目录

[结构体 2](#_Toc15828070)

[Namespace.Cpp 2](#_Toc15828071)

[New.cpp 3](#_Toc15828072)

[星号的作用 4](#_Toc15828073)

[引用 4](#_Toc15828074)

[其他类型的引用 5](#_Toc15828075)

[引用做参数 5](#_Toc15828076)

[引用返回值 6](#_Toc15828077)

[For的增强 7](#_Toc15828078)

[默认值 7](#_Toc15828079)

[函数重载 8](#_Toc15828080)

[Class 9](#_Toc15828081)

[访问修饰符（public） 10](#_Toc15828082)

[访问修饰符和继承（protected） 11](#_Toc15828083)

[友元 12](#_Toc15828084)

[构造函数 13](#_Toc15828085)

[含参数的构造函数（头文件的使用） 14](#_Toc15828086)

[初始化与赋值 16](#_Toc15828087)

[初始化列表 16](#_Toc15828088)

[Heat1.h 17](#_Toc15828089)

[引用和const初始化 18](#_Toc15828090)

[数组和结构体使用初始化列表 19](#_Toc15828091)

[析构函数 20](#_Toc15828092)

[This 21](#_Toc15828093)

[常函数 22](#_Toc15828094)

[static 23](#_Toc15828095)

[静态函数的累加 24](#_Toc15828096)

[拷贝构造函数 24](#_Toc15828097)

[默认拷贝构造函数 25](#_Toc15828098)

[深拷贝 26](#_Toc15828099)

[内联函数 28](#_Toc15828100)

[运算符重载 29](#_Toc15828101)

[istream运算符重载 30](#_Toc15828102)

[ostream运算符重载 31](#_Toc15828103)

[二元运算符重载 32](#_Toc15828104)

[一元运算符重载 33](#_Toc15828105)

[对象的强制转换 34](#_Toc15828106)

[下标运算符重载 35](#_Toc15828107)

[自加自减 36](#_Toc15828108)

[继承（子类对父类） 37](#_Toc15828109)

[继承限定词 38](#_Toc15828110)

[构造函数的继承 39](#_Toc15828111)

[含参数的构造函数的继承 40](#_Toc15828112)

[析构函数的继承 41](#_Toc15828113)

[继承含相同成员的覆盖 42](#_Toc15828114)

[虚函数（父类对子类） 43](#_Toc15828115)

[虚函数的特点（重写） 44](#_Toc15828116)

[虚表 46](#_Toc15828117)

[虚析构 47](#_Toc15828118)

[纯虚函数 48](#_Toc15828119)

[虚继承 49](#_Toc15828120)

[联编 50](#_Toc15828121)

[单例 51](#_Toc15828122)

[异常 52](#_Toc15828123)

[内部类 54](#_Toc15828124)

[类型转换 55](#_Toc15828125)

[函数模板 56](#_Toc15828126)

[类模板 58](#_Toc15828127)

[继承模板 59](#_Toc15828128)

[多态模板 60](#_Toc15828129)

[类型是类的模板 61](#_Toc15828130)

## 结构体

//也可以使用c语言的头文件

//#include<stdio.h>

#include <iostream>

using namespace std; //命名空间

//结构体是特殊的类，操作与类一致

struct Node

{

int n=12;

void fun();

};

int main()

{

Node no;

no.fun();

cout<<no.n<<endl;

system("pause");

return 0;

}

//函数fun()类内声明，类外定义

void Node:: fun()

{

cout << n << endl;

}

## Namespace.Cpp

#include<iostream>

using namespace std;

namespace stu/\*命名空间的声明\*/

{

void sort()

{

cout << "stu" << endl;

}

}

namespace stu1

{

void sort()

{

cout << "stu1" << endl;

}

}

int main()

{

//using namespace stu;/\*可以放在第11和第19行但是不能放在第3行，命名空间不能放在命名空间的声明之前\*/

/\*using namespace stu; \*//\*也可以输出stu1,也以放在第19行，不能放在第11和第3行，第12行是stu1的命名空间的声明\*/

stu1::sort(); //这里使用了stu::sort()或者stu1::sort()，而不用using namespace stu的命名空间也能的带相同的结果，::是作用于运算符

system("pause"); //方便查看结果，防止结果的一闪而过

return 0;

}

## New.cpp

#include<iostream>

#include <stdlib.h>

using namespace std;

struct Node

{

int a = 12;

};

int main()

{

//单个空间的内存分配，c语言的方法

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

int \*p1 = new int(121); //new+type(初始化的值) 类型的匹配

char \*p2 = new char; //可以不初始化

\*p2 = 'a'; //写

cout << \*p1 << "\t" << \*p2 << endl; //读

delete p1,p2; //删除指针

////结构体生成对象空间申请

Node \*p=new Node;

cout << p->a << endl; //使用指针来调用结构体中的成员

////数组P分配空间内存

int \*P = new int[5];

memset(P, 0, 5 \* 4); // memery set 给数组初始化（初始化为同一个值），初始化了就不用在给数组赋值了，第一个填空间的起始地址，第二个填的是值，第三个填空间的大小int是4个字节，共有5个元素

//void \*P= malloc(5\*4); // c语言的方法

//不使用memset函数，赋值每个数组元素

// P[0] = 12;

// P[1] = 23;

// P[2] = 32;

// P[3] = 44;

// P[4] = 51;

cout << P[0] << " " << P[1] << " " << P[2] << " " << P[3] << " " << P[4] << endl;

delete[] P; //[]是指数组的意思

system("pause");

return 0;

}

## 星号的作用

#include<iostream>

using namespace std;

int main()

{

//声明变量的时候\* 指针变量

int a = 12;

int \*p = &a;

//地址操作符 读 写

\*p;

cout << \*p << endl; //读

\*p = 123; //写 可以针对\*p进行写操作

cout << a << endl;

//数字的相乘

int b = 12;

a = b \* 13;

cout << "a= " << a << endl;

//对内存的操作 读 写 取地址&

system("pause");

return 0;

}

## 引用

#include<iostream>

using namespace std;

int main()

{

//变量起别名

int a = 12; //引用声明时必须初始化

int &b = a; //声明a的一个引用 b，b是a的一个别名

int &c = a; //一个变量可以多个引用

cout << &a << " " << &b << " " << &c << endl; //同一个变量的不同引用的地址是相同的

int &d = b; //一个引用可以被另外的量引用

// int f = 15;

// int &b = f; //已经引用过了的b不能再引用别的变量

b = 13; //可以对引用赋值，变量也会跟着赋值

cout << b << " " << d << " " << a << endl;

system("pause");

return 0;

}

## 其他类型的引用

#include<iostream>

using namespace std;

int main()

{

//常量引用 12 12.23 'q'

const int &a = 12; //对于常量的引用必须加const，此后a不能被修改，只能读

cout << a << endl;

//数组的引用

int arr[12];

int(&b)[12] = arr; //先引用 后类型！！

b[2] = 13;

cout << arr[2] << endl;

//二维数组的引用

int arr2[2][3];

int(&c)[2][3] = arr2; //先引用 后类型！！

c[1][2] = 123;

cout << arr2[1][2] << endl;

//指针的引用

int d = 12;

int \*point = &d;//声明一个指向d的指针point，这里的&是取地址符号

int\* (&e) = point;//指针的引用，这里的&是引用符号,在这的括号可以去了

\*e = 24; //给指针赋值

cout << \*point << " " <<d << endl;

system("pause");

return 0;

}

## 引用做参数

#include<iostream>

using namespace std;

//引用做参数

void fun(int &a) //引用做参数，a接受b的赋值（a=b），a可以改变fun的内部值（a=new a） a可以反过来改变外部b的值(b=new a)，（int &a）是形参

{

a = 13;

cout << a << endl;

}

void fun1(int a) //普通变量做参数 a接受b的赋值(a=b)，a可以改变fun1的内部值(a=new a) 但a没法改变外部b的值(b==b)，（int a）是形参

{

a = 14;

cout << "fun1= " << a << endl; //a=14

}

void fun2(int \*a) //指针做参数，b的地址传递给a指针的地址做初始化，函数的内部对\*a进行地址的重新赋地址，也就是对b的地址做修改地址的操作，（int \* a）是形参

{

\*a = 15;

cout << \*a << endl;

}

int main()

