# 1. C++基础概念

C++语言是在C语言基础之上为支持面向对象而设计的这么一套程序语言，而面向对象在C++中的体现就是类的机制，所以C++也称作是“带类的C语言的”。

## 1.1 类

C++的类类似于C语言的结构体，但是类除了可以包含很多个基本变量类型以外，还可以包含很多函数。

class stu

{

int num;

char sex;

int math\_score;

int en\_score;

int c\_score;

int total\_score()

{

return math\_score + en\_score + c\_score;

};

};

class stu A;

注：C++类中的成员变量叫做属性，类里的函数叫做方法。即类拥有属性与方法两部分；

## 1.2 c++的小程序

为了解决多个模块间命名冲突的问题，C++把相同的名字都放到不同的空间里，来防止名字的冲突。

所谓namespace，是指标识符的各种可见范围。C++标准程序库中的所有标识符都被定义于一个名为std的namespace中。using namespace std——表示使用名为std的命名空间；#include<iostream>是c++的标准库，不带后缀的，带后缀和不带后缀是两个不同的文件，且使用时必须使用namespace std；这样才能正确使用cout。

Eg：

#include<iostream>

using namespace std;

int main()

{

cout<<"Nice to meet you!"<<endl;

return 0;

}

/\*std :: cout 相当于c语言中的printf；std :: cin 相当于c语言中的scanf\*/

/\* std :: endl 起输出换行的效果, 相当于c语言中的\n\*/

/\* cout<< 将输出数据插入到cout中；cin>> cin到输入数据中；\*/

注：cin >> num >> sex >> score1 >> score2 >> score3; 输入时以空格为分界

# 2. 函数

## 2.1 带默认形参的函数

C++中，允许函数的形参带默认值，如果调用该函数时，没有传实参，则该函数按默认形参运行，如果调用时有实参，则按实参运行。

Eg：

int add(int a=3,int b=5)

{

return a+b;

}

int main()

{

cout<<add(10,20)<<endl;//将10和20分别给a和b

cout<<add(30)<<endl;//将30给a，b为默认的5

cout<<add()<<endl;//使用a、b的默认值3和5

return 0;

}

另外，如果函数提前声明，可以在函数声明中给形参默认值，在函数定义时就不用给默认值了。

注：由于参数是从右至左入栈，所以默认参数必须是函数参数表中最右边的参数。

Eg：

int add(int a,int b=5)，在调用时add(3),表示a = 3, b默认为5；但是如果int add(int a = 3,int b)，在调用时add(2),表示a = 2, b没有赋值就错了。

## 2.2 函数重载

函数重载即两个或两个以上的函数，函数名相同，但形参类型或形参个数不同，编译器会根据调用时传进去的参数类型和个数，自动选择合适的函数进行运行。

Eg：

int add(int a,int b)

{

cout<<"(int ,int)\t";

return a+b;

}

double add(double a,double b)

{

cout<<"(doble ,double)\t";

return a+b;

}

int main()

{

cout<<add(2,3)<<endl;

cout<<add(2.9,15.3)<<endl;

}

## 2.3 函数模板

函数重载中，会出现同一个函数名，但参数类型不同，为了精简代码，出现了函数模板，即创建一个通用的函数，可以支持多种参数类型。

用关键字template<class 类型名1，class 类型名2……>

注：template<class 类型名1，class 类型名2……>相当于类型声明（不需要分号），其中class也可以换成typename，在template语句后必须紧跟函数模板，中间不可以加其他语句，否则会报错。

Eg：

template < class T1, class T2>

//template < typename T1, typename T2>

T1 add(T1 a, T2 b)

{

cout << sizeof(a) << " " << sizeof(b) << "\t";

return a + b;

}

int main()

{

cout << add(12.5, 112) << endl;

cout << add(5, 10) << endl;

cout << add(33.2, 59.5) << endl;

cout << add('A',5) << endl;

}

## 2.4 内联函数

函数在调用的时候才会为其分配空间，调用结束清理释放空间，可以看出，每次调用都会带来时间和空间上的花销，如果一个函数比较短，又频繁被调用，这样反而降低了开发效率，所以C++提供了内联函数，在编译的时候直接将内联函数复制插入到函数调用处，像普通代码执行顺序一样。

用法：在函数前加关键字 inline 声明即可。

Eg：

inline int add(int a, int b)

{

return a + b;

