第8课　多继承、异常

内容概述

1. 多重继承概念
2. 多重继承构造与析构
3. 多重继承赋值兼容
4. 多重继承名字查找
5. 多重继承虚函数表
6. 虚继承和虚基类
7. 虚继承对象内存分布
8. 虚继承练习
9. 多重继承、虚继承小结
10. 异常概念
11. 栈展开
12. catch匹配规则
13. noexcept
14. 标准库异常类
15. 析构函数与异常
16. 构造函数与异常
17. 异常安全
18. 异常练习
19. 异常小结

多继承：引入

多重继承：是指从多个直接基类中产生派生类的能力。
多重继承的派生类继承了所有基类的成员（数据成员和成员函数），除了构造函数（指:默认构造、拷贝构造、移动构造）和析构函数。

```
using Engine = int;
using Wheel = int;
using Propeller = int;
class vehicle { //车 类
public:
    void run() { cout << "车在马路上跑\n"; }
private:
    Engine engine; //引擎
    Wheel wheel;   //轮子
};
class ship { //船 类
public:
    void swim() { cout << "船在水中漂\n"; }
private:
    Engine engine;        //引擎
    Propeller propeller; //螺旋桨
};
```

```
class Carship :public vehicle, public ship {
private:
    int special;
};
int main() {
    Carship cs;
    cs.run();
    cs.swim();
    return 0;
}
```

```
车在马路上跑
船在水中漂
请按任意键继续
```

Carship cs对象:

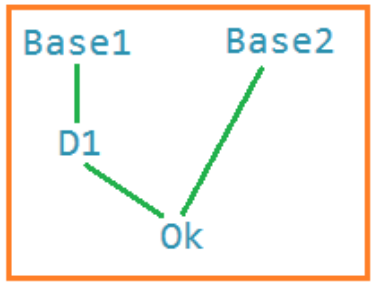
vehicle:	Engine engine; Wheel wheel;
ship:	Engine engine; Propeller propeller;
Carship:	int special;

多继承：构造与析构

派生类的构造函数会分别调用其直接基类的构造函数(默认构造或指定构造)，调用次序与派生类列表中基类的出现次序一致(与初始值列表中的次序无关)。析构次序与构造次序相反。

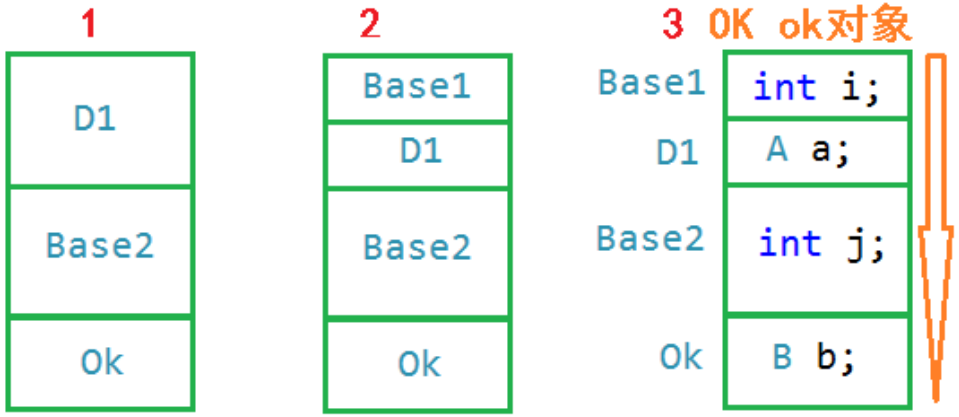
```
struct A { A() { cout << "A()\n";} };
struct B { B() { cout << "B()\n"; } };
class Base1 {
public:
    Base1() { cout << "Base1()\n"; }
private:
    int i;
};
class Base2 {
public:
    Base2(int jj):j(jj) { cout << "Base2()\n"; }
private:
    int j;
};
class D1 :public Base1 {
public:
    D1() { cout << "D1()\n"; }
    //实际是：D1():Base1(),a()
private:
    A a;
};
```

继承关系：




```
class Ok :public D1, public Base2 {
public:
    Ok():Base2(2) { cout << "Ok()\n"; }
    //实际是：Ok():D1(),Base2(2),b()
private:
    B b;
};

int main() {
    Ok ok;
    return 0;
}
```

```
Base1()
A()
D1()
Base2()
B()
Ok()
请按任意键
```