{

int b = 12;

fun(b); //b=12 带入fun 得a=12 然后a赋新值=13 所以输出a=13 故b=new a=13

cout << b << endl;

fun2(&b); //输入b=13，&b取地址给a，经过\*a得到b的值13，然后经过\*a=15赋值，输出\*a为15，而b的值也被赋值为15

cout << b << endl;

fun1(b); //b=15带入fun1中，所以a=15，然后a=赋新值=14 所以输出a=14，而b值却没有发生改变，b=15

cout << b << endl;

system("pause");

return 0;

}

## 引用返回值

#include <iostream>

using namespace std;

int &fun()

{

int a = 12;

return a; //这里的a是局部变量

}

int main()

{

int &b = fun(); //a是局部变量，调用完了fun()之后，a的地址将会被释放

cout << b << endl; //b的操作是非法空间，其结果是未知的

system("pause");

return 0;

}

## For的增强

#include <iostream>

using namespace std;

int main()

{

//int a;

//a = 0;

//for ( a = 0; a < 5; a++) //这里也可以去掉去掉前面的对a的定义 在a=0前可以加上int 即int a=0 是没有问题的 这就是for的增强

// cout << a << endl;

//for ( a = 0; a < 5; a++) //这里还可以接着进行一个一样的for循环，也是for的增强

// cout << a << endl;

for (int a = 0; a < 5; a++) //这里的int a的int类型的作用域只是在局部的作用，不能作用于别的地方

cout << a << endl;

for (int a = 0; a < 5; a++)

cout << a << endl;

system("pause");

return 0;

}

## 默认值

#include <iostream>

using namespace std;

void fun(int a = 12, char c = 'c') //全部指定

{

cout << a << " " << c << endl;

}

void fun1(int a , char c = 'c', float b = 12.22) //部分指定 必须从右往左依次指定 连续

{

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

}

//假如将函数定义放在主函数后面，则要把函数声明放在主函的前面，并且在声明后面加上分号，在函数定义后面不用加分号

void fun2(int a=15, char d='d'); //在函数声明的地方已经对参数进行了赋值，就不用在后面的函数定义那里再进行赋值，否则就会重定义

//或则在声明和定义时都不对参数进行赋值，在主函数的调用时进行传递实参（赋值）

int main()

{

//函数调用时，如果有默认值的参数，可以不用传递实参,直接调用

fun();

//没指定默认值的参数，一定要传递实参来调用

fun1(123); //这里的123就是对fun1的int a进行传递实参

//有默认值，再传递实参，将会覆盖默认值

fun(13 ,'b'); //也可以传一个

fun2( );

system("pause");

return 0;

}

void fun2(int a , char c )

{

cout << a << " " << c << endl;

}

## 函数重载

#include <iostream>

using namespace std;

void fun(int a)

{

cout << "1:" << a << endl;

}

void fun(int a, int b)

{

cout << "2:" << a << " " << b << endl;

}

void fun(char c, float d)

{

cout << "3:" << c << " " << d << endl;

}

//void fun(int a)

//{

// cout << "1:" << a << endl;

//}

//void fun(char b)

//{

// cout << "2:" << b << endl;

//}

//void fun(float c)

//{

// cout << "3:" << c << endl;

//}

int main()

{

//调用函数时，只要对相应的函数参数类型进行赋值就行，要一一对应

fun(12);

fun(15, 20);

fun('e', 12.23); //这里的12.23后面一定要加上一个f，与前面的函数的参数类型要一致，因为计算机对小数的类型默认是double类型的,但是这里不加f也得到了结果,而在下面的例子中却必须加上f才有结果??

//函数的重载的好处，函数调用更加灵活

//fun(12.44f); //在这里的函数调用时，里面实参可以随意选择上面的函数参数类型，已达到调用目的

system("pause");

return 0;

}

## Class

#include <iostream>

using namespace std;

class cpeople // 类的声明 构造了cpeople的一个类，就是在people前加上了一个c

{

public: //在这要注明 访问修饰符:public private protected.如果没有注明就是默认为private

int a; //类里面就包括数据成员和下面的函数成员。**在这不给a赋初始值的原因是，只有静态常量整型才可以在在这初始化（在C++11之后就可以初始化了）。**

void fun() //函数成员

{

cout << "fun=" << a << endl;

}

}; //这里的花括号后面要加上分号！！

int main()

{

//这里构造的对象是要具有相同的属性（数据成员）和相同的行为（函数成员） 只有在构造对象的时候才会给对象分配空间，成员才会被分配空间，这和指针的一开始new的分配空间后才能调用是一个道理

cpeople op; //在这是用cpeople这个类构造了op这个对象

cpeople op1; //还可以构造许多个类

op.a = 12; //调用数据成员

op.fun(); //调用函数成员

cpeople\* op1 = new cpeople; //用指针的方法构造一个对象，一定要分配空间

op1->a = 13; //指针调用数据成员

op1->fun(); //指针调用函数成员

delete op1; //要注意记得释放new的空间

system("pause");

return 0;

}

## 访问修饰符（public）

#include <iostream>

using namespace std;

class cpeople

{

//访问修饰符 作用范围：从书写开始到下一个修饰符，或则到类的结尾的花括号

public: //类外可见 是整个程序公用的 在下面的class cbird类、自定义的函数和主函数内都是可用的 一般是将函数成员public

//private: //类内可见 是所在类中私用的 如果不加public就是默认private 一般是将数据成员private

int a;

void fun()

{

cout << "fun=" << a << endl;

}

};

//public的类外可见可用

class cbird

{

public:

cpeople op2;

void fun2()

{

op2.fun();

}

};

void fun()

{

cpeople op1;

op1.fun();

}

int main()

{

cpeople op;

op.fun();

system("pause");

return 0;

}

## 访问修饰符和继承（protected）

#include <iostream>

using namespace std;

//三种访问修饰符的作用

//private： 纯私有 类内使用

//protected： 受保护 类内和子类可以使用

//public： 公有的 都可以使用

class cpeople

{

private:

int a;

protected: //受保护 类内和子类可以使用

void fun()

{

cout << "fun=" << a << endl;

}

};

//子类对父类的继承，就可以对protected进行访问

class cxm :public cpeople

{

public:

void fun4()

{

fun();

}

};

//非子类的类调用protected是错误的

//class cbird

//{

//public:

// cpeople op2;

// void fun2()

// {

// op2.fun();

// }

//};

//类外调用protected是错误的

//void fun()

//{

// cpeople op1;

// op1.fun();

//}

int main()

{

cpeople op;

//op.fun(); //类外调用protected的函数是错误的

cxm op1; //子类调用是允许的

op1.fun4();

system("pause");

return 0;

}

## 友元

#include <iostream>

using namespace std;

class cstu

{

private: //访问修饰符具有封装性，权限

//protected： 这里是protected也可以

int age;

void fun()

{

age = 12;

cout << "age=" << age << endl;

}

//使用接口函数，也能达到从外面访问private的内容

public:

int Get()

{

return age;

}

void set()

{

age = 15;

}

friend void fun1(); //友元 类成员对于这个函数相当于public 直接是：friend+函数声明

friend int main(); //这里也可以直接对主函数进行友元

friend class cteach; //还可以对其他类进行友元

};

void fun1()

{

cstu stu;

stu.fun();

}

class cteach

{

public:

cstu stu2;

void fun2()

{

stu2.fun();

}

};

int main()

{

// fun1(); //调用自定义函数，从而访问类的private和protected的内容

// cstu stu; //主函数也可以访问private和protected的内容

// stu.fun();

// cteach teach; //构造老师的类，访问学生类的private或protected的内容

//teach.fun2();

cstu stu1; //构造stu1这个对象

stu1.set();

int a = stu1.Get();

cout << a << endl;

system("pause");

return 0;

}

## 构造函数

#include <iostream>

using namespace std;

class cstu

{

public:

int age;

float f;

//构造函数: 使用构造函数后在主函数创建对象时不需要人为去调用所要的内容，而是自动调用内容，直接输出就行

cstu() //没有返回值

{

age = 12;

f = 12.12f;

}

// //没有构造函数时，则要定义函数，对数据成员进行赋值

//void fun()

//{

// age = 12;

// f = 12.12f;

//}

};

//构造函数：1、作用，对数据成员赋值，以便调用方便 2、执行时间，对象创建时自动调用构造函数

int main()