}

# 3. 类和对象

## 3.1 类及其类型

（1）类就是数据类型，对象就是类型的变量，（int a； 类相当于int，a相当于对象）。

（2）类的关键字为class ，在类中既可以定义的属性（即成员）也可以定义方法（即函数），定义方法时，可以在类中仅声明（必须是有返回值的函数），在外面再定义。

Eg：

class company

{

public:

int num;

char name[50];

double salary;

int print();

};

int company :: print() //返回值在前面，然后是 类 :: 方法；

{

cout << name << ":" << salary << endl;

}

（3）类内定义成员时得加控制成员访问的权限：

Public声明的成员:可以被该类的任何对象访问，是完全公开的数据；

Private声明的成员：仅能被该类里的成员访问，外界不能访问，是最封闭的一种权限；

Protected声明的成员：比Private公开点，除了可以被类内的成员访问外，还可以被它的子类（派生类）访问。

对于想保护但又想控制的私有变量，通常将其声明为private，然后同时定义一个public类型来专门给private的成员赋值。

Eg：

class company

{

private:

int num;

char name[50];

double salary;

public:

int func(char \* str, int s);

int print();

};

int company::func(char \* str, int s)

{

strcpy(name, str);

salary = s;

return 0;

}

注：如果没加访问权限，默认非public，不能通过：对象.变量的方式直接访问。

## 3.2 对象

（1）对象的指针

对象也是一片连续的内存空间，也可以用指针来存储对象的地址。

Eg：

company A;

company \*p;

p = &A;

p->func("lizhi", 10000);

p->print();

在传递参数时，建议用指针传递，减少内存开销，提高效率。

（2）对象的引用

引用是C++的新类型，对象引用相当于给对象A起了个别名叫B，本质上是把这个类对象的地址赋给了这个引用对象，两者指向同一块内存空间。

引用用“&”关键字来定义。

Eg：

company A;

company &B = A; //定义一个引用对象，将A赋给B作为初始化

B.func("lizhi", 10000);

B.print();

注：

两者必须是同类型才可以引用；

除非做函数的返回值或形参时，其余定义引用类型的同时就要初始化；

引用类型并非是新建一个对象，所以不会调用构造函数；

类对象是别名，所以使用方法和类对象一样，用别名 . 成员进行访问；

引用对象的本质是存地址，因此无论传参定义都不会太多内存开销，有指针的优势，同时使用起来和使用类对象一样，因此更方便、直观。

另外：在C++中void swap(int &a, int &b)；参数中的&表示引用型参数；一般参数传递时，形参是实参的副本，也就是说，形参虽然和实参一样，但是在内存中的位置不同；而引用型参数，在参数传递的时候，并不需要复制参数，形参和实参在内存中同一位置，因此修改形参，实参也变化。

Eg:

void swap(int &a, int &b) //引用型参数

{

int tempt;

tempt = a;

a = b; //改变形参值，实参也会变化

b = tempt;

}

## 3.2 构造函数

构造函数是一个函数名和类名相同，且没有返回值的函数。只要定义一个类对象，系统会自动调用构造函数来初始化对象（也就是说，构造函数就是为了初始化类中的成员）。如若用户没有自定义构造函数，系统会默认一个隐藏构造函数，是空的。如果用户自定义了构造函数，系统就不会再生成构造函数了，且用户可以定义多个构造函数（即构造函数可以重载），系统会根据参数来进行最优匹配。

另外，如果构造函数带默认参数了，在创建对象时，初始化也要带参数，防止构造函数重载，系统找不到匹配的。

Camera camera (10.1, 1.0,0.2)；

注：只有当类有自建的构造函数时，在创建对象时，就应该给自建构造函数传参，没有自建构造函数则不需要。

构造函数后面加冒号“：”其中一种含义表示：对类成员进行初始化.

Eg：

float speed = 0.05f;

float sensitivity = 0.05f;

class camera

{

public:

float UP;

float Position;

float MovementSpeed;

float Mousesensitivity;

camera(float pos, float up) :MovementSpeed(speed), Mousesensitivity(sensitivity)

{

UP = up;

Position = pos;

cout << "Position:" << Position << "\t" << "UP:" << UP << "\t" << "MovementSpeed:" << MovementSpeed << endl;

}

};

void main()

{

camera C(10.1, 1.0); //输出：Position:10.1 UP:1.0 MovementSpeed：0.05f

}

其中构造函数也可以写成：

camera(float pos, float up)

{

UP = up;

Position = pos;

MovementSpeed = speed;

Mousesensitivity = sensitivity;

cout << "Position:" << Position << "\t" << "UP:" << UP << "\t" << "MovementSpeed:" << MovementSpeed << endl;

