**实 验 报 告**

**课程名称：** 面向对象技术(C++)

**实验项目：** 虚函数与多态性

**实验仪器：** 计算机

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **项目** | **报告格式**  **Report format** | **代码质量**  **Code quality** | **注释质量**  **Comment quality** | **逻辑或思想描述**  **Necessitate logical description** | **独创性**  **Originality** | **合计**  **Total** |
| **百分比(%)**  **percentage** | **10** | **25** | **25** | **25** | **15** | **100** |
| 得分（score） |  |  |  |  |  |  |

**系 别：** 计算机系

**专 业：** 计算机大类

**班级姓名：** 计类1805/刘科宏

**学 号：**  2018011252

**日 期：** 5月4日

**成 绩：**

**同组成员：**

**指导教师：** 丁 濛

# 实验目的 (Objects)

1. 了解虚函数的重要意义；

Understand the reason why we need virtual methods.

1. 学习和掌握虚函数的声明方式；

Learn how to declare a virtual method.

1. 学习和掌握虚函数的使用，编程验证虚函数和非虚函数的调用结果；

Learn how to use virtual methods, and verify the different result of calling virtual and non-virtual functions.

1. 了解纯虚函数的作用与意义，学习抽象基类的声明与使用；

Understand the meaning of pure virtual functions and the intention of introducing such function. Learn how to declare and use an abstract base class.

# 实验内容 (Contents)

1. 将实验三中的Person，National\_Person，Student，Chinese，Chinses\_Student类的printProperty()声明为虚函数，用基类指针分别指向各个派生类对象，通过该基类指针访问printProperty函数，观察调用结果；

Revise each printPropery() method in class Person，National\_Person，Student，Chinese and Chinses\_Student defined previously in Experiment 3 to virtual function. Calling printProperty() method through a base-class type pointer which points each derived object one by one, and observe the calling result.

1. 设计测试用例，说明基类的析构函数声明为虚函数的意义;

Design a test case to explain why we should declare the destroyer of a base-class as a virtual one.

1. 设计基类 Figure，该类包含一个公有虚函数 double getArea()。以Figure为基类，自行设计派生类 矩形: Rectangle和 圆: Circle，以Rectangle为基类，设计正方形类 Square。编写一个函数compare，其功能是比较上述三种图形中任意两种图形的面积大小，如果第一个图形面积小于第二个图形的面积，则返回true，否则返回false。编写main函数验证以上各项内容。Figure类是否可以实例化？

Design a base-class Figure, which includes a virtual public function *double getArea().*

Design derived-class Rectangle, Circle which inherit from Figure. Declare and define whatever members you think are necessary.

Design a derived-class Square inherits from Rectangle. Declare and define whatever members you think are necessary.

Write a function compare, which will compare the area of any two figures from Rectangle, Circle and Square, return true if the first figure is smaller than the second one, return false if not.

Write a main function to verify your design. Can you instantiate an object directly from Figure?

1. 将3中的Figure类改造为抽象基类，在这种情况下 Figure类是否还可以被实例化？

Redefine class Figure in 3 as an abstract class. In this circumstance, can you instantiate an object directly from Figure?

# 实验内容 (Your steps or codes, and Results)

1.

#include <iostream>

#include <string>

#include <stdio.h>

using namespace std;

class Person{

protected://类成员必须得设置为保护型以便之后输出

string name;

char gender;

int year;

int month;

int day;

bool legalBirthday(const int y, const int m, const int d) const {

if(y< 0 || m < 0 || m > 12 || d < 0 || d > 31)

return false;

int a[12] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

if(m == 2) {

if( y% 4 != 0 || (y% 100 == 0 && y% 400 != 0)) {

if( d <= 28)

return true;

else

return false;

}

else {

if( d <= 29)

return true;

else

return false;

}

}

if(d <= a[m - 1])

return true;

else

return false;

}

public:

Person() : name(), gender('U'), year(1900), month(1), day(1){

printf("Person()\n");

}

Person(const string& n, char g, int y, int m, int d) : name(n), gender(g) {

printf("Person(const string&, char, int, int, int)\n");

if(legalBirthday(y, m, d)){

year = y, month = m, day = d;

}

}

Person(const Person& other) : name(other.name), gender(other.gender), year(other.year), month(other.month), day(other.day){

printf("Person(const Person&)\n");