{

cstu stu; //对象创建时自动调用构造函数

//stu.fun(); //没有构造函数时，人为去调用函数，然后去输出

cout << stu.age << endl; //直接输出调用的内容

cstu \*stu1 = new cstu; //指针走构造函数的步骤，要new空间,在new了新的空间时，才会调用构造函数

delete stu1;

system("pause");

return 0;

}

含参数的构造函数（头文件的使用）

#include <iostream>

#include "heat1.h"

using namespace std;

//heat1的代码如下：

//新建的头文件，包含函数声明，然后在别的文件中直接调用，就不用多次的函数声明，直接对函数定义就行，在定义时一定加上类名和作用域运算符cstu:: 在另外的文件调用格式是：#include "heat1.h"

//class cstu

//{

//public:

// int age;

// float f;

// cstu(int a = 15, float b = 12.23f); //如果这么声明，这里的第一个参数必须要有，第二个可以没有，如果赋值了就会把构造函数的初始化给覆盖了

// //cstu(int a, float b); //如果这么声明，后面的主函数传参两个参数都必须要有

//};

//有了自定义的头文件heat1后以下的类声明可以省略

class cstu

{

public:

int age;

float f;

//构造函数在类内声明与定义的写法 以下包含构造函数的重载

cstu(int a = 12, float b = 12.23f) //可以指定默认值

{

age = a;

f = b;

}

//构造函数在类内声明，在类外定义的写法（如下）。 普通函数也可以这么操作，记得要加类名和作用域运算符cstu::

cstu(int a = 15, float b = 12.5f);

};

//可直接使用类名和作用域运算符来定义构造函数

cstu::cstu(int a, float b) //在函数定义时，里面的参数不用赋值

{

age = a;

f = b;

}

int main()

{

cstu stu(12); //构造函数参数传递，这里的第一个参数必须要有，第二个可以没有，如果赋值了就会把构造函数的初始化给覆盖了 如cstu stu(12,15.23);的输出是12 15.23

//cstu stu(12,15.56); //对应的第二种声明的传参写法

cstu\*stu1 = new cstu(13, 13.5f); //指针构造函数参数传递

cout << stu.age << " " << stu.f << endl;

cout << stu1->age << " " << stu1->f << endl;

system("pause");

return 0;

}

## 初始化与赋值

#include <iostream>

using namespace std;

int main()

{

int a = 12; //对a初始化

int b; //对b赋值

b = 13;

int c[12] = { 1,2,3,4 }; //对数组进行初始化

int d[12]; //赋值

d[1] = 10;

d[2] = 11;

struct stu //结构体的初始化和赋值

{

int e;

};

stu f = { 15 }; //结构体初始化声明变量f初始化形式

f.e = 12; //f的赋值形式

int g = 20; //引用初始化

int &h = g;

const int i = 15; //const 使用时要初始化，如果没有及时赋值，就不能赋值了

system("pause");

return 0;

}

## 初始化列表

#include <iostream>

//#include "heat1.h"

using namespace std;

class cstu

{

public:

int a;

float f;

//cstu() :a(12), f(12.23f) //初始化列表 ！！！成员初始化顺序只与声明顺序有关，与初始化列表书写顺序无关

//{

//}

cstu(int b,float c):a(b),f(c) //通过参数传递对a和f进行初始化，这里或则可以只对其中的一个进行传递参数初始化：cstu(int b):a(b),f(12.23f) 后面就对b传参

{

//a=13; 里面如果再给a赋值就会覆盖a的初始化值

}

void show()

{

cout << a << " " << f << endl;

}

};

int main()

{

//cstu stu;

cstu stu(15,15.56); //对应的参数传递对a和f初始化

stu.show();

system("pause");

return 0;

}

## Heat1.h

//新建的头文件，包含函数声明，然后在别的文件中直接调用，就不用多次的函数声明，直接对函数定义就行，在定义时一定加上类名和作用域运算符cstu:: 在另外的文件调用格式是：#include "heat1.h"

//class cstu

//{

//public:

// int age;

// float f;

// cstu(int a = 12, float b = 12.23f);

//};

//如果有多个文件要使用内联函数，就把内联函数定义放在头文件里

void fun1();

inline void fun()

{

cout << endl;

}

## 引用和const初始化

#include <iostream>

//#include "heat1.h"

using namespace std;

class cstu

{

public:

int b;

int& a;

const int e;

//cstu() :a(b), b(12) //成员初始化

//{

// cout << a << " " << b << endl;

//}

//cstu(int c) : a(b), b(c) //参数初始化

//{

// cout << a << "\t" << b << endl;

//}

//cstu(int& c) : a(c), b(c),e(c) //类外初始化

//{

// cout << a << " " << b << " " << e << endl;

//}

cstu(int& c,int f) : a(c), b(c), e(f) //类外初始化

{

cout << a << " " << b << " " << e << endl;

}

//void show()

//{

// cout << a << " " << b << " " << e << endl;

//}

};

int main()

{

//cstu stu; //引用成员

//cstu stu();

//对不同变量传值时只要直接传参就好

//cstu stu(13); //引用参数

//引用类外，对引用传值时要另外对被引用变量赋值

//int d = 14;

//cstu stu(d);

int d = 14; //多个构造函数，参数对应的相应的构造函数

cstu stu(d,16);

/\*stu.show();\*/

system("pause");

return 0;

}

## 数组和结构体使用初始化列表

#include <iostream>

using namespace std;

//结构体可以直接初始化 可以在类外或则类内初始化

struct STU

{

int i;

float f;

};

class cstu

{

public:

int a[4];

cstu() :a()

{

}

STU st; //结构体构造对象

cstu(STU sd) :st(sd)

{

memset(&a[0], 0, 16);

}

void show()

{

int i;

for (i = 0; i < 4; i++);

{

cout << a[i] << endl;

}

}

};

int main()

{

cstu stu;

stu.show(); //输出有错

//STU s1 = { 12,12.23f };

//STU s2 = s1; //结构体可以相互赋值

//cout << s2.i << " "<<s2.f << endl;

//STU sf = { 123,123.23f };

//cstu stu(sf);

//cout << sf.i << " " << sf.f << endl;

system("pause");

return 0;

}

## 析构函数

#include <iostream>

using namespace std;

class cstu

{

public:

int nage;

int \*pp;

cstu()

{

cout << "构造函数"<<endl;

pp = new int(10);

}

cstu(int i)

{

cout << "构造函数" << endl;

}

~cstu() //析构函数形式不能重载，不能有参数，自动调用，作用域是主函数结束之前（return 0）或则有些是在 }花括号之前执行调用

{

cout << "析构函数" << endl;

delete pp; //一般在析构函数地方释放空间

}

};

int main()

{

//{

// cstu stu; //在局部变量里，析构函数在花括号 } 前调用析构函数

//}

//逐语句执行有错

//cstu stu; //在主体函数里，析构函数是在return 0; 之前调用析构函数

//逐语句执行有错

//cstu \*stu = new cstu; //析构函数在指针对象时，只有遇到delete释放空间时，才会调用析构函数

//delete stu;

//逐语句执行有错

cstu(12); //临时对象的构造 如 cstu tu(12) 就是构造了tu的临时对象

// 临时变量的作用域就是在当前语句中，语句将要结束时自动调用析构函数

// 也可以构造cstu() 这样无参的临时对象 所以构造临时对象时，要对应前面的构造函数的参数类型

//int c = int(12); 类型+（参数）构造临时对象 如q=12 c=q ：先构造一个q的临时变量，然后q对c进行赋值

//int b = 12; 直接给b赋值

system("pause");

return 0;

}

## This

#include <iostream>

using namespace std;

class cstu

{

public:

int a;

cstu(int a)

{

this->a = a; //this->a可以认为是cstu\*的a 就是外面的int a

this->show();

}

void show() //this不是成员，是类成员函数的隐含参数（show函数没有参数时，系统会默认给show函数的第一个参数是this）

// 所以在类内的各函数中是可以调用，但是不能在类内别的位置直接使用

{

cout << a << endl;

//this->a; //如在这里在show函数里面调用this

}

cstu\* GetAddr() //得到对象地址 或则得到对象\*this

{

return this;

}

};

int main()

{

cstu st(12); //没有对象，就没有this指针

st.show();

//st.this; //this不是成员， 所以这里是调用不了this的

cstu\* p= st.GetAddr(); //得到st的地址

cout << p << endl;

p->show();

cstu st1(13);

p = st1.GetAddr(); //得到st1的地址

cout << p << endl;

system("pause");

return 0;