}

};

## 3.3 析构函数

析构函数是一个函数名和类名相同但带个“~”，且没有返回值的函数，没有参数。在销毁对象时，系统也会自动调用析构函数，其目的主要是在删除对象前，关闭文件，释放内存。如果用户不自定义析构函数，系统也会自动生成一个空的析构函数，如果用户自定义了析构函数，则对象在销毁时自动调用。

另外，虽然析构函数和构造函数都是公开类型，但析构函数不能重载，只能有一个，因为它没有参数，析构函数可以是虚函数。

Eg:

class camera

{

public:

float pos;

float up;

camera(float Position, float camera\_UP, float Move\_Speed) /\*构造函数\*/

{

cout << "it is good time!" << endl;

pos = Position;

up = camera\_UP;

speed = Move\_Speed;

cout << "speed:" << speed << endl;

}

~camera() /\*析构函数\*/

{

cout << "the time is over!" << endl;

}

private:

float speed;

};

int main()

{

camera A(11.1, 1.0, 2.5);

Return 0;

}

输出：

>>it is good time!

speed:2.5

the time is over!

注：局部变量存储在栈区，也遵循“先进后出”，所以如果2个对象A和B，先掉A的构造函数，B的构造函数，然后调用B的析构函数，A的析构函数。

## 3.4 拷贝构造函数

拷贝构造函数的函数名与类名相同，没有返回值，但其参数为本类对象的引用类型。

形式：classname ( const classname &obj) //obj为本类对象的引用

一般系统会自动创建一个拷贝构造函数，但是如果类中存在指针成员，并有动态内存分配，则需自行创建拷贝构造函数。

3种情况下会调用拷贝构造函数：

1. 使用同一类的对象来初始化新的对象：

当创建一个新对象时，用同类型的对象去初始化该对象，则会调用拷贝构造函数。（拷贝构造函数是为了初始化对象）

（2）将对象作为参数传入函数；

（3）复制对象，并从函数返回这个对象。

Eg:

camera :: camera(float Position, float camera\_UP, float Move\_Speed) /\*构造函数\*/

{

cout << "构造函数" << endl;

pos = Position;

up = camera\_UP;

speed = Move\_Speed;

cout << "speed:" << speed << endl;

}

camera::camera(const camera& obj) /\*拷贝构造函数\*/

{

cout << "拷贝构造函数" << endl;

pos = obj.pos;

up = obj.up;

speed = obj.speed;

cout << "speed:" << speed << endl;

}

Int main（）

{

camera A(11.1, 1.0, 2.5); //调用构造函数初始化

camera B (A); //调用拷贝构造函数初始化,camera B = A两者一样

}

注：this是C++的一个关键字。this 只能用在类的内部，通过 this 可以访问类的所有成员，包括 private、protected、public 属性的。（只要定义一个类，系统就会预定义一个名为this且指向当前对象的指针）

## 3.5 浅拷贝和深拷贝

如果不自行创建拷贝函数，系统会默认一个拷贝函数，但是默认拷贝函数仅对数值进行拷贝，当类中存在指针成员且在构造函数中对其分配内存，如果用默认拷贝函数，仅将数值进行赋值，也就是说，对象A和B指向同一块内存空间，但是在释放内存时，会分别调用析构函数，即2次delete同一块内存，会出错。

深拷贝即当类中存在指针变量且其在构造函数中分配空间，则必须用户自定义拷贝函数，在拷贝函数中也分配内存，也就是说，深拷贝既可以进行数据拷贝，也可以分配内存空间。

Eg：

class team

{

public:

char \*name;

int num;

team(char\* str, int number)

{

name = new char[strlen(str) + 1];

strcpy(name,str);

num = number;

cout << "构造函数" << endl;

}

~team()

{

cout << "析构函数" << endl;

delete []name;

}

}

team A("lizhi", 1);

team B = team(A);

## 3.6 友元函数

由于private私有成员只能该类中的成员才能访问，类外是不能访问的，类外函数或其他类的成员函数如果想要访问该类的私有成员，则这个函数必须是这个类的友元函数（友元不仅可以是一个函数，也可以是一个类）。

对于友元函数，只需在类内对这个函数进行声明，并加上关键字：friend，就赋予了这个函数访问该类内私有成员的权利。注：友元函数不是该类的成员，不能用this，也不可以被继承。

Eg：

class point

{

private:

double x;

double y;

public:

point(double xx, double yy)