}

Person(const string& n)

{

name = n;

}

~Person(){

printf("~Person()\n");

}

void setGender(char g) { gender = g;}

void setBirthday(int y, int m, int d) {

if(legalBirthday(y, m, d)){

year = y, month = m, day = d;

}

}

void setName(const char \*str){

name = str;

}

virtual void printProperty() const {//设置为虚函数，因为在其派生类当中有同名作用类似的函数，这样会方便基类指针指向使调用的方便

cout << "Person::printProperty()\n";//表明是Person类的输出

cout << "Name:" << name << "\n";

cout << "Gender:" << gender << "\n";

printf("Birthday:%d-%.2d-%.2d\n", year, month, day);

}

};

class National\_Person : virtual public Person{

protected:

string Nationality;

public:

National\_Person() : Nationality() {

cout << "National\_Person()\n";

}

National\_Person(const string& n, const string& na, char g, int y, int m, int d) : Person( n, g, y, m, d),Nationality(na)

{

cout << "National\_Person(const string&, const string&, char, int, int, int)\n";

}

National\_Person(const string& na) : Nationality(na){}

~National\_Person()

{

printf("~National\_Person()\n");

}

void setNationality(const string& nat)

{

Nationality = nat;

}

void printNationality()

{

Person::printProperty();

cout << "Nationality:" << Nationality << '\n';

}

void printProperty ()const

{

cout << "National\_Person::printProperty()\n";//表明是National\_Person类的输出

cout << "Name:" << name << "\n";

cout << "Gender:" << gender << "\n";

printf("Birthday:%d-%.2d-%.2d\n", year, month, day);

cout << "Nationality:" << Nationality << '\n';

}

};

class Chinese : public National\_Person{

public:

public:

Chinese() : National\_Person("CHN"){}

Chinese(const string& n, char g, int y, int m, int d) :National\_Person(n,"CHN",g,y,m,d){

printf("Chinese(const string&, char, int, int, int)\n");

}

~Chinese()

{

cout << "~Chinese()\n";

}

void printProperty ()const

{

cout << "Chinese::printProperty()\n";//表明是Chinese类的输出

cout << "National\_Person::printProperty()\n";

cout << "Name:" << name << "\n";

cout << "Gender:" << gender << "\n";

printf("Birthday:%d-%.2d-%.2d\n", year, month, day);

cout << "Nationality:" << Nationality << '\n';

}

};

class Student : virtual public Person{

protected:

string schoolName;

string studentID;

int grade;

public:

Student() : schoolName(),studentID(),grade(0){

cout << "Student()\n";

}

Student(const string& name, char g,int y,int m,int d,const string& sname,const string& sID,int \_grade) :

Person(name,g,y,m,d),schoolName(sname),studentID(sID),grade(\_grade){

cout << "Student(const string&, char, int, int, int, const string&, const string&, int)\n";

}

~Student()

{

cout << "~Student()\n";

}

void setSchoolName(const string& N)

{

schoolName = N;

}

void setStudentID(const string& ID)

{

studentID = ID;

}

void setGrade(int a)

{

grade = a;

}

void printSchoolName()

{

cout << schoolName << '\n';

}

void printStudentID()

{

cout << studentID << '\n';

}

void printGrade()

{

cout << grade << '\n';

}

void printProperty ()const

{

cout << "Student::printProperty()\n";//表明是Student类的输出

cout << "Name:" << name << "\n";

cout << "Gender:" << gender << "\n";

printf("Birthday:%d-%.2d-%.2d\n", year, month, day);

cout << "School Name:" << schoolName << '\n';

cout << "Student ID:" << studentID << '\n';

cout << "Grade:" << grade << '\n';

}

};

class Chinese\_Student : public Student,public Chinese{

private:

public:

Chinese\_Student() {

cout << "Chinese\_Student()\n";

}

Chinese\_Student(const string& name, char g,int y,int m,int d,const string& sname,const string& sID,int \_grade) :

Student( name, g, y, m, d, sname, sID, \_grade),Chinese(name, g, y, m, d), Person( name,g, y, m, d){

cout << "Chinese\_Student(const string&, char, int, int, int, const string&, const string&, int)\n";

}

~Chinese\_Student()