}

## 常函数

#include <iostream>

using namespace std;

class cstu

{

public:

int a;

//const 不能用于构造函数和析构函数

cstu()

{

a = 12;

}

~cstu()

{

}

//const的使用方法和使用位置

void show()const //const cstu\* 指针的const

{

// a = 13; //常函数不能修改数据成员，可以输出数据成员int a

int b = 2; //常函数可以修改内部的数据

b = a\*b;

//this->a = 15; 错误的操作，this也不能修改数据成员

cout << b<<" "<<a<<" I am show" << endl;

}

void fun()

{

cout << "i am fun" << endl;

}

};

int main()

{

const cstu stu; //这里构造了常对象stu，常对象只能调用常函数，不能调用普通函数

stu.show();

//stu.fun(); //这里常对象试图调用普通函数，是错误的操作

cstu stu1; //普通对象可以调用常函数，也可以调用普通函数

stu1.show();

stu1.fun();

system("pause");

return 0;

}

## static

#include <iostream>

using namespace std;

class cstu

{

private:

//static int a;

public:

static int a; //静态函数参数不能在类内初始化，可以在类外初始化

static const int b = 15; //在C++11之前的版本（C++98/C++03）只有 静态常量整型 数据成员才能在类中初始化

int b=15; //在C++11之后的版本是允许初始化的

static void fun() //静态成员函数的使用

{

cout << "i am static" << endl;

a = 20;

cout << a << endl;

// b = 25; //静态成员函数不能使用普通的数据成员，只能使用静态的成员函数

}

cstu()//:a(2) //这里是无法给静态函数进行初始化的

{

a = 12; //重新给a进行赋值 假如这里没有赋值，那么在输出a的值就会将静态成员函数的a的值输出

}

};

int cstu::a = 13; //静态函数在类外初始化

int main()

{

// 这里可以使用类名作用域来调用静态成员函数和静态函数，说明静态成员函数和静态函数是在类创建的时候申请空间的

cout << cstu::a << endl;

cstu::fun();

//这里也可以使用生成对象的方式来调用静态成员和静态函数

cstu stu;

//stu.a;

cout << stu.a << endl;

stu.fun();

system("pause");

return 0;

}

## 静态函数的累加

#include <iostream>

using namespace std;

class cstu

{

private:

//static int a;

public:

static int b;

cstu()

{

b++; //重新给b累加

}

};

int cstu::b = 0; //静态函数在类外初始化

int main()

{

cstu st[5]; //这里构造5个对象，分别去构造函数那里累加

cout << cstu::b << endl; //输出最后的b的值

system("pause");

return 0;

}

## 拷贝构造函数

#include <iostream>

using namespace std;

class cstu

{

public:

cstu()

{

}

cstu(const cstu&a) //拷贝构造函数的形式和使用方法，本质是构造函数，参数是本类的常数引用

{

}

};

int main()

{

//几种现有对象给新的对象赋值，走拷贝构造函数的情况

//拷贝构造的作用将一个对象赋值给另外一个对象

cstu st1; //1、一个对象给新的对象初始化，走拷贝构造函数

//cstu stnew(st1);

//cstu stnew1 = st1; //2、 新的对象给现有对象进行赋值，走拷贝构造函数

//cstu stnew2 = cstu(st1); //3、临时对象给现有对象进行赋值，走拷贝构造函数

cstu\*stnew3 = new cstu(st1); //4、对指针的操作，走拷贝构造函数

system("pause");

return 0;

}

## 默认拷贝构造函数

#include <iostream>

using namespace std;

class cstu

{

public:

int b;

char c[4];

cstu()

{

b = 12;

c[0] = 'a';

strcpy(c, "abc"); //字符串拷贝函数，第一个参数是目标字符串的首地址，第二个参数是源

}

//默认的拷贝构造函数是要执行拷贝的操作，而默认的构造函数是不执行任何操作

//cstu(const cstu &a) //这里是默认拷贝构造函数，是不执行的，默认内容为空。如果写了拷贝构造函数，就会走

//如果写了拷贝构造函数，如下所示，就会走拷贝构造函数

//{

//this->b=a.b;

//strcpy(this->c,a.c);

//}

};

int main()

{

cstu st;

cout << st.b << " " << st.c << endl;

cstu st1 = st;

cout << st1.b << " " << st1.c << endl;

system("pause");

return 0;

}

## 深拷贝

#include <iostream>

using namespace std;

class cstu

{

public:

int\* a;

cstu()

{

//在构造函数对a进行初始化并且赋予空间

a = new int[2];

a[0] = 12;

a[1] = 13;

}

//同一个地址多次释放空间 系统就会崩溃，为避免这个问题，使用深拷贝方法

//当是指针成员是，要使用深拷贝，重新给a赋予空间 。没有指针成员就使用浅拷贝

cstu(const cstu&b)

{

//拷贝构造函数对a进行空间的申请

this->a = new int[2];

//this->a[0] = b.a[0]; //进行复制

//this->a[1] = b.a[1];

memcpy(this->a, b.a, 8); //复制函数，当有多个变量需要复制时，可以使用这个函数

}

~cstu()

{

delete[]a;

}

/\*cstu(const cstu &a) //这里是默认拷贝构造函数，是不执行的，默认为空。如果写了拷贝构造函数，就会走

{

}\*/

};

cstu& fun(cstu& a) //传递引用，则不需要调用拷贝构造函数，从而解决以上问题，也提高了代码的效率

{

return a;

}

cstu\* fun(cstu\* a) //同理传递指针的方法也不需要调用拷贝构造函数

{

return a;

}

int main()

{

//在这at=st 在执行过程中，会两次调用析构函数，而且两次先后释放都是同一个空间，就会崩溃

{

cstu at;

cout << at.a[0] << " " << at.a[1] << endl;

fun(at); //对应传递引用的方法

fun(&at); //对应传递指针的方法

cstu st;

cout << st.a[0] << " " << st.a[1] << endl;

}

system("pause");

return 0;

}

## 内联函数

#include <iostream>

using namespace std;

//内联函数的作用：在主函数调用其他函数的过程中，将其他函数的代码复制到主函数调用的位置

//区别于普通函数调用的方式

//比常规函数稍快，但是需要的内存更多

//当函数体过大时，递归函数都不能调用内联函数

//内联函数定义一般写在头文件里

#define SUM(x) x\*x //调用宏,功能是宏参数自乘

inline void fun(int i)

{

cout << (i\*i) << endl;

}

class cstu

{

public:

void fun() //没加上inline就是默认为隐式定义的内敛函数

{

}

//inline void fun() //加了inline就是显示定义

//{

//}

void fun1();

};

inline void cstu::fun1() //如果这里没有加inline，就不是内联函数

{

}

int main()

{

cout << (SUM(2 + 3)); //使用宏的输出的结果是：2+3\*2+3=11 ((x)\*(x))这个结果与内联函数是一样的

fun(2 + 3); //内联函数结果的结果是：（2+3）\*（2+3）=25

system("pause");

return 0;

## 运算符重载

#include <iostream>

using namespace std;

class cstu

{

public:

int nage;

double dscore;

cstu()

{

nage = 12;

dscore = 12.12;

}

};

//重载运算的限定词就是operator，后面接什么运算符就是表示哪种运算符重载。在类外声明定义

//和函数重载一样，在主函数相应的运算要与前面定义运算符重载的类型和顺序一致。

//在括号内第一个参数可以是传对象，最好使用传引用或者指针（避免调用拷贝构造函数）这里的对象不用和主函数声明的对象一致

//第二个参数可以是运算符操作的另外一个参数

void operator +(cstu& st, int a)

//这里是无返回值的

{

cout << st.nage + a << endl; //在输出时把两个部分进行运算操作

}

void operator +(int a, cstu& st)

//这里括号里的参两个数可以互换位置

//这里是无返回值的

{

cout << st.nage + a << endl;

}

void operator +(cstu& st1, cstu& st2) //这里括号里的参两个数可以互换位置

//这里是无返回值的

{

cout << st1.nage + st2.nage << endl;

}

int operator +(cstu& st, int a)

//这里是有返回值的

{

return (st.nage + a);

}

cstu& operator +(cstu& st, cstu& st1) //这里是两个对象的运算，所以返回值是类的类型cstu

//其次因为作为返回值，return会调用拷贝构造函数，所以在这传递引用，就可避免调用

{

st1.nage += st.nage;

return st1;

