#include "iostream"//包含c++的头文件

using namespace std;//使用命名空间 std 标准的命名空间 （在这个命名空间中定义了很多标准定义）

void main01()

{

//printf("hollo world");

//cout标准输出相当于黑屏幕，<<左移操作符在c++里面进行了功能的改造（增强）====》c++语言重载、、3day

//endl 回车换行\n

//打印到屏幕并且回车换行

cout << "hollo...." << endl;

system("pause");

}

//求圆的面积

//用面向过程的方法求解了圆的面积

//<< >>不会用；标准输入cin和标准输出cout始终写在操作符的左边

void main02()

{

double r = 0;

double s = 0;

cout << "请输入圆形的半径：";

//cin 标准输入 代表键盘

cin >> r;

cout << "r的值时：" << r << endl;

s = 3.14\*r\*r;

cout << "圆的面积是s：" << s << endl;

system("pause");

}

//在C++中定义了一个自定义数据类型

struct Circle

{

double m\_s;//圆的面积

double m\_r;//圆的半径

};

//在C++中定义了一个自定义数据类型MyCicle圆形，和c不一样的地方在于在类里面可以写函数

class Mycicle

{

public:

double m\_s;//圆的面积 属性 成员变量

double m\_r;//圆的半径

public:

void setR(double r)//成员函数

{

m\_r = r;//给此成员变量赋值

}

double getR()

{

return m\_r;

}

double getS()

{

m\_s = 3.14\*m\_r\*m\_r;

return m\_s;

}

protected:

private:

};

void main03()

{

Mycicle c1, c2, c3;//用类 定义 变量 对象

double r;

cout << "请输入c1圆形的半径:";

cin >> r;

//给c1圆形的属性赋值

c1.setR(r);//运行到此处，调用C1对象的类的成员函数

cout << "请输入c1图形的半径" << c1.getS() << endl;

system("pause");

}

//用面向对象的方法

//1 类的抽象 成员变量和成员函数

//2 实例化 类的对象

//3 求面积

//面向过程加工的是：一个一个的函数

//面向对象加工的是：一个一个的类

//4 main 集成测试

//思考1 类的调用 执行过程分析==》类代码并不是一步一步执行的

//思考2 类是一个数据类型（固定大小内存块的别名），定义一个类是一个抽象的概念，不会给你分配内存，

//用数据类型定义变量的时候，才会分配内存

void main04()

{

Mycicle c1, c2, c3;//用类 定义 变量 对象

double r1,r2,r3;

cout << "请输入c1圆形的半径r1:";

cin >> r1;

//给c1圆形的属性赋值

c1.setR(r1);//运行到此处，调用C1对象的类的成员函数

cout << "请输入c1图形的面积c1：" << c1.getS() << endl;

system("pause");

//圆2

cout << "请输入c2圆形的半径r2:";

cin >> r2;

//给c2圆形的属性赋值

c2.setR(r2);//运行到此处，调用C1对象的类的成员函数

cout << "请输入c2图形的面积c2： " << c2.getS() << endl;

system("pause");

//圆3

cout << "请输入c3圆形的半径r3:";

cin >> r3;

//给c3圆形的属性赋值

c3.setR(r3);//运行到此处，调用C1对象的类的成员函数

cout << "请输入c3图形的面积c3：" << c3.getS() << endl;

system("pause");

}

//c++编译器是如何区分c1对象，c2，c3对象调用了类的方法（如何处理多个对象调用单个函数的）

//2.class类是一个数据类型和对象1;n的关系

//学习c++的方法是和面向过程是不一样的

system("pause");

}

类把属性和方法做了一个封装



找不到32win控制台解决方式

c++的优化部分

c++可在程序中添加变量

c++中如何快速注释

1,先选中要注释的代码段   
2，按住ctrl+k+c注释本段代码   
3，按住ctrl+k+u取消注释本段代码

#include "iostream"

using namespace std;

//求圆形的面积

//封装有2层含义

//1 把属性和方法进行封装

//2 对属性和方法进行访问控制

//3 类的内部和类的外部

//4 类的访问控制关键字

//public: 修饰的成员变量和函数 可以在类的内部和类的外部访问

//private: 修饰的成员变量和函数 只能在类的内部被访问,不能在类的外部访问

//protected: 修饰的成员变量和函数 只能在类的内部被访问,不能在类的外部访问, 用在继承里面

class MyCircle

{

public:

double m\_r; //属性 成员变量

double m\_s;

public:

double getR()