{

this->x = xx;

this->y = yy;

}

friend double destance(point& xy);

};

double destance(point &xy)

{

double result,xx,yy;

xx = xy.x;

yy = xy.y;

result = sqrt(xx \* xx + yy \* yy);

return result;

}

int main()

{

point A(3, 4);

double z = destance(A);

cout << "距离:" << z << endl;

return 0;

}

以上可以看出，double destance(point &xy)原本是类外的一般函数，但是为了访问类point的私有成员，我们在class point中，将该函数声明为友元函数friend double destance(point& xy);这样，它就可以访问point的私有成员了。

友元类和友元函数用法类似，都是在想要访问私有成员的类内加上关键字：friend进行声明。将class Tool声明为class point 的友元类，则class Tool就可以访问class point中的私有成员了。

友元机制的优缺点：

优点：更方便快捷的访问类内私有成员；

缺点：打破了C++的封装思想。

## 3.7 C++的常量

（1）常量数据成员

类中的常量数据成员用const修饰，可以是const int a 或 int const a，常量数据成员要在构造函数中的初始化列表中初始化（clock(int x, int y): a(x) ,b(y)，冒号前面的括号是参数表冒号后面的是初始化列表），但静态static常量数据成员，要在类外初始化。

注：类中静态static变量属于类，而不是对象，只能初始化一次，所以static变量在类中声明，在类外初始化，初始化时可以不带static。另外，静态常量数据由于它是常量，所以就算在类内其值也不会改变，所以静态常量数据也可以在类内初始化。

Eg:

class clock

{

public:

int const a;

const int b; //和上面一样

static const int c ; //静态static常量数据成员，在类内声明，类外初始化

clock(int x, int y): a(x) ,b(y) //常量数据成员在构造函数的初始化列表中初始化

{

cout << "构造函数" << endl;

}

int output()

{

cout << "a = " << a << " "<< "b = " << b << " " << "c = " << c << endl;

return 0;

}

};

const int clock::c = 80; //类外初始化可以不带static

int main()

{

clock A(100,200);

A.output();

clock B(300, 400);

B.output();

return 0;

}

（2）常对象

const clock A或 clock const A，常对象一经初始化就不能改变了，所以常对象不能调用普通成员或普通成员函数（防止成员变量被改变），只能调用常量成员函数。

Eg：

Class clock

{

Public：

int const a;

const int b; //和上面一样

int d;

clock(int x, int y): a(x) ,b(y)

{

d = 33;

cout << "构造函数" << endl;

}

int output()

{

cout << "a = " << a << " "<< "b = " << b << " " << "c = " << c << endl;

return 0;

}

int put\_d() const //常量成员函数

{

cout << "d = " << d << endl;

return 0;

}

}；

int main()

{

const clock C(500, 50);

//C.output(); //error,不能调用非常量成员函数

C.put\_d(); //调用常量成员函数

return 0;

}

（3）常成员函数

在类中的格式：返回值 函数名（参数表） const；

常成员函数声明和定义时都要带const。常成员函数只能调用常成员函数，不能调用非常成员函数，访问但不可以更改非常成员变量。

# 4. 继承和派生

## 4.1 继承和派生的基本概念

有2个类，一个类拥有另一个类的全部属性，就说明这个类继承了原有类，原有类产生新类的过程叫做派生，我们把原有类叫做父类或基类，由基类派生出来的类叫做派生类或子类。

（1）继承和派生的好处：

1）体现了面向对象的编程思想，更好的表达各类型的关系；

2）派生类除了可以继承基类的所有属性外，还可以添加自己的特殊属性；

3）派生类继承到基类的成员是自动、隐藏拥有的，不需要重新定义，这就节省了大量代码，体现了代码复用的软件工程思想。

（2）形式

class 派生类名 : public 基类名 （通过“：”继承，public继承方式）

Eg:

class clock

{

private:

int ah;

int bh;

int ch;

public:

int settime(int a, int b, int c)

{

this->ah = a;

this->bh = b;

this->ch = c;

return 0;

}

int outputclock()

{

cout << "ah:" << ah << " bh:" << bh << " ch:" << ch << endl;

return 0;

}

};

class alartclock :public clock

{

private:

int dh;

int eh;

public:

int setalartclock(int d, int e)

{

this->dh = d;

this->eh = e;

return 0;

}

int outputalartclock()

{

cout << "dh:" << dh << " eh:" << eh << endl;

return 0;

}

};

int main()

{

alartclock A;

A.settime(1, 2, 3);

A.setalartclock(100, 200);

A.outputclock();

A.outputalartclock();

return 0;

}

如上，派生类对象A可以调用基类的settime(),outputclock()方法。

## 4.2 三种继承方式

继承方式有：公有继承public，私有继承private，保护继承protected.