}

int main()

{

//普通的+运算符

// int a = 13;

// a = 13;

//运算符重载的应用：两个对象直接运算操作系统会报错，或者是对象直接与数据做运算也是会报错

//故使用运算符重载去解决问题，如上所示

cstu st1, //连续声明两个对象1和对象2，之间用逗号 , 分开 最好分行书写

st2;

st1 + 15; //对象与整数的运算

15 + st1;

st1 + st2; //可以进行两个对象的运算

//连算时前两个按照运算符重载计算完之后，需要有返回值与第三个数进行计算

cout<<(st1 + 15 + 13)<<endl;

cout << (st1 + st2) + 23 << endl;

system("pause");

return 0;

}

不能重载的运算符为 :: .\* . ?:

## istream运算符重载

#include <iostream>

using namespace std;

class cstu

{

private:

int nage;

double height;

public:

//cstu() //数据成员一般放在private里面

//{

//nage = 0;

//height = 0;

//}

void show()

{

cout << nage <<" "<<height<< endl;

}

friend istream& operator >>(istream& ist, cstu& st); //使用友元来访问private的数据成员，类内声明，类外定义

};

istream& operator >> (istream& ist, cstu& st)

{

ist >> st.nage>>st.height ;

//如果输入有错，就使输出的值初始化

if (ist.fail())

{

st.nage = 0;

st.height = 0;

}

return ist;

}

int main()

{

cstu st;

cin >> st;

st.show();

system("pause");

return 0;

}

## ostream运算符重载

#include <iostream>

using namespace std;

class cstu

{

private:

int nage;

public:

cstu()

{

nage = 12;

}

friend ostream& operator <<(ostream& os, const cstu& st);

};

//输出必须是在类外重载

//void operator <<(ostream& os,const cstu& st) //不修改就要加上const

//{

// os<<st.nage ;

//}

ostream& operator <<(ostream& os, const cstu& st)

{

os << st.nage;

return os;

}

int main()

{

cstu st;

cout << st << endl;

system("pause");

return 0;

}

## 二元运算符重载

#include <iostream>

using namespace std;

//二元运算符是指运算符两端有两个操作数

//一元运算符是指运算符两端有一个操作数

//同理三元运算符是指运算符两端有三个操作数如 ？：

class cstu

{

public:

int nage;

/\*cstu()

{

nage = 10;

}\*/

int operator >= (cstu& st2) //类内运算符重载的形式 关系运算符 >=

{

return (nage >= st2.nage);

}

};

//类外关系运算符重载>= 以及其他运算符重载

//int operator >=(cstu& stu1, cstu& stu2)

//{

// return (stu1.nage >= stu2.nage);

//}

int main()

{

cstu stu1;

stu1.nage = 15;

cstu stu2;

stu2.nage = 20;

cout << (stu1 >= stu2) << endl;;

system("pause");

return 0;

}

## 一元运算符重载

#include<iostream>

using namespace std;

class cstu

{

public:

int nage;

cstu(int age)

{

nage = age;

}

//重载类内的写法（默认有this指针）

int operator -()

{

return (-nage);

}

};

// // 重载类外的写法 不能与类内的定义重载了

//int operator -(cstu& dt)

//{

// return (-dt.nage);

//}

int main()

{

cstu st1(14);

cout << -st1 << endl; //在这需要对 - 进行重载

system("pause");

return 0;

}

## 对象的强制转换

#include <iostream>

using namespace std;

//对象的强制转换要求

//没有显示返回类型，但是要写返回值

//没有参数

//必须定义成类的成员函数

//不应该改变对象的内容，所以是const函数

class cstu

{

public:

int a;

double b;

cstu()

{

a = 13;

b = 12.12;

}

operator int() const //对象的类型转换形式， 不修改值，就加上const

{

return a;

}

operator double() const //类型转换形式，没有返回类型，但是有返回值 必须放在类内

{

return b;

}

};

int main()

{

//(int)a 或者 a(int) 数值的两种类型转换的形式：前者是c语言中的写法，后者是C++的写法

float a = 12.12f;

cout << (int)a << endl;

//对象的类型的转换，推荐使用第一种

cstu st;

cout << (int)st << endl;

cout << (double)st << endl;

system("pause");

return 0;

}

## 下标运算符重载

#include <iostream>

using namespace std;

//[]的三种作用:声明变量的时候有[]，表示声明的是数组

//函数有[]表示指针

//地址+[]表示下标运算

class cstu

{

public:

int a;

int b;

int c;

double d;

int nError; //在这定义了错误返回的变量

cstu()

{

a = 12;

b = 23;

c = 34;

d = 45;

nError = -1; //对错误返回变量进行赋值

}

//下表运算只能在类内声明定义

//在主函数里需要返回的是成员，所以在这要返回引用

//int& operator [](int n)

//{

// switch (n)

// {

// case 0:

// return a;

// case 1:

// return b;

// }

// return nError; //在返回的是引用变量，所以要返回错误变量nError

//}

//非整形的指针下标运算符

void\* operator [](int n) //针对非整形的下表运算，使用指针取地址的方法

{

switch (n)

{

case 0:

return &a;

case 1:

return &d;

}

return &nError;

}

};

int main()

{

int a[10]= { 1,2,3 };

a[1] = 13; //在这就是下标运算对a[1]进行赋值，就是等价于 \*(a+1)=13 可读写操作

//cstu st; //st[1]是类中的第二个成员

//cout << st[1] << endl; //读

//st[1] = 15; //写

//cout << st[1] << endl;

//非整形的指针下标运算符

cstu st;

cout << \*(double \*)st[1] << endl; //void\*使通用的类型转换 可以看成是中转站 void没有确定的大小，所以明确取double类型大小的空间内容 类型转换

cout << \*(int \*)st[0] << endl; //对int的取值操作

//指针的写 与读

\*(double \*)st[1] = 15;

cout << \*(double \*)st[1]<< endl;

system("pause");

return 0;

}

## 自加自减

**++a 返回的是加了之后的值**

**a++ 返回的是加之前的值**

#include <iostream>

using namespace std;

class cstu

{

public:

int nage;

cstu()

{

nage = 12;

}

//类内重载

//int operator++()

//{

// nage += 1;

// return nage;

//}

//类内重载 this->的方法

//int operator--()

//{

// this->nage -= 1;

// return nage;

//}

};

//类外前置++

int operator++(cstu& st) //类外重载

{

st.nage += 1;

return st.nage;

}

//类外前置--

int operator--(cstu& st) //类外重载

{

st.nage -= 1;

return st.nage;

}

//类外后置++

int operator ++(cstu& st, int n) //int n是用来区分前向自加还是后向自加的标志

{

int a = st.nage; //用变量a记录自加前的值

st.nage += 1; //nage自加

return a; //返回自加前的值

}

int main()

{

// int a=15;

cstu st;

//前置++ int b = ++a; b=13 a=13

cout << ++st << endl; //输出13

cout << --st << endl; //输出12

//后置++ int b = a++; //b=12 a=13

cout << st++ << endl; //这里输出12

cout << st.nage << endl; //这里输出13

system("pause");

return 0;

}

## 继承（子类对父类）

**父类含参的构造函数在继承的子类的构造函数的初始化列表去传递参数**

#include <iostream>

using namespace std;

class cpeople //基础类 父类

{

public:

void study()

{

cout << "study" << endl;

}

};

//多个子类可以继承同一个父类

class cchild:public cpeople //派生类 子类

{

public:

void gotoschool()

{

cout << "gotoschool" << endl;

}

};

class cadlt :public cpeople //派生类 子类

{

public:

void gotowork()

{

cout << "gotowork" << endl;

}

};

class colder :public cpeople //派生类 子类

{

public:

void gotopark()

{

cout << "gotopark" << endl;

}

};

int main()

{

//子类生成的对象可以直接调用父类的成员

cchild child;

child.study();

//指针的调用方法

cchild \* child1 = new cchild;

child1 -> study();

system("pause");

return 0;

}

## 继承限定词

#include <iostream>

using namespace std;

class cpeople //基础类 父类

{

private:

void fun1()

{

cout << "father fun1" << endl;

}

protected:

void fun2()

{

cout << "father protacted" << endl;

}

public:

void fun3()

{

cout << "father public" << endl;

}

};