{

return m\_r;

}

void setR(double r) //成员函数

{

m\_r = r;

}

double getS()

{

m\_s = 3.14\*m\_r\*m\_r;

return m\_s;

}

protected:

private:

};

//类的封装的威力

//类封装的成员变量和成员函数 //可以调用类的成员函数

void printCircle01(MyCircle \*pC)

{

cout<<"r"<<pC->getR()<<endl;

cout<<"s"<<pC->getS()<<endl;

}

void printCircle02(MyCircle &myc)

{

cout<<myc.getS()<<endl;

}

void printCircle03(MyCircle myc)

{

}

void main\_类的封装第一层含有()

{

MyCircle c1, c2;

c1.setR(10);

cout<<"c1 s: "<<c1.getS()<<endl;

c1.setR(11);

printCircle01(&c1);

c2.setR(20);

printCircle01(&c2);

//引用

printCircle02(c2);

cout<<"hello...."<<endl;

system("pause");

}

class MyAdvCircle

{

private:

double m\_r2; //没有权限修饰的成员变量 函数 是 默认是私有属性

private:

double m\_r; //属性 成员变量

double m\_s; //相当于你的银行卡密码

public:

double getR() //你的名字

{

return m\_r;

}

void setR(double r) //成员函数

{

m\_r = r;

}

double getS()

{

m\_s = 3.14\*m\_r\*m\_r;

return m\_s;

}

};

void main()

{

MyAdvCircle c1, c2;

//c1.m\_r = 10;

double m=c1.setR(10);

cout << m << endl;

system("pause");

}

struct AAA

{

int a; //用struct关键字定义的类,不写,默认属性是public的.

//class private

public:

protected:

private:

};

void main02()

{

AAA a1;

a1.a = 11;

system("pause");

}

声明头文件时，头文件一定要和文件名一样

面向过程面向对象转变案例01

// 案例1 设计立方体类(cube)，求出立方体的面积和体积

// 求两个立方体，是否相等（全局函数和成员函数）

#include <iostream>

using namespace std;

class Cube

{

public:

void setA(int a)

{

m\_a = a;

}

void setB(int b)

{

m\_b = b;

}

void setC(int c)

{

m\_c = c;

}

void setABC(int a=0, int b=0, int c=0)

{

m\_a = a; m\_b = b; m\_c = c;

}

int getA()

{

return m\_a;

}

int getB()

{

return m\_b;

}

int getC()

{

return m\_c;

}

public:

int getV()

{

m\_v = m\_a\*m\_b\*m\_c;

return m\_v;

}

int getS()

{

m\_s = 2\*(m\_a\*m\_b + m\_a\*m\_c + m\_b\*m\_c);

return m\_s;

}

int judgeCube(Cube &v1, Cube &v2) //3k

{

if ( (v1.getA() == v2.getA()) &&

(v1.getB() == v2.getB()) &&

(v1.getC() == v2.getC()) )

{

return 1;

}

else

{

return 0;

}

}

int judgeCube(Cube &v2) //成员函数 函数重载 //4k

{

//好多学员 分不清 这个场景下的m\_a 是属于v1 还是 属于v2 vn

if ( (m\_a == v2.getA()) &&

(m\_b == v2.getB()) &&

(m\_c == v2.getC()) )

{

return 1;

}

else

{

return 0;

}

return 0;

}

private:

int m\_a;

int m\_b;

int m\_c;

int m\_v;

int m\_s;

};

//全局函数 PK 成员函数

//1相等 0不相等

int judgeCube(Cube &v1, Cube &v2)

{

if ( (v1.getA() == v2.getA()) &&

(v1.getB() == v2.getB()) &&

(v1.getC() == v2.getC())

)

{

return 1;

}

else

{

return 0;

}

}

void main()

{

Cube v1, v2;

v1.setABC(1, 2, 3);

cout<<v1.getS()<<endl;

cout<<v1.getV()<<endl;

cout<<"hello..."<<endl;

//

v2.setABC(1,2,4);

//问题抛出 冗余

//函数调用 意味着: 去执行 v1对象的类的成员judgeCube ,

int tag = v1.judgeCube(v2); //判断v1立方体 和 v2立方体是否相等 ,通过类的成员函数

if ( tag == 1)

{

cout<<"相等"<<endl;

}

else

{

cout<<"不相等"<<endl;

}

//判断两个立方体是否相等

if ( v1.judgeCube(v1, v2) == 1)

