangle类的area函数" << endl;

return fabs((xCoord - x2Coord ) \* (yCoord - y2Coord));

}

test.cpp

// 类的多态性

#include <iostream>

#include "shape.h"

#include "circle.h"

#include "rectangle.h"

using namespace std;

void func(const Shape& sp); //在func函数中不允许对Shape类的引用sp作修改

void func2(Shape& sp);

int main(int argc,char\* argv[])

{

Circle circle(2.0,5.0,4.0);

// 调用func()函数时，实际是将基类的引用绑定到了派生类对象上，会触发动态绑定

// 注意：如果func()的形参不是引用，而直接定义成func(const Shape sp)，则不会触发动态绑定技术，虚函数不会有效，全部调用基类Shape的area()函数

func(circle);

Rectangle rect(2.0, 4.0, 1.0, 2.0);

func(rect);

func2(circle);

return 0;

}

void func(const Shape& sp)

{

cout << sp.area() << endl;

}

void func2(Shape& sp)

{

sp.f(); // 会触发动态绑定技术，因为f()是虚函数

}

输出为：

调用Circle类的area函数

50.24

调用Rectangle类的area函数

2

调用Circle类的f()函数

C++动态绑定技术的关键：基类的引用和指针的静态类型（编译时的类型）和动态类型（运行时的实际类型）可以不同。

产生这种不同的原因：可以使用基类类型的指针或引用来引用派生类型对象，所以，使用基类类型的引用或指针时，不知道指针或引用所绑定的对象的类型：基类类型的引用或指针可以引用基类类型对象，也可以引用派生类型对象。

通过基类引用或指针调用基类的函数，如果调用非虚函数，则无论实际对象是什么类型，都会调用基类定义的函数。如果调用虚函数，则直到运行时，才能确定调用引用所绑定的对象或指针所指向的对象所属类型定义的函数。

注意：如果直接通过基类的对象来调用虚函数，则不会触发动态绑定（无论赋给它的是本身的对象，还是派生类的对象），因为对象的静态类型和动态类型是相同的。另外，如果强制将派生类的指针指向基类的对象（这种做法是不安全的），通过该指针只能调用基类的函数，同样也不会触发动态绑定。见下例：

例：程序class\_test15

// 类的多态性

#include <iostream>

using namespace std;

class A // 基类

{

public:

virtual A\* f()

{

cout << "A的f()" << endl;

return this

}

};

class B : public A // 派生类

{

public:

virtual B\* f()

{

cout << "B的f()" << endl;

return this;

}

};

int main(int argc, char\* argv[])

{

A\* a\_ptr = new A;

a\_ptr->f(); // 执行A的f()

delete a\_ptr;

a\_ptr = NULL;

a\_ptr = new B; // 基类A指针指向派生类B对象

a\_ptr->f(); // 执行B的f()

delete a\_ptr;

a\_ptr = NULL;

B b;

A& a\_ref = b; // 基类A引用绑定派生类B对象

a\_ref.f(); // 执行B的f()

A a = b;

a.f(); // 执行A的f()，基类A的对象直接调用，不触发动态绑定

B\* b\_ptr = NULL;

a\_ptr = new A;

b\_ptr = (B\*)a\_ptr; // 这样做是不安全的。

b\_ptr->f(); // 执行A的f()，b\_ptr只能调用基类A的函数

delete a\_ptr;

a\_ptr = NULL;

b\_ptr = NULL;

return 0;