//public 继承之后，子类继承了父类的所有成员，父类的public成员在类内类外都能访问。

//但是父类的protected成员只能在子类的类内访问，在类外不能访问。

//父类的private成员子类在类内类外都不能访问，除非是子类友元。

//protected 继承之后，父类的public降级为protected 。

//就是被子类继承的父类public成员变成protected成员，被继承的父类protected成员和private成员还是原来的特性。

//private 继承之后，父类所有的成员都不能被子类调用

class cXM :public cpeople //派生类 子类

{

public:

int a;

void gotoschool()

{

cout << "gotoschool" << endl;

}

void fun4()

{

fun2();

}

};

int main()

{

cXM xm;

xm.fun3();

xm.fun4();

system("pause");

return 0;

}

## 构造函数的继承

#include <iostream>

using namespace std;

class cpeople //基础类 父类

{

public:

cpeople()

{

cout << "cpeople" << endl;

}

};

class cXM :public cpeople //派生类 子类

{

public:

cXM()

{

cout << "cXM" << endl;

}

};

class cXH :public cXM //派生类 子类

{

public:

cXH()

{

cout << "cXH" << endl;

}

};

int main()

{

//构造函数继承的调用顺序是先调用父类的构造函数，然后是调用子类构造函数

cXM xm;

cXH xh;

system("pause");

return 0;

}

## 含参数的构造函数的继承

#include <iostream>

using namespace std;

class cgrandfather //祖父类

{

public:

cgrandfather(int a,int b)

{

cout << "i am grandfather" << endl;

}

cgrandfather(int c)

{

}

};

class cfather: public cgrandfather //基础类 父类

{

public:

cfather(int a): cgrandfather(1,2) //父类对祖父类进行了继承，所以要对其参数初始化，但下面的子类不能对祖父类进行参数初始化

{ //基类有多个构造函数，通过子类的初始化列表的形式选择性对应去调用那个父类的构造函数

cout << "i am fatherclass" << endl;

}

};

class cson :public cfather //派生类 子类

{

public:

int a;

cson():cfather(1) //父类的构造函数有参数的时候，要在子类构造函数的初始化列表进行初始化传参

{ //谁继承，谁对其父类进行参数初始化

cout << "cXM" << endl;

}

};

int main()

{

cfather father(1);

cson son;

system("pause");

return 0;

}

## 析构函数的继承

#include <iostream>

using namespace std;

//析构函数的调用是，先从子类开始调用，然后调用父类的析构函数，依此往上调用

class cgrandfather

{

public:

cgrandfather()

{

cout << "cgrandfather" << endl;

}

~cgrandfather()

{

cout << "cgrandfather" << endl;

}

};

class cfather : public cgrandfather

{

public:

cfather()

{

cout << "cfather\n";

}

~cfather()

{

cout << "cfather\n";

}

};

class cson :public cfather

{

public:

cson()

{

cout << "cson\n";

}

~cson()

{

cout << "cson\n";

}

};

int main()

{

{

cson so;

}

system("pause");

return 0;

}

可设断点一步一步进入查看调用顺序

## 继承含相同成员的覆盖

#include <iostream>

using namespace std;

//子类有与父类相同的成员时，在主函数调用成员时会默认调用子类的成员,故覆盖了父类的成员

class cfather

{

public:

int a;

cfather()

{

a=12;

}

int fun(int a)

{

cout << "cfather fun\n";

return 0;

}

};

class cson:public cfather //如果这里不写继承的限定词，那就是默认为private

{

public:

int a;

cson()

{

a = 10;

}

void fun()

{

//输出成员变量

cout << a << endl;

cout << cfather::a << endl; //解决方法是使用类名作用域的方法去调用父类的成员，如cfather::a cfather::fun(1)

//输出成员函数

cout << "cson fun" << endl;

cout << cfather::fun(1) << endl;

}

};

int main()

{

cson so;

so.fun();

//so.fun(1); //在父类中fun是有参数的，但是在子类继承中会覆盖父类的成员，所以在这调用含参数是错误的

so.cfather::fun(1);

system("pause");

return 0;

}

## 虚函数（父类对子类）

**虚函数只针对函数成员，并不指数据成员**

#include <iostream>

using namespace std;

//虚函数，就是父类对子类的调用

//多态思想，父类的指针调用子类的函数

//父类对子类的调用，必须是父类和子类有一样的成员

class cfather

{

public:

virtual void show() //虚函数的写法 在这里加了virtual，那在主函数的指针调用就可以调用到子类的show函数

{ //这里的调用必须是父类和子类的函数成员名字一样都为show（）

cout << "class cfather\n";

}

};

class cson :public cfather

{

public:

int a;

virtual void show() //这里子类的virtual可加可不加

{

cout << "class cson\n";

}

};

class cson1 :public cfather

{

public:

int a;

void show()

{

cout << "class cson1\n";

}

};

int main()

{

//父类对子类的调用 只能使用父类的指针调用子类的函数

cfather\*fa = new cson; //指针调用子类谁的成员，是要看指针指向谁的空间。在这是cson申请了空间，故fa指针指向cson的show（）

fa->show(); //普通的情况（对象调用），前面没有virtual，这里调用的结果只能是子类对父类继承的show()函数

cfather\*fa1 = new cson1; //在这是cson1申请了空间，故fa指针指向cson1的show（）

fa1->show();

//子类对父类的继承

cfather fa2; //父类的函数前有virtual 可是使用对象调用时，任然是调用子类对父类继承的show()函数

fa2.show();

system("pause");

return 0;

}

## 虚函数的特点（重写）

#include <iostream>

using namespace std;

//重写就是父类和子类相同，父类是虚函数

//虚是指函数成员

//子类重写的函数，默认是虚函数，可以显示加上vritual，或者不加

class cfather

{

public:

//inline virtual void show() //这里加了inline可是对程序没什么作用，所以说虚函数不能是内联函数

//virtual cfather () //虚函数也不能是构造函数,内联是构造函数的唯一合法存储类

virtual void show()

{

cout << "class cfather\n";

}

////重写函数的形式和返回值必须一致，除非是协变

////在这重写的函数类型不一致的例子（cfather的show函数带参数）

////cfather类的指针调用cson类的函数输出是class cfather，而不是class cson，故重写错误

////子类的show（）函数也传参（int a），则输出是class cson，重写正确

//virtual void show(int a)

//{

// cout << "class cfather\n";

//}

////在这重写的函数是有返回值（int show）

////在这是直接报出 返回类型与重写虚拟函数 "cfather::show" 的返回类型 "int" 既不相同，也不协变

////在子类写成int show（），则输出正确。在子类的show函数中记得加上返回值：return 0;

//virtual int show()

//{

// cout << "class cfather\n";

// return 0;

//}

////协变的形式

//virtual cfather& show()

//{

// cout << "class cfather\n";

// return (\*this);

//}

};

class cson :public cfather

{

public:

int a;

void show()

{

cout << "class cson\n";

}

////子类的协变形式😂只有这样返回类型不一致，也能重写

// cson& show()

//{

// cout << "class cson\n";

// return (\*this);

//}

};

//测试cson类重写了cfather之后，show（）函数有默认的virtual，在这也可以显示的加上virtual

class cgrandson :public cson

{

public:

int a;

void show()

{

cout << "class cgrandson\n";

}

};

int main()

{

cfather\*fa = new cson;

fa->show(); //普通的情况，前面没有virtual，那就是父类的指针去调用子类的空间，所以这里调用的结果只能是父类的show()函数

//测试cson类的指针调用cgrandson类的函数

//输出是class cgrandson

cson \*son = new cgrandson;

son->show();

system("pause");

return 0;

}

## 虚表

#include <iostream>

using namespace std;

class cfather

{

public:

virtual void fun()

{

cout << "class cfather\n";

}

virtual void show()

{

cout << "class cfather\n";

}

};

class cson :public cfather

{

public:

void show()

{

cout << "class cson\n";

}

};

int main()

{

//父类所有的虚函数都按地址存放在虚表里，每次系统都会检查父类的函数成员是否含有virtual。

//如果有，就会进入虚表，按照地址取相应的虚函数。

//对象首地址的前四个字节就是虚表的首地址

//虚表里的各个虚函数的类型可能是不一致的

cfather\* fa=new cson;

fa->show();

//\*fa 是整个对象的地址

//\*(int \*)fa 取对象的前四个字节，也就是进入虚表首地址

//(int \*)\*(int \*)fa; //把虚表的首地址强转换成int类型，然后整个虚标的虚函数类型都是int类型，可进行地址增量自加，取到下一个虚函数。在这然是取地址操作