{

cout<<"相等"<<endl;

}

else

{

cout<<"不相等"<<endl;

}

system("pause");

return ;

}

void main01\_全局函数调用()

{

Cube v1, v2;

v1.setABC(1, 2, 3);

cout<<v1.getS()<<endl;

cout<<v1.getV()<<endl;

cout<<"hello..."<<endl;

//

v2.setABC(1,2,4);

//判断两个立方体是否相等

if (judgeCube(v1, v2) == 1)

{

cout<<"相等"<<endl;

}

else

{

cout<<"不相等"<<endl;

}

system("pause");

return ;

}

点与圆位置关系

#include <iostream>

using namespace std;

class MyPoint

{

public:

void setPoint(int \_x1, int \_y1)

{

x1 = \_x1;

y1 = \_y1;

}

int getX1()

{

return x1;

}

int getY1()

{

return y1;

}

private:

int x1;

int y1;

};

class AdvCircle

{

public:

void setCircle(int \_r, int \_x0, int \_y0)

{

r = \_r; x0 = \_x0; y0 = \_y0;

}

public:

int judge(MyPoint &myp)

{

int dd = (myp.getX1() - x0)\*(myp.getX1() - x0) + (myp.getY1() - y0)\*(myp.getY1() - y0);

if (dd < r\*r)

{

return 0;

}

else if (dd = r\*r)

{

return 2;

}

else

{

return 1;

}

}

private:

int r;

int x0;

int y0;

};

void main()

{

AdvCircle c1;

MyPoint p1;

c1.setCircle(2, 3, 3);

p1.setPoint(3, 5);

//在圆形1 不在圆形0

int tag = c1.judge(p1);

if (tag == 1)

{

cout << "点在圆外" << endl;

}

else if(tag == 0)

{

cout << "点在圆内" << endl;

}

else

{

cout << "点在圆上" << endl;

}

cout << "hollo..." << endl;

system("pause");

return;

}

两点之间的距离

#include "iostream"

#include "math.h"

using namespace std;

class Point

{

public:

void setPoint(double \_x, double \_y)

{

x = \_x;

y = \_y;

}

int getX()

{

return x;

}

int getY()

{

return y;

}

public:

double dis(Point &p)

{

double distant =sqrt( (p.getX() - x)\*(p.getX() - x) + (p.getY() - y)\*(p.getY() - y));

return distant;

}

private:

double x;

double y;

protected:

};

void main()

{

Point p1, p2;

p1.setPoint(3, 3);

p2.setPoint(4, 4);

double d = p1.dis(p2);

cout << "两点之间的距离为 ：" << d << endl;

system("pause");

}

两圆相交判定（bug版本相交判定条件没补齐）

#include "iostream"

#include "math.h"

using namespace std;

class Circle

{

public:

void setCircle(int \_r, int \_x0, int \_y0)

{

r = \_r; x0 = \_x0; y0 = \_y0;

}

public:

int judge(Circle &myp)

{

if (abs(x0 - myp.x0) <= r &&abs(y0 - myp.y0) <= r+myp.r)

{

return 1;

}

else

{

return 0;

}

}

private:

int r;

int x0;

int y0;

};

void main()

{

Circle c1, c2;

c1.setCircle(2, 0, 0);

c2.setCircle(1,5,0);

int tag = c1.judge(c2);

if (tag ==1)

{

cout << "两圆相交" << endl;

}

else

{

cout << "两圆不相交" << endl;

}

system("pause");

}

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <iostream>

using namespace std;

class Test

{

public:

Test() //无参数 构造函数

{

a = 10; //作用完成对属性的初始化工作

p = (char \*)malloc(100);

strcpy(p, "aaaaffff");

cout << "我是构造函数 被执行了" << endl;

}

void print()

{

cout << p << endl;

cout << a << endl;

}

~Test() //析构函数

{

if (p != NULL)

{

free(p);

}

cout << "我是析构函数,被调用了" << endl;

}

protected:

private:

int a;

char \*p;

};

//给对象搭建一个舞台,研究对象的行为

void objplay()

{

//先创建的对象 后释放

Test t1;

t1.print();

printf("分隔符\n");

Test t2;//Test1 ，2先后执行完之后，先调用test2的析构函数

t2.print();

}

void main()

{

objplay();

cout << "hello..." << endl;

system("pause");

return;