{

cout << "~Chinese\_Student()\n";

}

void printProperty() const{

cout << "Chinese\_Student::printProperty()\n";//表明是Chinese\_Student类的输出

cout << "Nationality:" << Nationality << '\n';

cout << "Name:" << name << "\n";

cout << "Gender:" << gender << "\n";

printf("Birthday:%d-%.2d-%.2d\n", year, month, day);

cout << "School Name:" << schoolName << '\n';

cout << "Student ID:" << studentID << '\n';

cout << "Grade:" << grade << '\n';

}

};

int main() {

Person \*p;//基类指针

Chinese chnlkh("LEO", 'M', 1999, 10, 6);

p = &chnlkh;

p->printProperty();

cout << '\n';

National\_Person nalkh("LEO", "CHN", 'M', 1999, 10, 6);

p = &nalkh;

p->printProperty();

cout << '\n';

Student stulkh("LEO", 'M', 1999, 10, 6, "BISTU", "2018011252", 100);

p = &stulkh;

p->printProperty();

cout << '\n';

Person lkh("LEO", 'M', 1999, 10, 6);

p = &lkh;

p->printProperty();

cout << '\n';

Chinese\_Student cstulkh("LEO", 'M', 1999, 10, 6, "BISTU", "2018011252", 100);

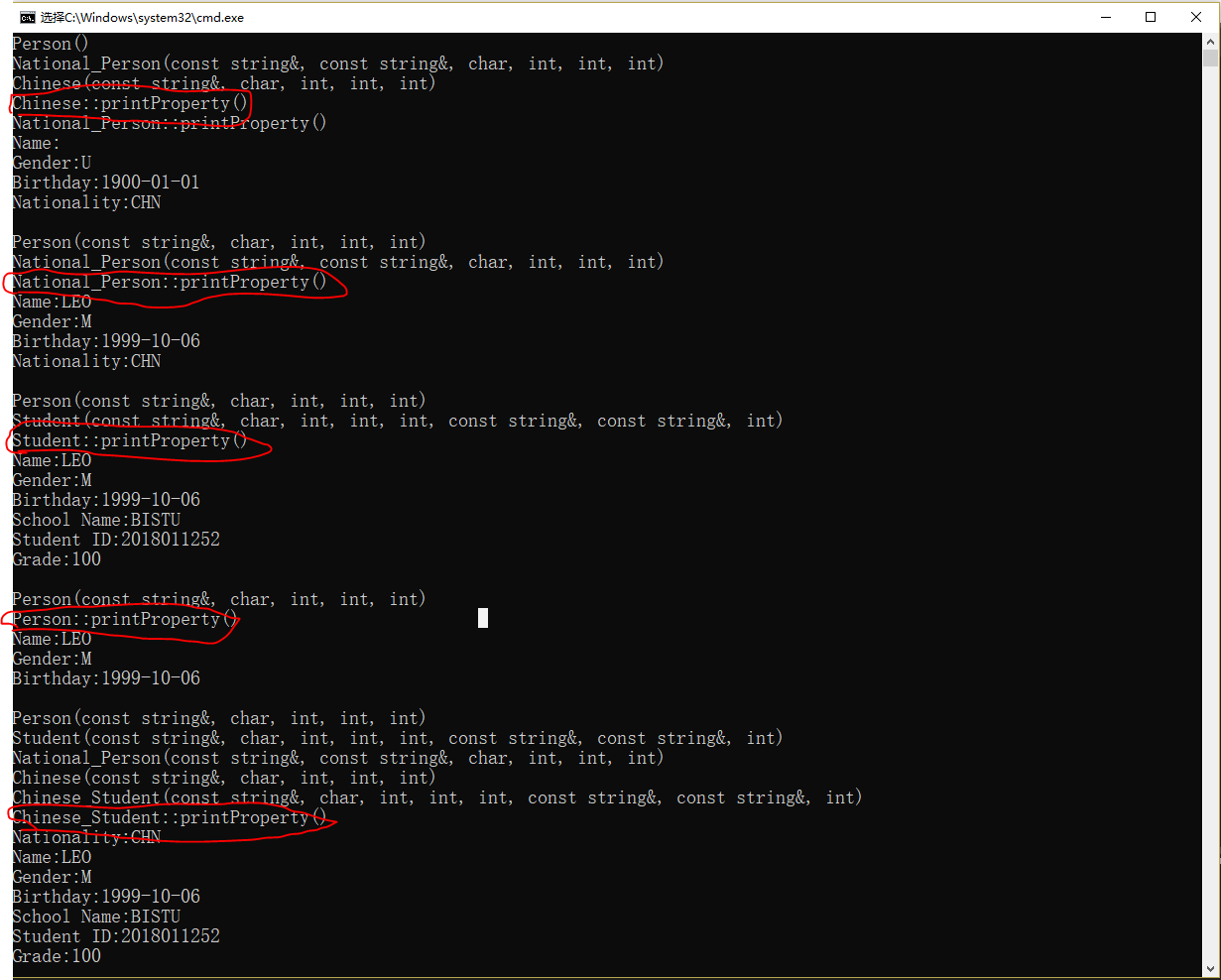
p = &cstulkh;