//\*((int \*)\*(int \*)fa + 0); //这里再取\*就是对虚表首地址的首元素

//\*((int \*)\*(int \*)fa + 1); //这里再取\*就是对虚表第二个地址的首元素

//取出的内容是int的函数类型，还要将函数转换成与父类的虚函数类型相一致

//typedef是将void函数指针类型名命为p

//((p)(\*((int \*)\*(int \*)fa + 0))) 然后整体强转成p的类型，就是一个函数指针了

//在后面加（）就可以调用这个函数了，如下所示

typedef void(\*p)();

((p)(\*((int \*)\*(int \*)fa + 0)))();

输出有问题

system("pause");

return 0;

}

## 虚析构

#include <iostream>

using namespace std;

//在多态环境中，如果释放父类指针，只会调用父类的析构函数。

//如果在父类的析构函数前加上virtual，据构成了虚析构，就可以调用子类和父类的析构函数。

//先是调用子类的析构函数，然后是父类的析构函数

class cfather

{

public:

virtual ~cfather()

{

cout << "cfather\n";

}

};

class cson :public cfather

{

public:

~cson() //子类的析构函数前面是默认有一个virtual,也可以显示的加上virtual

{

cout << "cson\n";

}

};

class cgrandson //:public cson

{

public:

~cgrandson()

{

cout << "cgrandson\n";

}

};

int main()

{

//cfather\*fa = new cson;

//delete fa;

//cson \* son = new cgrandson;

//delete son;

cfather\*fa = new cson; //前提是cgrandson没有继承cson的类。

delete (cgrandson\*)fa; //delete哪个类型的指针，就调用谁的析构函数，这里强转( cgrandson\* )的类型指针，所以调用cgrandson的析构函数

//如果cgrandson类继承了cson类，那么这里调用的还是cson类和cfather类的析构函数

system("pause");

return 0;

}

## 纯虚函数

#include <iostream>

using namespace std;

//含纯虚函数的类是不能在主函数里直接生成对象

//但是子类继承含有纯虚函数的类，并且成员函数要一致。这样子类可以在主函数生成对象

//只要含有纯虚函数的就是抽象类（可以含有别的类型函数）

class cfather

{

public:

virtual void fun() = 0;

void show()

{

}

};

////全都是纯虚函数和纯虚析构函数的叫做接口类

////变量和构造函数是不能成为纯虚函数，故可以写在接口类里

//class cstu

//{

//public:

// int a;

// cstu()

// {

//

// }

// virtual void fun() = 0;

// virtual ~cstu() = 0;

//};

//子类也继承了父类的纯虚函数

class cson :public cfather

{

public:

void fun() //函数名要与父类的纯虚函数名一致，否则子类也无法在主函数里生成对象

{

cout << "cson\n";

}

};

int main()

{

cson pf;

pf.fun();

system("pause");

return 0;

}

## 虚继承

#include <iostream>

using namespace std;

//虚继承是解决多继承调用访问不明确的问题

//虚继承不建议使用，结构复杂，内存开销较大

//B和C都继承了A的数据成员a，那么B和C都相当于复制了A的数据成员a。

//然后D类继承了B和C，相当于D复制了B和C的数据成员a。

//那么D里面就有两个数据成员a，在主函数的调用时就会显示调用不明确的问题

//整个虚继承的过程中只有A类的一个数据成员a，B类、C类和D类只是得到了A类中数据成员a的使用权，并没有复制a。

//就不会出现a调用访问不明确的问题

class cA //虚基类

{

public:

int a;

};

class cB :virtual public cA //虚继承的写法，就是在继承的前面加上virtual

{ //这里B虚继承了A 只是从A那里得到a的使用权，而不是像继承那样直接复制父类A

public:

};

class cC :virtual public cA //这里C虚继承了A 这里也是得到a的使用权

{

public:

};

class cD :public cB,public cC //这D继承了B和C 也是相当于得到a的使用权。

{

public:

};

int main()

{

cD d;

d.a;

system("pause");

return 0;

}

## 联编

#include<iostream>

using namespace std;

//联编就是相当于函数调用的过程，分为静态联编和动态联编

//一般来说都是静态联编(在编译期系统就能知道的函数的调用)

//动态联编（在运行的时候系统才知道调用的函数）用于多态的特殊条件下

class CA

{

public:

virtual void fun()

{

cout << "CA" << endl;

}

};

void fun1()

{

cout << "CA" << endl;

}

class CB :public CA

{

public:

void fun()

{

cout << "CB" << endl;

}

};

int main()

{

//普通的静态联编

CA fa;

fa.fun();

fun1();

//多态的动态联编

CA\*a ;

int b;

cin >> b;

switch(b)

{

case 1: a = new CA; break;

case 2: a = new CB; break;

}

a->fun();

system("pause");

return 0;

}

## 单例

#include <iostream>

using namespace std;

//单例的思想是在外部的主函数里，类只生成一个对象

//当类内的构造函数是private或则protected时，主函数就无法使用类生成对象了，就限制了对象的生成

//类中的非静态成员，在类外的主函数里必须用对象去调用。

//静态成员函数，既可以使用对象去调用，也可以用类名作用域去调用

class cfather

{

private:

cfather()

{

}

cfather(const cfather &a) //拷贝构造函数系统会默认写在public中

//所以避免生成对象时系统默认生成拷贝构造函数，将其主动写在private中

{

}

cfather &operator=(cfather &a) //同理复值函数（=的重载函数）也是系统会默认写在public中，也将其写在private中

{

}

public:

static int nflag; //静态成员要在类外初始化

static cfather \* CreateQJ()

{

if (nflag == 1)

{

nflag = 0;

return (new cfather);

}

else

{

return NULL;

}

}

~cfather() //使在主函数可以再次申请一个对象

{

nflag = 1;

}

};

int cfather::nflag=1; //类外初始化，要使用类名作用域来调用

int main()

{

//cfather as;

cfather \* pf = cfather:: CreateQJ();

delete pf;

cfather \* pf1 = cfather:: CreateQJ();

delete pf1;

//if (pf1 = NULL)

//{

//}

system("pause");

return 0;

}

## 异常

#include <iostream>

#include <cstdlib>

using namespace std;

//使用引用，避免拷贝构造

class cpeople

{

public:

int b;

cpeople()

{

b = 12;

}

};

void fun(int a) //abort（）的异常处理方式

{

if (a == 0)

{

abort();

}

cout << "a" << a << endl;

}

void fun1(int a)

{

while (a < 10)

{

if (5 == a)

{

throw a; //抛出一个数

//trow 12.12; //抛出一个浮点数，对应下面的默认形式

}

a++;

//cout<<a<<endl;

}

}

//使用引用和指针，避免拷贝构造

void fun2(cpeople &c)

{

while (c.b < 20)

{

if (15 == c.b)

{

throw c; //抛出引用c

//throw &c; //抛出地址c

}

c.b++;

//cout<<a<<endl;

}

}

int main()

{

//abort()异常函数的实现

fun(1);

////try catch异常的实现

try

{

fun1(2);

}

//catch (int b)

//{

// cout << b << endl;

//}

//输出比5大的能继续运行

catch (int b)

{

try

{

fun1(b + 1);

}

catch (int a)

{

}

}

//当catch的类型没有写全，使用这种形式也可以操作

catch (...)