}

[c语言](https://baike.baidu.com/item/c%E8%AF%AD%E8%A8%80/105958" \t "https://baike.baidu.com/item/%E9%80%97%E5%8F%B7%E8%A1%A8%E8%BE%BE%E5%BC%8F/_blank)提供一种特殊的[运算符](https://baike.baidu.com/item/%E8%BF%90%E7%AE%97%E7%AC%A6/7752795" \t "https://baike.baidu.com/item/%E9%80%97%E5%8F%B7%E8%A1%A8%E8%BE%BE%E5%BC%8F/_blank)，[逗号](https://baike.baidu.com/item/%E9%80%97%E5%8F%B7/998430" \t "https://baike.baidu.com/item/%E9%80%97%E5%8F%B7%E8%A1%A8%E8%BE%BE%E5%BC%8F/_blank)运算符，优先级别最低，它将两个及其以上的式子联接起来，从左往右逐个计算表达式，整个表达式的值为最后一个表达式的值。如：（3+5,6+8）称为逗号表达式，其求解过程先表达式1，后表达式2，整个表达式值是表达式2的值，如：（3+5，6+8）的值是14；a=(a=3\*5,a\*4)的值是60，其中（a=3\*5,a\*4）的值是60， a的值在逗号表达式里一直是15，最后被逗号表达式赋值为60，a的值最终为60。

构造函数的种类和调用

#include <iostream>

using namespace std;

class Test2

{

public:

Test2() //无参数构造函数

{

m\_a = 0;

m\_b = 0;

cout << "无参数构造函数" << endl;

}

Test2(int a)

{

m\_a = a;

m\_b = 0;

}

Test2(int a, int b) //有参数构造函数 //3种方法

{

m\_a = a;

m\_b = b;

cout << "有参数构造函数" << endl;

}

//赋值构造函数 (copy构造函数) //

Test2(const Test2& obj)

{

cout << "我也是构造函数 " << endl;

}

public:

void printT()

{

cout << "普通成员函数" << endl;

}

private:

int m\_a;

int m\_b;

};

void main21()

{

Test2 t1; //调用无参数构造函数

cout << "hello..." << endl;

system("pause");

return;

}

//调用 调用有参数构造函数 3

//调用不同的构造函数就已经将不同的对象初始化了

void main()

{

//1括号法

Test2 t1(1, 2); //调用参数构造函数 c++编译器自动的调用构造函数

t1.printT();

// 2 =号法

Test2 t2 = (3, 4, 5, 6, 7); // = c+对等号符 功能增强 c++编译器自动的调用构造函数,右边为逗号表达式，等于调用Test2 t2 =4

Test2 t3 = 5;

//3 直接调用构造函数 手动的调用构造函数

Test2 t4 = Test2(1, 2); //匿名对象 (匿名对象的去和留) 抛砖 ....//t4对象的初始化

//只会调用一次构造函数

t1 = t4; //把t4 copy给 t1 //赋值操作

//对象的初始化 和 对象的赋值 是两个不同的概念

cout << "hello..." << endl;

system("pause");

return;

}

构造函数的调用方法是

自动调用（按照规则调用）

也可以显示的初始化类的属性和

#include <iostream>

using namespace std;

class Test3

{

public:

Test3()

{

a = 1;

b = 2;

cout << a << endl;

}

protected:

private:

int a;

int b;

};

void main()

{

//类没有提供构造函数,c++编译器会自动给你提供一个默认的构造函数

//类没有提供构造函数 copy构造构造函数, c++编译器会自动给程序员提供一个 默认的copy构造函数 =

Test3 t1;

int a = 10;

int b = 20;

t1.init(a, b);

Test3 tArray[3];

tArray[0].init(1, 2);

tArray[1].init(1, 2);

tArray[2].init(1, 2);

//

Test3 t21; t21.init(1, 2);

Test3 t22; t22.init(1, 2);

Test3 t23; t23.init(1, 2);

Test3 bArray[3];

//在这种场景之下 显示的初始化方案 显得很蹩脚

//Test3 tArray2[3] = { t21, t22, t23 };

//在这种场景之下,满足不了,编程需要

//Test3 tArray3[1999] = {t21, t22, t23};

cout << "hello..." << endl;

system("pause");

return;

}

Copy函数的调用

#include <iostream>

using namespace std;

class Test4

{

public:

Test4() //无参数构造函数

{

m\_a = 0;

m\_b = 0;

cout<<"无参数构造函数"<<endl;

}

Test4(int a)

{

m\_a = a;

m\_b = 0;