p->printProperty();

return 0;

}

执行结果：



图表 1：实验一执行结果

红笔表示了来自于不同类的函数。

2.

首先是不把基类析构函数设为虚函数时

#include<iostream>

#include<stdio.h>

using namespace std;

class A {

private:

int \*a;

public:

A() {

cout << "A()\n";//表明使用到了A类的构造函数，下B类同理

a = new int[10];//在A类构造时开出10个int的空间

}

~A() {

cout << "~A()\n";//表明使用到了A类的析构函数

delete[] a;//在A析构时释放出A开出的空间

}

};

class B :public A {

private:

int \*b;

public:

B() {

cout << "B()\n";

b = new int[10];//在B类构造时开出10个int的空间

}

~B() {

cout << "~B()\n";

delete[] b;//释放在构造时开出的空间

}

};

int main()

{

A \*a;

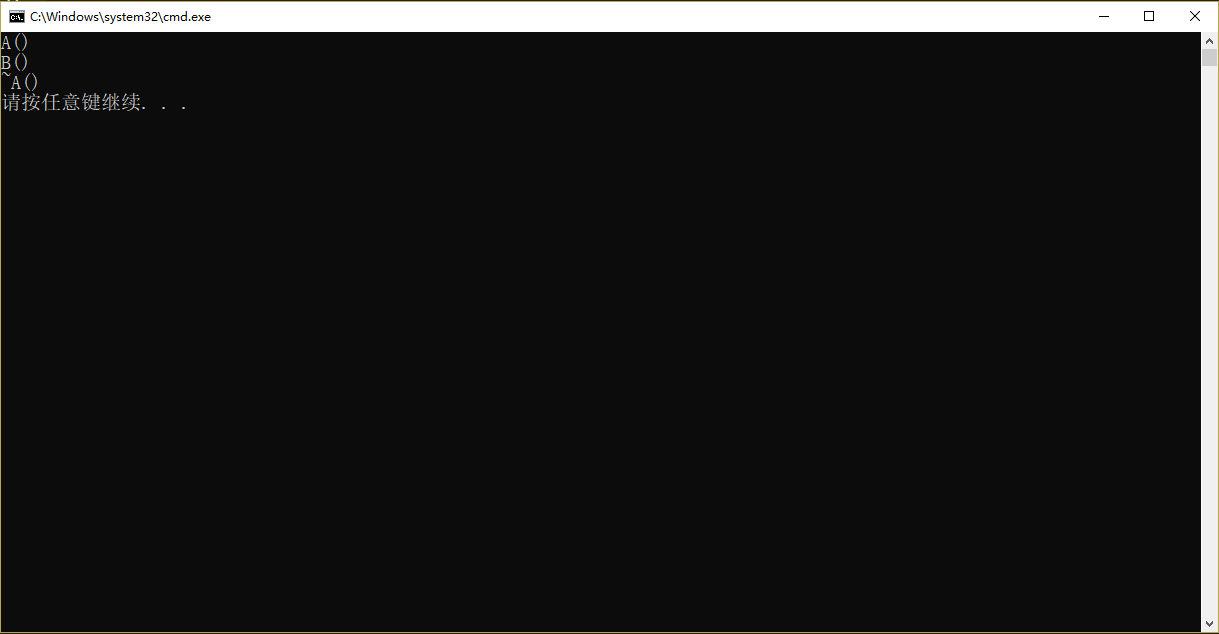
a = new B;//使用基类指针开出派生类的空间

delete a;//删去a开出的空间

return 0;