{

cout << "default" << endl;

}

//抛出引用和指针

cpeople po;

try

{

fun2(po);

}

catch (cpeople& a) //抛出引用

{

}

catch (cpeople \*d) //抛出地址

{

d->b++;

}

system("pause");

return 0;

}

## 内部类

#include <iostream>

using namespace std;

class Cout

{

public:

int a;

//Cout()

Cout():in (this) //在外部类的构造函数给内部类的对象传递指针

{

a = 12;

}

//内部类的创建 也是嵌套类

public:

class Cin

{

public:

int b;

Cout \*p;

Cin(Cout \*pf):p(pf)

{

b = 13;

}

void fun()

{

//cout << a << endl; //内部类不能直接使用外部类的成员

//Cout ou; //可以使用外部类的对象来调用成员

//cout << ou.a << endl;

cout << p->a << endl;

}

};

//在外部类生成一个内部类的对象

public:

Cin in;

};

int main()

{

//主函数里调用内部类的fun函数

//在外部类里用内部类生成一个对象，在主函数里通过如下调用

Cout oi;

oi.a = 15;

oi.in.fun();

system("pause");

return 0;

}

## 类型转换

#include <iostream>

using namespace std;

class cfather

{

public:

int a;

virtual void fun()

{

}

};

class cson :public cfather

{

public:

int b;

void fun()

{

}

};

class cother

{

public:

double f;

};

int main()

{

cfather \*p;

cson \*s;

cother \*o;

//static\_cast的类型转换，可用于不同关系的类的转换

p=(cfather \*)s; //旧式强制转换

p = static\_cast<cfather \*>(s); //新式的强制转换

//新型强制转换不允许两个不相关的类强制转换，如下把s强制转换成cother类，是错误的

//o = static\_cast<cother \*>(s);

//但是使用旧式的强制类型转换时，编译不会报错。

//但是会将a和b都转换成double类型，后面的调用时就会出现不清楚的情况，如下所示

//o= (cother \*)s;

////reinterpret\_cast的类型转换，用于天生危险的类型转换

//o = reinterpret\_cast<cother\*>(s);

//struct dat

//{

// short a;

// short b;

//};

//long value = 0xA224B118;

////struct结构体里含有两个二字节的short类型的变量，所以整个结构体是四字节

////而long类型也是四字节，将long的value转换成结构体的dat\*类型，则是将long的value拆分成两个二字节

//dat\* p1 = reinterpret\_cast<dat\*>(value);

//cout << hex << p->a;

////dynamic\_cast的类型转换，多用于有继承关系的两个类之间的转换

//p = dynamic\_cast<cfather\*>(s); //将子类转换成父类

//s = dynamic\_cast<cson\*>(p); //将父类转换成子类，要求必须是多态的条件下

//cson\*s1 = dynamic\_cast<cson\*>(s); //自己转换成自己的类

////const\_cast的类型转换

//const cfather\* p;

//cson\* so;

//cfather\* p1 = const\_cast<cfather\*>(p); //将const cfather\* p的const去掉之后的cfather\* p的类型转换

////cson\* p1=const<cson\*>(p); //在这去掉const之后任然是cfather\* 的类型，故在这使用cson\*的类型转换是无效的

////so=(cson\*)p; //在这使用旧式的强制类型转换系统是不会报错的，但是也是会存在错误的

system("pause");

return 0;

}

## 函数模板

#include<iostream>

using namespace std;

//模板的思想：泛型编程思想。使用一个函数体，实现多个功能

//函数模板如下，T是通用的类型。这样就可以传递多个类型的变量

//在这可以写成template<class T> class与typename相同

//template<typename T>

//void fun(T a)

//{

// cout << a << endl;

//}

////可以写多个通用的类型

//template<typename T, typename Y>

//template<typename T, class Y> //typename和class可以同时使用

//void fun(T a,Y b)

//{

// cout << a <<" "<< b << endl;

//}

struct node

{

int a;

double b;

};

template<typename T>

void fun(T a)

{

cout << a << endl;

}

//函数模板具体化

//函数的具体化：就是针对特殊问题，把特殊的模板具体写一遍,优于模板。普通函数最先执行

//template<>void fun <node>(node no)

//{

// cout << no.a << " " << no.b << endl;

//}

//template<>void fun<int>(int no)

//{

// cout << no<< endl;

//}

//void fun(int a)

//{

//

//}

//函数模板实例化

//就是在这生成指定函数的定义

template void fun<int>(int a);

int main()

{

//模板函数的调用

//fun(12.12);

//模板函数具体化的调用

//node no = { 12, 12.12 };

//fun(no);

//模板函数实例化的调用

//fun(12);

system("pause");

return 0;

}

////之前的要写多个函数

//void fun(int a)

//{

// cout << a << endl;

//}

//void fun(double d)

//{

// cout << d << endl;

//}

## 类模板

#include<iostream>

using namespace std;

//除了类外，任何位置出现cfather都要加上模板参数列表（就是模板的通用形式，如：）,而且都要一致

//模板类的写法

//template <typename T=char> //定义默认的char类型

//template <typename T=char，typename Y=int> //参数的指定是从右向左，在主函数的传递时也要从右向左传

//template <typename T，typename Y=int> //在这也可只给右边第一个类型指定类型

//类模板的作用域是在这个类的作用域，在类外如有需要使用到类模板，则需要重新加上类模板的声明

//template <typename T>

template <typename T = char, typename Y = int>

class cfather

{

public:

T a;

cfather(T t)

{

a = 10; //赋值

}

//void fun()

//{

// cout << a << endl;

// }

//在这是fun函数在类内声明在类外定义，如下所示

void fun();

};

template <typename T = char, typename Y = int> //在上一条的类模板作用域之外，要重新加上类模板的声明

void cfather<T, Y>::fun() //在这要加上类名的作用域运算符，a参数才可以被识别

{

cout << a << endl;

}

//在C++11编译器函数模板可以使用参数默认值，之前的编译器不允许使用

template<typename T = char>

void fun(T t)

{

}

int main()

{

//cfather<char> fa('a'); //如果类模板没有默认的类型，则在这里的cfather后加上对应所要传递的参数类型，也就是模板参数列表

cfather<> fa('a'); //如果类模板有包含所要传递的参数类型，在cfather后的尖括号就不用传递类型了，也就是模板参数列表不需要传递

//cfather<int,char> fa(13，'a'); //多类型时，类型传递从右向左

//cfather<int> fa(13); //也可只转递一个类型

//cfather<int, char> fa(12); //模板参数列表<int,char>，给构造函数传参可以只传递一个参数

fa.fun();

cfather<int, char>\* pf = new cfather<int, char>(12); //（12）是构造函数传参

pf->fun();

system("pause");

return 0;

}

## 继承模板

#include<iostream>

using namespace std;

template<typename T=char,typename Y=int>

class cfather

{

public:

T a;

//cfather() //父类构造函数无参数

cfather(T t)

{

a = t;

}

void show()

{

cout << "show" << endl;

}

};

//子类没有模板

//class cson:public cfather<int,char> //在这的cfather后面必须加上模板初始化列表

//{

//public:

// //cson() //父类的构造函数无参数子类的构造函数写法

// cson():cfather<int,char>(12) //父类的构造函数是含参数的，所以在子类构造函数的初始化列表去传参。在这里的cfather不用加模板初始化列表，加了可以

// {

//

// }

//};

template<typename T , typename Y > //在这没有指定的类型，所以在子类的构造函数里可以随意传递类型

class cson :public cfather<T, Y> //在这的cfather后面必须加上模板初始化列表

{

public:

//cson() //父类的构造函数无参数子类的构造函数写法

cson():cfather<T, Y>(12) //父类的构造函数是含参数的，所以在子类构造函数的初始化列表去传参。在这里的cfather不用加模板初始化列表，加了可以

{

}

};

int main()

{

////子类没有模板，向父类的模板传递类型

//cson so;

//子类是模板，向父类的模板传递类型

cson<int ,char> so;

so.show();

system("pause");

return 0;

}

## 多态模板

#include<iostream>

using namespace std;

template<typename T = char, typename Y = int>

class cfather

{

public:

virtual void fun()

{

cout << "cfather" << endl;

}

};

////子类含模板

//template<typename M,typename T, typename Y >

//class cson :public cfather<T, Y>

//{

//public:

// void fun()

// {

// cout << "cson" << endl;

// }

//

//};

//子类不含模板

class cson :public cfather<int, char>

{

public:

void fun()

{

cout << "cson" << endl;

}

};

int main()

{

//子类含模板，传递类型

//cfather<int,char>\* fa = new cson<short,int, char>; //cson<short,int, char>这里的类型要与子类的模板一致,有三个类型，而且是一一对应，分别对应 M,T,Y

//fa->fun(); //前面的cfather<int,char>的类型要与子类继承父类的模板一致public cfather<T ,Y>,而且要一一对应,分别是 T,Y

//子类不含模板，传递类型

cfather<int, char>\* fa = new cson;

system("pause");

return 0;

}

## 类型是类的模板

#include<iostream>

using namespace std;

template<typename T>

class cA

{

public:

int a;

};

template<typename T , typename Y >

class cfather

{

public:

cfather(cA<char>& a)

{

}

void fun()

{

cout << "cfather" << endl;

}

};

int main()

{

cA<char> ca;

cfather <int, cA<char>>pf(ca);

system("pause");

return 0;

}