}

Test4(int a, int b) //有参数构造函数 //3种方法

{

m\_a = a;

m\_b = b;

cout<<"有参数构造函数"<<endl;

}

//赋值构造函数 (copy构造函数) //

Test4(const Test4& obj )

{

cout<<"我也是构造函数 " <<endl;

m\_b = obj.m\_b + 100;

m\_a = obj.m\_a + 100;

}

public:

void printT()

{

cout<<"普通成员函数"<<endl;

cout<<"m\_a"<<m\_a<<" m\_a"<<m\_b<<endl;

}

private:

int m\_a;

int m\_b;

};

//1 赋值构造函数 用1个对象去初始化另外一个对象

void main41()

{

Test4 t1(1, 2);

Test4 t0(1, 2);

//赋值=操作 会不会调用构造函数（不会调用构造函数）

//operator=()//抛砖

t0 = t1; //用t1 给 t0赋值 到操作 和 初始化是两个不同的概念

//第1种调用方法

Test4 t2 = t1; //用t1来初始化 t2

t2.printT();

cout<<"hello..."<<endl;

system("pause");

return ;

}

//第二种调用时机

void main44()

{

Test4 t1(1, 2);

Test4 t0(1, 2);

Test4 t2(t1); //用t1对象 初始化 t2对象

t2.printT();

cout<<"hello..."<<endl;

system("pause");

return ;

}

第三种调用

#include <iostream>

using namespace std;

class Location

{

public:

Location()

{

X = 1; Y = 2; cout << "111Constructor Object.\n";

}

Location(int xx , int yy = 0)

{

X = xx; Y = yy; cout << "Constructor Object.\n";

}

//copy构造函数 完成对象的初始化

Location(const Location & obj) //copy构造函数

{

X = obj.X; Y = obj.Y;

}

~Location()

{

cout << X << "," << Y << " Object destroyed." << endl;

}

int GetX() { return X; } int GetY() { return Y; }

private: int X, Y;

};

//业务函数 形参是一个元素

void f(Location p)//用实参bcopy构造函数p，完成了对p的初始化

{

cout << p.GetX() << endl;

}//p的内存最先释放，析构函数最先运行

void playobj()

{

Location c;

Location a(1, 2);

Location b = a;

cout << "b对象已经初始化完毕" << endl;

f(b); //b实参取初始化形参p,会调用copy构造函数

}//p先释放析构函数，再到b释放析构函数，最后到a

void main()

{

playobj();

cout << "hello..." << endl;

system("pause");

return;

}

匿名对象调用场景

#include <iostream>

using namespace std;

class Location

{

public:

Location( int xx = 0 , int yy = 0 )

{

X = xx ; Y = yy ; cout << "Constructor Object.\n" ;

}

//copy构造函数 完成对象的初始化

Location(const Location & obj) //copy构造函数

{

X = obj.X; Y = obj.Y;

}

~Location()

{

cout << X << "," << Y << " Object destroyed." << endl ;

}

int GetX () { return X ; } int GetY () { return Y ; }

private : int X , Y ;

} ;

//g函数 返回一个元素

//结论1 : 函数的返回值是一个元素 (复杂类型的), 返回的是一个新的匿名对象(所以会调用匿名对象类的copy构造函数)

//

//结论2: 有关 匿名对象的去和留

//如果用匿名对象 初始化 另外一个同类型的对象, 匿名对象 转成有名对象

//如果用匿名对象 赋值给 另外一个同类型的对象, 匿名对象 被析构

//

//你这么写代码,设计编译器的大牛们:

//我就给你返回一个新对象(没有名字 匿名对象)

Location

g()

{

Location A(1, 2);

return A;//用a对象创建了一个匿名对象，所以执行了copy构造函数

}

//

void objplay2()

{

g();

}

//

void objplay3()

{

//用匿名对象初始化m 此时c++编译器 直接把匿名对转成m;(扶正) 从匿名转成有名字了m

Location m = g();

printf("匿名对象,被扶正,不会析构掉\n");

cout<<m.GetX()<<endl;;

}

void objplay4()

{

//用匿名对象 赋值给 m2后, 匿名对象被析构

Location m2(1, 2);

m2 = g();//此时类似于赋值，g()被析构掉

printf("因为用匿名对象=给m2, 匿名对象,被析构\n");

cout<<m2.GetX()<<endl;;

}

void main()

{

//objplay2();

//objplay3();

objplay4();

cout<<"hello..."<<endl;

system("pause");

return ;