1. 公有继承

基类中是公有类型的，在派生类中仍为公有类型，无论派生类内成员函数还是派生对象都可以访问；

基类中是私有类型的，无论派生类内成员函数还是派生对象都不可以访问；

基类中是保护类型的，派生类内成员函数可以访问，但是派生对象不可以访问。

1. 私有继承

基类中是公有类型和保护类型的，在派生类中都变为私有类型，只有派生类的成员函数可以访问，不能在类外访问；

基类中是私有类型的，在派生类内或派生类外都不可以访问。

1. 保护继承

基类中是公有类型和保护类型的，在派生类中都变为保护类型，派生类的成员函数可以访问，但派生类对象不可以访问；

基类中是私有类型的，在派生类内或派生类外都不可以访问。

结论：派生类从基类中继承的成员访问权限是其在基类中的访问权限和继承类型的权限两者中最低的。



## 4.3 派生类的构造函数和析构函数

（1）构造函数

派生类在继承基类的所有成员时，并不会继承他的构造函数和析构函数；如果基类没有构造函数或者构造函数没有参数，在创建派生类对象时，会先调用基类的构造函数，然后调用派生类的构造函数，这是因为隐式调用，不用写出来，系统会自动动调用。

但如果基类的构造函数是带参数的，我们则需在创建派生类的构造函数时，显式的、明确写出来，并指定参数的调用基类的构造函数，形式如下：

派生类构造函数名（总形参表）：基类构造函数名（基类构造函数实参表）

Eg:

alartclock(int z, int e,int k) : clock(z, e) //基类clock只要2个参数，派生类alartclock和基类clock共3个参数

注：一旦基类构造函数带有参数，派生类中则必须有显式传参的派生类构造函数，来实现基类中参数的传递，来实现初始化工作。

（2）析构函数

同构造函数一样，派生类在继承基类的所有成员时，也不会继承他的析构函数。但是析构函数在调用时，先调用派生类的构造函数，后调用基类的构造函数，这是因为派生类对象是一个局部变量，存在栈中，遵守“先进后出”。

构造函数调用顺序：基类->派生类；

析构函数调用顺序：派生类->基类。

## 4.4 虚基类

如果2个派生类father1和father2同时继承了基类grandpa，又有一个派生类grandson同时继承了派生类father1和father2，那么在派生类grandson中就包含了2个基类grandpa，这就出现了重复性，对定义对象调用基类grandpa的成员时，就会出现错误。

为了避免以上错误，出现了“虚基类”，即如果两个以上的派生类继承同一个基类时，在继承方式public前加“virtual”，这样派生类和基类就只会维护一个基类对象，避免多次拷贝，出现歧义。

Eg：

class grandpa

{

public:

int key;

grandpa()

{

cout << "grandpa构造函数"<< endl;

}

~grandpa()

{

cout << "grandpa析构函数"<< endl;

}

};

class father1 : virtual public grandpa

{

public:

father1()

{

cout << "father1构造函数"<< endl;

}

~father1()

{

cout << "father1析构函数"<< endl;

}

};

class father2 : virtual public grandpa

{

public:

father2()

{

cout << "father2构造函数"<< endl;

}

~father2()

{

cout << "father2析构函数"<< endl;

}

};

class grandson : public father1, public father2

{

public:

grandson()

{

cout << "grandson构造函数" << endl;

}

~grandson()

{

cout << "grandson析构函数" << endl;

}

};

int main()

{

grandson A;

A.key = 1;

return 0;

}

>>grandpa构造函数

>>father1构造函数

>>father2构造函数

>>grandson构造函数

>>grandson析构函数

>>father2析构函数

>>father1析构函数

>>grandpa析构函数

# 多态性

## 5.1 多态性概念

在面对对象程序设计中，同样的方法被不同对象执行时，会有不同的效果。

多态的实现分为2种：静态联编和动态联编；其中函数重载、函数模板的实例化就属于静态联编，即在编译时确定操作过程，动态联编是在运行时才能确认执行哪段代码。

两者相比之下，静态联编由于编译时已经确认好怎么执行，所以执行的效率高；动态联编虽然慢一些，但胜在灵活，各有千秋。

注：当派生类和其父类都有一个相同的成员函数，但代码块不同，C++默认静态联编，所以根据对象的类型来决定调用哪个成员函数，即编译器看对象的类型。

Eg:

#define PI 3.1415926

class point

{

private:

int a;

int b;

public:

point(int x1, int x2)

{

a = x1;

b = x2;

cout << "point的构造函数" << endl;

}

int area()

{

cout << a << endl;

return a;

}

};

class circle :public point

{

private:

int r;

public:

circle(int q, int w, int radio):point(q,w)

{

r = radio;

cout << "circle的构造函数" << endl;

}

int area()

{

int ss = PI \* r \* r;

cout << ss << endl;

return ss;

}

};

int main()

{

circle A(1, 2, 3);

circle \*B = &A;

point \*D = &A;

circle &C = A;

point &E = A;

A.area();

B->area(); //因为B是circle类型，所以调circle的area()，即PI\*3\*3=28

D->area(); //因为D是point类型，所以调point的area()，即1

C.area(); //因为C是circle类型，所以调circle的area()，即PI\*3\*3=28

E.area(); //因为E是point类型，所以调point的area()，即1

return 0;

}

## 5.2 虚函数

（1）虚函数

形式：virtual 返回值 函数名（形参表）{实现体}

编译器看指针的内容，而不是它的类型。当我们在基类中声明了一个虚函数，如果他的派生类中又重新定义了该虚函数，此时会告诉编译器，这个函数不用静态联编，也就是说，只有在运行时，根据所调用对象的内容来选择调用的函数，这种操作叫做动态联编。

Eg：

virtual int area(); //class point中

int main()

{

circle A(1, 2, 3);

circle \*B = &A;

point \*D = &A;

circle &C = A;

point &E = A;

A.area();

B->area();

D->area();

C.area();

E.area();

return 0;

}

由于A是circle 类型，且基类point中已定义area（）为虚函数，B,C,D,E都是调用的A，所以在调用area（）时，都是调用的circle 的area（）函数，所以输出都是28。

（2）纯虚函数

在基类中定义一个虚函数，以便在派生类中重新定义该函数更好的适用于对象。但有时在基类中又不能给虚函数实际意义，这时用到纯虚函数。可以把基类中的虚函数写为：

Eg：

class point

{

private:

int a;

int b;

public:

point(int x1, int x2)

{

a = x1;

b = x2;

cout << "point的构造函数" << endl;

}

virtual int area() = 0; //虚函数

/\*{

cout << a << endl;

return a;

}\*/

};

（3）性质

1）虚函数不能是静态成员函数，或友元函数，因为他们不属于某个对象；

2）构造函数不能是虚函数，析构函数可以是虚函数，而且通常将基类的析构函数声明为虚函数；

3）虚函数在多态的情况下，只有在执行时才能确定调用的函数，这样的特性阻止了虚函数的内联。当然，如果是静态调用虚函数（没有多态），编译器还是会采取内联的形式（只有在派生类内重新定义虚函数时，才会告诉编译器，这个函数用动态编译）。

内联函数不是强制的，而是在编译时，给编译器提了个建议，允许他在可以内联的情况下采取内联。另外，对于虚函数来说，内联可能不成功，但并不是说，虚函数不能定义为内联函数。

注：

派生类circle指针操作派生类对象，area()函数不是虚函数：

#define PI 3.1415926

class point

{

public:

int area(int r)

{

cout << r << endl;

return r;

}

~point()

{

cout << "point的析构函数" << endl;

}

};

class circle :public point

{

public:

int area(int r)

{

int ss = PI \* r \* r;

cout << ss << endl;

return ss;

}

~circle()

{

cout << "circle的析构函数" << endl;

}

};

int main()

{

circle \*pp = new circle; //派生类指针操作派生类对象

pp->area(2);

delete pp;

return 0;

}

>>12

>> circle的析构函数

>> point的析构函数

此时area()调用的是circle类的函数（因为area不是虚函数，所以是静态链接，根据变量的类型来决定调用哪个函数），删除pp时，pp是circle类指针，所以先释放派生类资源，再释放基类资源，不会造成内存泄漏；

基类point指针操作派生类对象，area()函数不是虚函数：

int main()

{

point \*pp = new circle; //基类指针操作派生类对象

pp->area(2);

delete pp;

return 0;

}

>>2

>> point的析构函数

此时area()调用的是point类的函数（因为area不是虚函数，所以是静态链接，根据变量的类型来决定调用哪个函数），删除pp时，pp是point类指针，所以只释放了基类资源，没有释放派生类资源，会造成内存泄漏；