}

执行结果：



图表 2：实验2执行结果（1）

这个结果中可以看出，在构造函数时，同时调用了A与B两个构造函数，而析构时仅仅调用了A的析构函数，这说明构造A类时开出来的空间被释放了，而对于B类构造开出来的空间并未被释放出来，导致了内存泄漏。

当把基类析构函数变为虚函数后

#include<iostream>

#include<stdio.h>

using namespace std;

class A {

private:

int \*a;

public:

A() {

cout << "A()\n";//表明使用到了A类的构造函数，下B类同理

a = new int[10];//在A类构造时开出10个int的空间

}

virtual ~A() {

cout << "~A()\n";//表明使用到了A类的析构函数

delete[] a;//在A析构时释放出A开出的空间

}

};

class B :public A {

private:

int \*b;

public:

B() {

cout << "B()\n";

b = new int[10];//在B类构造时开出10个int的空间

}

~B() {

cout << "~B()\n";

delete[] b;//释放在构造时开出的空间

}

};

int main()

{

A \*a;

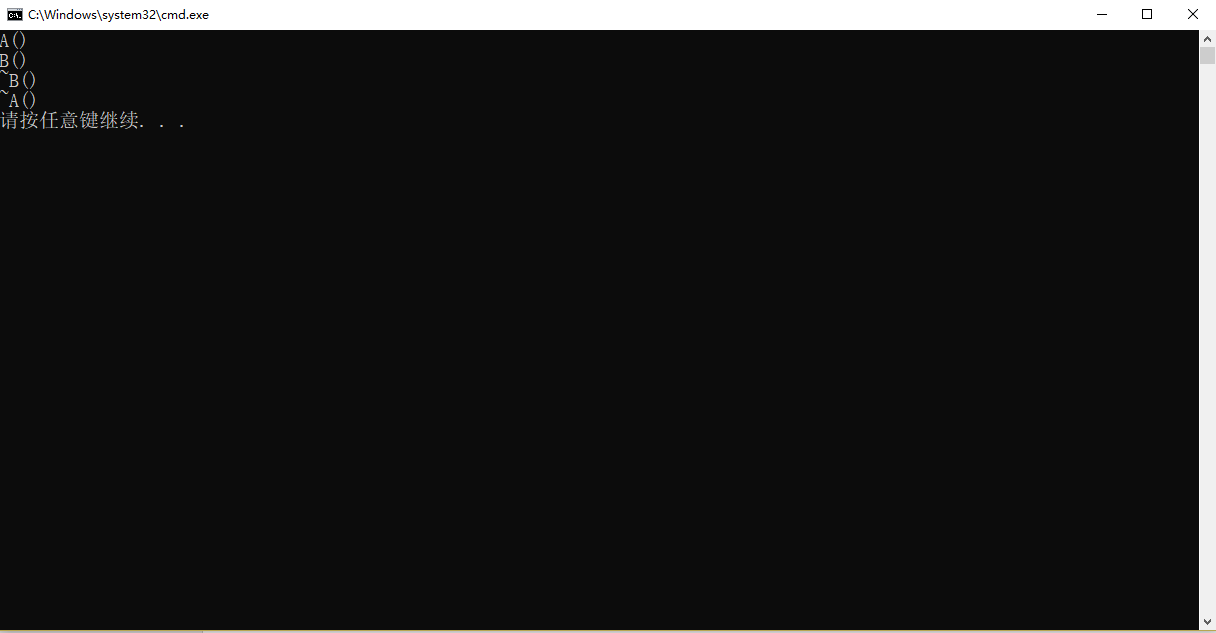
a = new B;//使用基类指针开出派生类的空间

delete a;//删去a开出的空间

return 0;

}

执行结果：



图表 3：实验二执行结果（2）

从这里可以看出，构造与析构都调用了A与B这两个类的函数，说明开出的空间可以被正常的释放，不会造成内存泄漏的情况发生。

3.