}

函数构造的初始化列表

#include <iostream>

using namespace std;

class A

{

public:

A(int \_a)

{

a = \_a;

cout << "构造函数" << "a" << a << endl;

}

~A()

{

cout << "析构函数" << "a" << a << endl;

}

protected:

private:

int a;

};

//1 构造函数的初始化列表 解决: 在B类中 组合了一个 A类对象 (A类设计了构造函数)

//根据构造函数的调用规则 设计A的构造函数, 必须要用;没有机会初始化A

//新的语法 Constructor::Contructor() : m1(v1), m2(v1,v2), m3(v3)

class B

{

public:

B(int \_b1, int \_b2) : a1(1), a2(2), c(0)

{

}

B(int \_b1, int \_b2, int m, int n) : a1(m), a2(n), c(0)

{

b1 = \_b1;

b2 = \_b2;

cout <<"B的构造函数"<<endl;

}

~B()

{

cout<<"B的析构函数" <<endl;

}

protected:

private:

int b1;

int b2;

A a2;

A a1;

const int c;

};

//2 先执行 被组合对象的构造函数 (A a1,A A2)

//如果组合对象有多个,按照定义顺序, 而不是按照初始化列表的顺序

//析构函数 : 和构造函数的调用顺序相反

//3 被组合对象的构造顺序 与定义顺序有关系 ,与初始化列表的顺序没有关系.

//4 初始化列表 用来 给const 属性赋值 ，如果初始化列表中有const则在构造函数中必须对其初始化

void obj10play()

{

//A a1(10);

//B ojbB(1, 2);

//1参数传递

B ojbB2(1, 2,3, 4);

//2 调用顺序

return ;

}

void main100()

{

obj10play();

system("pause");

}

构造和析构函数强化

//对象做函数参数

//1 研究拷贝构造

//2 研究构造函数，析构函数的调用顺序

//总结 构造和析构的调用顺序

#include "iostream"

using namespace std;

class ABCD

{

public:

ABCD(int a, int b, int c)

{

this->a = a;

this->b = b;

this->c = c;

printf("ABCD() construct, a:%d,b:%d,c:%d \n", this->a, this->b, this->c);

}

~ABCD()

{

printf("~ABCD() construct,a:%d,b:%d,c:%d \n", this->a, this->b, this->c);

}

int getA()

{

return this->a;

}

protected:

private:

int a;

int b;

int c;

};

class MyE

{

public:

MyE():abcd1(1,2,3),abcd2(4,5,6),m(100)

{

cout<<"MyD()"<<endl;

}

~MyE()

{

cout<<"~MyD()"<<endl;

}

MyE(const MyE & obj):abcd1(7,8,9),abcd2(10,11,12),m(100)

{

printf("MyD(const MyD & obj)\n");

}

protected:

//private:

public:

ABCD abcd1; //c++编译器不知道如何构造abc1

ABCD abcd2;

const int m;

};

int doThing(MyE mye1)//用实参初始化形参相当于Mye（myE）进入copy构造函数

{

printf("doThing() mye1.abc1.a:%d \n", mye1.abcd1.getA());

return 0;

}

int run2()

{

MyE myE;

doThing(myE);

return 0;

}

//

int run3()

{

printf("run3 start..\n");

//ABCD(400, 500, 600); //匿名对象。临时对象的生命周期 ，只存在于这一句话，因为没有对象去接

ABCD abcd = ABCD(100, 200, 300);

//若直接调用构造函数呢？

//想调用构造函数对abc对象进行再复制，可以吗？

//在构造函数里面调用另外一个构造函数，会有什么结果？

printf("run3 end\n");

return 0;

}

int main()

{

//run2();

run3();

system("pause");

return 0;

} .

#include "iostream"

using namespace std;

//构造中调用构造是危险的行为

class MyTest

{

public:

MyTest(int a, int b, int c)

{

this->a = a;

this->b = b;

this->c = c;

}

MyTest(int a, int b)

{

this->a = a;

this->b = b;

MyTest(a, b, 100); //产生新的匿名对象，没有函数去接，直接匿名对象的析构函数

}

~MyTest()

{

printf("MyTest~:%d, %d, %d\n", a, b, c);

}

protected:

private:

int a;

int b;

int c;

public:

int getC() const { return c; }

//void setc(int val)

//{

// c = val;

//}

//此段可省

};

int main()