}

虚函数的使用要求：

1. 只有类的成员函数才能声明为虚函数，普通函数不能声明为虚函数。
2. 虚函数的声明应该在基类中，而且一旦在基类中声明为虚函数，就一直为虚函数。如果在派生类中声明为虚函数，而基类中未声明，则基类的引用或指针调用该函数时，不触发动态绑定。
3. 派生类中虚函数的声明必须与基类的定义方式完全匹配，但有一个例外：返回对基类的引用（或指针）的虚函数，派生类中的虚函数可以返回基类函数所返回类型的派生类的引用(或指针)。见上例中的f()函数。
4. 静态成员函数不能是虚函数，因为其不包含this指针，不关联任何对象，而虚函数的调用是通过对象来实现的，在语法上也不能将static和virtual同时使用。
5. 内联（inline）函数不能是虚函数，因为内联函数在编译期间就在调用点展开，而虚函数则是在运行时在调用点上根据实际的对象类型确定函数调用。但在语法时，可以将函数同时声明为inline和virtual，因为inline函数声明只是向编译器建议，而具体是否真正为inline函数，则需要编译器决定。inline和virtual不会同时起作用。
6. 构造函数不能是虚函数，因为虚函数是通过实例对象中的虚函数表指针的调用的，如果构造函数是虚函数，执行构造函数之前实例对象不存在，就无法调用虚函数，所以构造函数不能为虚函数。虚函数是通过基类的指针或引用去调用派生类的成员函数，而构造函数是自动调用的，不是通过基类的指针或引用去调用，所以构造函数不能为虚函数。
7. 析构函数可以是虚函数，而且通常声明为虚函数，这是考虑到虚析构的情况。
8. 赋值运算符重载时也不应该声明为虚函数，没有什么用处。

覆盖虚函数机制，见下例：

// 覆盖虚函数机制

#include <iostream>

class Base

{

public:

virtual void func()

{

number = 5;

std::cout << "number = " << number << std::endl;

}

private:

int number;

};

class Derive : public Base

{

public:

void func()

{

count = 10;

std::cout << "count = " << count << std::endl;

}

private:

int count;

};

int main(int argc, char\* argv[])

{

Derive derive;

Base\* b\_ptr = &derive;

b\_ptr->Base::func(); // 调用基类Base的func()函数，通过作用域操作符::覆盖了虚函数机制

return 0;

}

之所以希望覆盖虚函数机制，是为了让派生类的虚函数调用基类中的版本，这样基类中的版本可以完成继承层次中的所有公共任务，而每个派生类添加自己的特殊工作，见下例：

例：程序class\_test16

// 覆盖虚函数机制

#include <iostream>

class Base

{

public:

virtual void func()

{

number = 5;

std::cout << "number = " << number << std::endl;

}

private:

int number;

};

class Derive : public Base

{

public:

void func()

{

Base::func(); // 不能少了作用域操作符::，否则会引起无穷递归调用

count = 10;

std::cout << "count = " << count << std::endl;

}

private:

int count;

};

int main(int argc, char\* argv[])

{

Derive derive;

Base\* b\_ptr = &derive;

b\_ptr->func();

return 0;

}

虚函数与默认实参

虚函数可以带有默认实参，但默认实参是在编译时确定，而与动态类型无关，所以有这样的特点：通过基类的引用或指针调用虚函数，默认实参为在基类虚函数声明中指定的值，如果通过派生类的指针或引用调用虚函数时，默认实参为在派生类中的版本中声明的值。

在同一虚函数的基类版本和派生类版本中的默认实参最好相同，否则可能会引起麻烦，见下例：

例：程序class\_test17

// 虚函数与默认实参

#include <iostream>

class Base

{

public:

virtual void func(char\* name = "base")

{

std::cout << "调用基类Base的func()函数" << name << std::endl;

}

};

class Derive : public Base

{

public:

void func(char\* name = "derive")

{

std::cout << "调用派生类Dervie的func()函数" << name << std::endl;

}

};

int main(int argc, char\* argv[])

{

Derive derive;

Base\* b\_ptr = &derive;

b\_ptr->func();

return 0;

}

输出为：

调用派生类Dervie的func()函数base

可以看出：调用了派生类中的func()函数，但默认值却是基类版本中定义的默认实参。

C++类的多态性主要的用途：

多态性是为了接口重用（参考之前的Shape，Circle，Rectangle构成的继承程序中的area()函数）。在进行面向对象的程序设计时，可以在基类中定义接口，不实现具体的功能，而在派生类中进行具体的实现，不同的派生类对基类中的同一个接口进行不同的实现，这样整个程序可以由不同的开发人员共同实现。