#include <iostream>

#include <string>

using namespace std;

class Figure

{

public:

constexpr static double pi = 3.1415927;

virtual double getArea() const {return 0;}//虚函数，方便用于之后的图形求面积，由于是double类型，因此还有一个return 0；

virtual void print() const {}//虚函数，方便之后的图形面积结果的输出

};

class Rectangle : public Figure//矩形派生于图形这个基类

{

protected:

double a;

double b;

public:

Rectangle() : a(0),b(0){}//必须有自己的构造函数，否则无法单独定义

Rectangle(double \_a, double \_b) : a(\_a),b(\_b){}

double getArea() const

{

return a \* b;

}

void print() const

{

cout << "Rectangle:" << getArea();

}

};

class Circle : public Figure//圆形

{

private:

double r;

public:

Circle() : r(0){}

Circle(double a) : r(a){}

double getArea() const

{

return pi\*r\*r;

}

void print() const

{

cout << "Circle:" << getArea();

}

};

class Square : public Rectangle//正方形，因为他是特殊的矩形，因此派生于Rectangle

{

public:

Square() : Rectangle(){}

Square(double \_a) : Rectangle( \_a, \_a){}

double getArea() const

{

return a \* b;

}

void print() const

{

cout << "Square:" << getArea();

}

};

bool compare(Figure \*p, Figure \*q)//由于判断图形面积大小的函数，传入的变量被基类指针指向，因为基类函数为纯虚函数，所以计算面积会调用自己类的计算面积函数

{

if(p->getArea() < q->getArea())

return true;

else

return false;

}

int main(){

Figure A;//尝试实体化基类

Rectangle rect(3. , 5.);

Circle circle(8. / Figure::pi);

Square square(4.);

Figure \*pList [] = {&rect, &circle, &square};

Figure \*min = pList[0];

for(int i = 1; i < 3; ++i){

if(compare(pList[i], min)){

pList[i]->print();

printf(" < ");

min->print();

printf("\n");

min = pList[i];

}

else{

min->print();

printf(" < ");

pList[i]->print();

printf("\n");

}

}

printf("Minimum:\n");

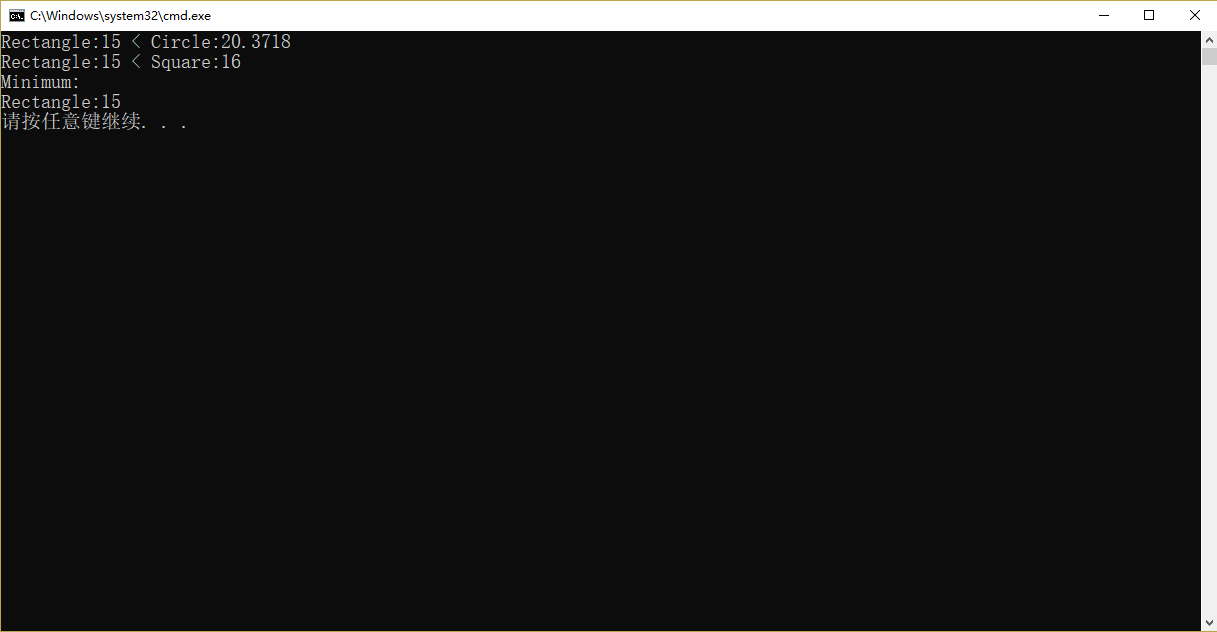
min->print();

printf("\n");

return 0;

}

执行结果



图表 4：实验三执行结果

实验结果证明，在基类没有包含纯虚函数的情况下是可以被实体化的。

4.