{

MyTest t1(1, 2);

printf("c:%d", t1.getC()); //请问c的值是？

system("pause");

return 0;

}

#include "iostream"

using namespace std;

//构造中调用构造是危险的行为

class MyTest

{

public:

MyTest(int a, int b, int c)

{

this->a = a;

this->b = b;

this->c = c;

}

MyTest(int a, int b)

{

this->a = a;

this->b = b;

MyTest(a, b, 100); //产生新的匿名对象，没有函数去接，直接匿名对象的析构函数

}

~MyTest()

{

printf("MyTest~:%d, %d, %d\n", a, b, c);

}

protected:

private:

int a;

int b;

int c;

public:

int getC() const { return c; }

//void setc(int val)

//{

// c = val;

//}

//此段可省

};

#include <iostream>

using namespace std;

// 1

// malloc free c语言的函数

// new delete 操作符 c++的语法

//2 new 基础类型变量 分配数组变量 分配类对象

//3

////分配基础类型

void main1301()

{

//

int \*p = (int \*)malloc(sizeof(int));

\*p = 10;

free(p);

int \*p2 = new int; //分配基础类型

\*p2 = 20;

free(p2);

//

int \*p3 = new int(30);

printf("\*p3:%d \n", \*p3);

delete p3;

cout << "hello..." << endl;

system("pause");

return;

}

//分配数组变量

void main1302()

{

//c语言分配数组

int \*p = (int \*)malloc(sizeof(int) \* 10); //int array[10];

p[0] = 1;

free(p);

//c++分配数组

int \*pArray = new int[10];

pArray[1] = 2;

delete[] pArray; //数组不要把[] 忘记

char \*pArray2 = new char[25]; //char buf[25]

delete[] pArray2;

cout << "hello..." << endl;

system("pause");

return;

}

class Test

{

public:

Test(int \_a)

{

a = \_a;

cout << "构造函数执行" << endl;

}

~Test()

{

cout << "析构函数执行" << endl;

}

protected:

private:

int a;

};

//分配对象new delete

//相同 和 不同的地方 new能执行类型构造函数 delete操作符 能执行类的析构函数

void main()

{

//c

Test \*pT1 = (Test \*)malloc(sizeof(Test));

\*pT1= Test(10);

free(pT1);

//c++

Test \*pT2 = new Test(10);

delete pT2;

cout << "hello..." << endl;

system("pause");

}

Static关键字

#include <iostream>

using namespace std;

class BB

{

public:

void printC()

{

cout<<"c:"<<c<<endl;

}

void AddC()

{

c = c + 1;

}

static void getC() //静态成员函数

{

cout<<"c:"<<c<<endl;

//请在静态成员函数中,能调用 普通成员属性 或者 普通成员函数吗?

cout<<"a:"<<a<<endl; //error C2597: 对非静态成员“BB::a”的非法引用

}

protected:

private:

int a;

int b;

static int c; //静态成员变量

};

//静态函数中 不能使用 普通成员变量 普通成员函数 ..

int BB::c = 10;

void main()

{

BB b1, b2, b3;

b1.printC(); //10

b2.AddC(); //11

b3.printC(); //11

//静态成员函数的调用方法

b3.getC(); //用对象.

BB::getC();//类::

cout<<"hello..."<<endl;

system("pause");

return ;

}

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <iostream>

using namespace std;

//

class Name

{

public:

Name(const char \*myp)

{

m\_len = strlen(myp);

m\_p = (char \*)malloc(m\_len + 1); //+1是为了保存/0

strcpy(m\_p, myp);

}

//Name obj2 = obj1;

//解决方案: 手工的编写拷贝构造函数 使用深copy

Name(const Name& obj1)//深拷贝拷贝的是类obj1.m\_p所指向的内容，而浅拷贝拷贝的是单纯的地址

{

m\_len = obj1.m\_len;

m\_p = (char \*)malloc(m\_len + 1);

strcpy(m\_p, obj1.m\_p);

}

~Name()

{

if (m\_p != NULL)

{

free(m\_p);

m\_p = NULL;

m\_len = 0;

}

}

protected:

private:

char \*m\_p;

int m\_len;

};

//对象析构的时候 出现coredump

void objplaymain()

{

Name obj1("abcdefg");

//Name obj2 = obj1; //C++编译器提供的 默认的copy构造函数 浅拷贝

Name obj3(obj1); // C++编译器提供的 等号操作 也属 浅拷贝

//Name obj3=obj1; 语风较差

}

void main()