基类point指针操作派生类对象，area()函数是虚函数：

virtual int area(int r)；

int main()

{

point \*pp = new circle; //基类指针操作派生类对象

pp->area(2);

delete pp;

return 0;

}

>>12

>> point的析构函数

此时area()调用的是circle类的函数（因为area是虚函数，所以是动态链接，根据指针的内容决定调用哪个函数），删除pp时，pp是point类指针，所以只释放了基类资源，没有释放派生类资源，会造成内存泄漏；

基类point指针操作派生类对象，area()函数是虚函数，基类的析构函数也是虚函数：

virtual int area(int r)；

virtual ~point()；

int main()

{

point \*pp = new circle; //基类指针操作派生类对象

pp->area(2);

delete pp;

return 0;

}

>>12

>> circle的析构函数

>> point的析构函数

此时area()调用的是circle类的函数（因为area是虚函数，所以是动态链接，根据指针的内容决定调用哪个函数），删除pp时，pp是point类指针，基类指针操作派生类对象（基类指针指向派生类对象）时，如果基类的析构函数是虚函数，则释放基类指针时，可以同时释放掉派生类的空间，防止内存泄漏；

注：C++默认的析构函数不是虚函数，以为虚函数要占额外的内存空间，对于可能会被继承的基类，我们应当将其析构函数设置为析构函数，防止内存泄漏

在公有继承中，基类对派生类及其对象的操作，只能影响那些从基类继承下来的成员，如果想要用基类对非继承成员进行操作，则要把基类的这些函数定义为虚函数。

## 5.3 虚析构函数

在C++中，不能把构造函数定义为虚函数，因为在实例化一个对象时才会调用构造函数，且虚函数的实现本质上是通过一个虚函数指针来调用的，还没有对象更没有内存空间当然无法调用，故没有实例化一个对象之前的虚构造函数没有任何意义，也不可能实现。

但析构函数却可以是虚函数，且大多时候都声明为虚析构函数，这样就可以在用基类指针指向派生类对象释放时，根据实际所指向的对象类型动态联编调用派生类的析构函数，实现派生类和基类空间的共同释放，防止内存泄漏。

（1）point \*pp = new circle; //基类指针操作派生类对象，基类中的析构函数不是虚函数

delete pp; //释放时，仅调用基类的析构函数，会造成内存泄漏

（2）point \*pp = new circle; //基类指针操作派生类对象，基类中的析构函数为虚函数

delete pp; //释放时，先调用派生类的析构函数，再调用基类的析构函数防止内存泄漏

## 5.4 纯虚函数和抽象类

（1）纯虚函数

形式：virtual 返回值 函数名（参数表）= 0；

没有函数体的虚函数就叫做纯虚函数。包含纯虚函数的类就叫做抽象类，一个抽象类至少包含一个纯虚函数。

（2）抽象类

抽象类是为了提供一个高度抽象、对外统一的接口，然后通过多态的特性使用各自不同的方法，是C++面向对象设计及软件工程的核心思想。

特点：

1）抽象类无法实例出一个对象，只能作为基类让派生类完善其中的纯虚函数，然后再实例化使用；

Eg：

//point \*p = new point; //由于基类point中存在一个纯虚函数，所以它是一个//抽象类，抽象类不能实例化一个对象

point \*p1 = new circle1; //基类指针操作派生类circle1对象

p1->area(2);

point \*p2 = new circle2; //基类指针操作派生类circle2对象

p2->area(2);

delete p1;

delete p2;

return 0;

2）抽象类的派生类依然可以不完善基类中的纯虚函数，继续作为抽象类被派生，直到给出所有纯虚函数的定义，则成为一个具体类，才可以实例化对象。

（也就是说：基类中有一个纯虚函数，但是在它的派生类中并没有完善这个函数，这个派生类也就变成了抽象类，不能实例化了，直到纯虚函数被后面的派生类完善后，这个后面的派生类才变成具体类，才可以实例化）；

3）抽象类因为抽象，无法具化，所以不能作为参数类型，返回值，强转类型；

4）接第3条，抽象类一般是基类，抽象类可以定义一个指针、引用，指向其派生类（完善了纯虚函数），来实现多态特性。

Eg：

point \*p1 = new circle1; //基类指针操作派生类circle1对象（抽象类定义指针）

或

circle1 pg; //派生类circle1对象

point &p1 = pg; //基类（抽象类）定义引用

p1.area(2);