#include <iostream>

#include <string>

using namespace std;

class Figure//创建出一个抽象基类，方便之后的指针指向派生类函数的调用

{

public:

constexpr static double pi = 3.1415927;

virtual double getArea() const = 0;//纯虚函数，用于之后的图形求面积

virtual void print() const = 0;//纯虚函数，由于之后的图形面积结果的输出

};

class Rectangle : public Figure//矩形派生于图形这个基类

{

protected:

double a;

double b;

public:

Rectangle() : a(0),b(0){}//必须有自己的构造函数，否则无法单独定义

Rectangle(double \_a, double \_b) : a(\_a),b(\_b){}

double getArea() const

{

return a \* b;

}

void print() const

{

cout << "Rectangle:" << getArea();

}

};

class Circle : public Figure//圆形

{

private:

double r;

public:

Circle() : r(0){}

Circle(double a) : r(a){}

double getArea() const

{

return pi\*r\*r;

}

void print() const

{

cout << "Circle:" << getArea();

}

};

class Square : public Rectangle//正方形，因为他是特殊的矩形，因此派生于Rectangle

{

public:

Square() : Rectangle(){}

Square(double \_a) : Rectangle( \_a, \_a){}

double getArea() const

{

return a \* b;

}

void print() const

{

cout << "Square:" << getArea();

}

};

bool compare(Figure \*p, Figure \*q)//由于判断图形面积大小的函数，传入的变量被基类指针指向，因为基类函数为纯虚函数，所以计算面积会调用自己类的计算面积函数

{

if(p->getArea() < q->getArea())

return true;

else

return false;

}

int main(){

Figure A;

Rectangle rect(3. , 5.);

Circle circle(8. / Figure::pi);

Square square(4.);

Figure \*pList [] = {&rect, &circle, &square};

Figure \*min = pList[0];

for(int i = 1; i < 3; ++i){

if(compare(pList[i], min)){

pList[i]->print();

printf(" < ");

min->print();

printf("\n");

min = pList[i];

}

else{

min->print();

printf(" < ");

pList[i]->print();

printf("\n");

}

}

printf("Minimum:\n");

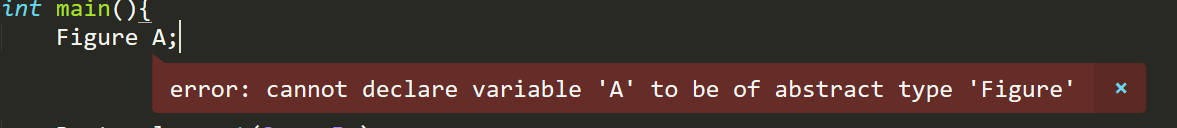
min->print();

printf("\n");

return 0;

}

执行结果：



图表 5：实验四执行结果

从结果明显看出作为抽象基类Figure无法实体化。

# 实验总结 (Conclusion)

在这一次实验当中，学习的主要内容便就是“多态”，对于多态的更多了解强化了我对于虚函数与抽象类的认识，也真正意义上有了对虚函数的有效理由，发现其对于代码的优化与简洁方便有着重大联系，对于之后的写代码能力有着大幅度的提升

1. 了解到了虚函数的运作方式，若一个类当中有虚函数的话，这个类的内存首地址会出现一个指针vptr，这个指针指向相应这个类的虚函数表，这个虚函数表里有着这个类对于函数应该调用哪一个。
2. 对于析构函数的虚函数也有了很好的理解，明白了写代码时应该注重每一个小的细节，一但有疏忽就会有者不可估量的后果，而且难以被发现，当然也有的基类根本用不着对析构函数的虚化，若盲目虚化析构函数，可能会使得代码变得繁琐以及复杂。
3. 了解到了纯虚函数的存在以及它存在的必要性，也明白了它不能被实体化，定义纯虚函数的目的在于，可以使的代码的重复内容减少，优化代码内容。