{

objplaymain();

cout << "hello..." << endl;

system("pause");

return;

}

#include <iostream>

using namespace std;

class Test

{

public:

int a;

int b;

public:

~Test()

{

cout<<"a:"<<a<<" b: "<<b;

cout<<"析构函数自动被调用"<<endl;

}

public:

void printT()

{

cout<<"a:"<<a<<" b: "<<b<<endl;

}

public:

//t3 = t1.TestAdd(t2);

Test TestAdd(Test &t2)

{

Test tmp(this->a + t2.a, this->b + t2.b);

return tmp;

}

//t1.TestAdd2(t2);

//返回一个引用 相当于返回自身

//返回t1这个元素 this就是&t1

Test& TestAdd2(Test &t2)

{

this->a = this->a + t2.a;

this->b = this->b + t2.b;

return \*this; //把 \*(&t1) 又回到了 t1元素//This就是&t1,加上\*相当于取值，再加上&引用

}

public:

Test(int a=0, int b=0)

{

this->a = a;

this->b = b;

}

};

//把成员函数 转成 全局函数 多了一个参数

void printT(Test \*pT)

{

cout<<"a:"<<pT->a<<" b: "<<pT->b<<endl;

}

//全局函数的方法

//全局函数 转成 成员函数 少了一个参数

Test TestAdd(Test &t1, Test &t2)

{

Test tmp;

return tmp;

}

void main1()

{

Test t1(1, 2);

Test t2(3, 4);

//t1 = t1 + t2;

t1.TestAdd2(t2);

t1.printT();

}

void main()

{

Test t1(1, 2);

Test t2(3, 4);

Test t3 ;

//全局函数方法

t3 = TestAdd(t1, t2);

//成员函数方法

{

//先把测试案例写出来

Test t4 = t1.TestAdd(t2); //匿名对象直接转化成t4

t4.printT();

Test t5;

t5 = t1.TestAdd(t2); //匿名对象 复制 给t5

t5.printT();

}

cout<<"hello..."<<endl;

system("pause");

return ;

}

友元函数友元类

#include <iostream>

using namespace std;

class A

{

public:

friend class B;//B类 是 A的好朋友 ，在B中可以访问A类的私有成员 私有函数

//1 声明的位置 和 public private没有关系

friend void modifyA(A \*pA, int \_a); //2 函数modifyA 是 类A的好朋友

A(int a=0, int b=0)

{

this->a = a;

this->b = b;

}

int getA()

{

return this->a;

}

private:

int a;

int b;

};

//

void modifyA(A \*pA, int \_a)

{

//pA->a = 100;

pA->a = \_a;

}

//

class B

{

public:

void Set(int a)

{

Aobject.a = a;

}

void printB()

{

cout<<Aobject.a <<endl;

}

private:

A Aobject;

};

//为什么设计友元类函数

// 1.java--->1.class（字节码） ==》反射机制分析1.class 找到类对象。直接修改类的私有属性。。。

//反射机制 成为一种标准。。。。jdk ...sun 做成标准 。。。jdk 的 api函数中有体现

//AOP

//2 1.cpp===>汇编

// 预编译 编译 连接 生成 。。gcc -E //gcc -s -

//gcc -o 1.exe 1.c

// 汇编往会找。。。。很难。。。。

//3 开了一个后门 。。。friend

/\*

gcc -E hello.c -o hello.i（预处理）

gcc -S hello.i -o hello.s（编译）

gcc -c hello.s -o hello.o（汇编）

gcc hello.o -o hello（链接）

以上四个步骤，可合成一个步骤

gcc hello.c -o hello（直接编译链接成可执行目标文件）

gcc -c hello.c或gcc -c hello.c -o hello.o（编译生成可重定位目标文件）

\*/

void main()

{

B b1;

b1.Set(300);

b1.printB();

system("pause");

}

void main2101()

{

A a1(1, 2);

cout<< a1.getA()<<endl;

modifyA(&a1, 300);

cout<< a1.getA()<<endl;

cout<<"hello..."<<endl;

system("pause");

return ;

}

站住ALt可以选中一竖线上的点，进行下一步操作

return strcmp(m\_p, s.m\_p);//若想等则返回0，str1<<str2则返回-1；str1<<str2则返回正数

写代码时先搭建框架，再写测试案例再填空

数组类，字符串类重载-基础-05day

#include "stdafx.h"头文件名得在新建项目时使用

添加相对路径时得添加到最里面一层如