# 异常处理

## 6.1 异常的概念

程序错误通常有：语法错误，逻辑错误，运行异常。

语法错误：代码不符合语法要求，在编译、链接时由编译器提示的错误；

逻辑错误：编译没有错误，可以运行，但程序结果或执行过程不如我们所愿，到不到预期效果，需不断调试、测试来发现错误；

运行异常：程序在运行过程中，由于意外原因造成程序异常终止，比如内存不足、打开的文件不存在，除数为0等情况。

这里我们针对运行异常详细展开。通常，导致程序异常错误，虽然无法避免，但是却可以预料，进行预见性的处理可以避免程序崩溃，从而保障程序的健壮性。这种行为称为异常处理。

捕获和处理异常的方法有很多，比如通过if…else判断调用函数返回值，或对关键的数据进行检查等等，如果出现问题，则excit()或abort()来终止程序。（用if来对关键部分进行捕获和预防，但这种方式用多了会降低程序的易读性。C++提供了异常处理的方案）。

## 6.2 C++异常处理机制

C++提供了一种结构化形式的异常处理机制，可以把程序中正在执行的代码和异常处理部分分开表示，使程序清晰易读，更为优雅。

形式：

try

{

//正常程序执行语句

throw(异常类型表达式)；

}

……

catch(异常类型1) //catch块跟在try块后面，用于捕获异常

{

//异常处理代码

}

catch(异常类型2)

{

//异常处理代码

}

catch(异常类型3)

{

//异常处理代码

}

……

如果执行到的throw抛出来的异常类型在catch中没有对应的，则编译器自动调用terminate（）结束函数，默认功能是abort（）终止程序。如果执行的程序没有throw异常类型，则跳过catch，继续向下执行。

Eg：

float fact(int n,int m)

{

if (m == 0)

{

throw "error,被除数为0"; //抛出了一个类型为const char\*的异常，所以 //catch的异常类型应该是const char\*

}

else

{

throw 1;

return n / m;

}

}

int main()

{

float result;

try

{

result = fact(3, 0);

}

catch (const char \* str)

{

cout << str << endl;

}

catch (int)

{

cout << "int error" << endl;

}

catch (…)

{

cout << "any error" << endl;

}

return 0;

}

>> error,被除数为0

注：catch (…)可以匹配任何类型的异常，catch的优先级是从上到下，所以他后面的catch就不起作用了，因此一般把他放到最后。在执行try的某一行代码块时，一旦他throw一个异常，会立马执行catch，try下面的就不执行了，如果在本函数没有处理该异常则会返回给上一层函数，如果没有处理该异常的catch，则编译器自动调用terminate（）结束函数。

## 6.3 C++异常规格说明

为了增强程序的可读性和可维护性，使程序员在使用一个函数时就可以看出这个函数可以抛出什么异常类型，C++允许在函数定义和声明时，加上它所能抛出的异常列表：

void func() throw (int, double, A, B, C);

func()函数可以抛出int,double,A,B,C五种类型的异常。其中throw()不是函数，而是异常规格说明，表示func()函数可以抛出异常的类型。

void func() throw ( ); //表示func函数不会抛出任何异常；

void func() //没有交代可以抛出的异常类型，则表示func函数会抛出任何异常；

注：throw()类型声明表在不同编译器下要求不一样，有的要求函数所抛出的异常类型必须是类型声明表中的一个，否则报错，而在用 Visual Studio 2010 编译出来的程序则不会出错，异常声明列表不起实际作用。

Eg：

float fact(int n,int m) throw(int, const char \*) // func()函数可以抛出int,和const char \*两种类型的异常。

{

if (m == 0)

{

throw "error,被除数为0";

cout << " fact " << endl;

}

else

{

throw 1;

return n / m;

}

}

Eg2：

try

{

throw point(); // point是一个类，throw的异常类型为point类，这样//会调用这个point类的构造函数

}

catch (point & A) //捕获point类的异常类型

{

cout << A.func() << endl;

}

另外：

string s = "lz"

const char \* func()

{

return s.c\_str();

}

C语言中没有string类型，为了兼容C语言，可以通过string类对象的成员函数c\_str()把string对象转换为C中的字符串格式。

原型：const char \*c\_str(); //赋值是要使用strcpy()函数等来操作方法c\_str()返回的指针

Eg2：

char c[20];

string s="1234";

strcpy(c,s.c\_str()); // s.c\_str()返回的是临时指针，不能对其进行操作