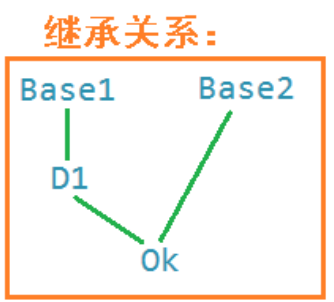
多继承：拷贝构造与赋值

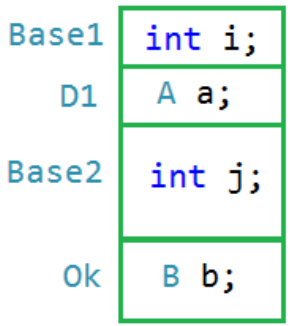
编译器合成的派生类拷贝构造函数：
会按次序调用各个基类的拷贝构造来完成其基类部分的拷贝。而自己定义的话则必须显式地调用基类的拷贝构造(否则会调用基类的默认构造)。

合成的赋值运算符类似。

```
struct A { int ai; A(int i) :ai(i){} };
struct B { int bi; B(int i) :bi(i){} };
class Base1 {
public:
    Base1(int ii):i(ii){}
    int i;
};
class Base2 {
public:
    Base2(int jj):j(jj) {}
    int j;
};
class D1 :public Base1 {
public:
    D1(int i):Base1(i),a(i){}
    A a;
};

int main() {
    Ok ok1(10);
    cout << ok1.a.ai << " " << ok1.b.bi << endl;
    Ok ok2(1);
    ok2 = ok1;
    cout << ok2.a.ai << " " << ok2.b.bi << endl;
}   return 0;
```

```
class Ok :public D1, public Base2 {
public:
    Ok(int i):D1(i),Base2(i),b(i) { cout << "Ok()\n"; }
    Ok(const Ok& other)
        :D1(other),Base2(other),b(other.b){}
    Ok& operator=(const Ok& other) {
        if (this == &other) return *this;
        D1::operator=(other);     //调用基类赋值
        Base2::operator=(other); //调用基类赋值
        b = other.b;
        return *this;
    }
    B b;
};
```

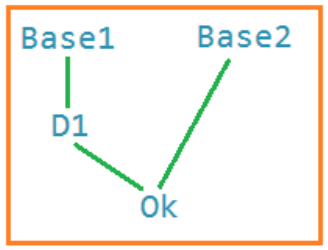




多继承：赋值兼容、类型转换

```
struct A { int ai=1;};
struct B { int bi=2;};
class Base1 {
public:
    int i=3;
};
class Base2 {
public:
    int j=4;
};
class D1 :public Base1 {
public:
    A a;
};
class Ok :public D1, public Base2 {
public:
    B b;
};
```

继承关系：



Base1	int i;
D1	A a;
Base2	int j;
Ok	B b;

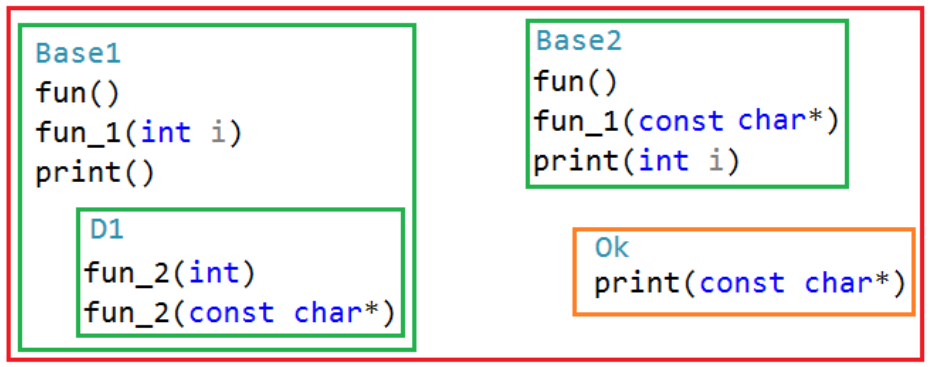
```
2
ok1的地址: 003FFD80
pb1指向的地址: 003FFD80
pd1指向的地址: 003FFD80
pb2指向的地址: 003FFD88
------------
2
pok3指向的地址: 003FFD80
2
base2地址:      003FFD38
pok4指向的地址:003FFD30
-858993460
请按任意键继续. . .
```

```
int main() {
    Ok ok1;
    cout << ok1.b.bi << endl;
    cout << "ok1的地址: " << &ok1 << endl;
    Base1 *pb1 = &ok1;
    cout << "pb1指向的地址: " << pb1 << endl;
    D1 *pd1 = &ok1;
    cout << "pd1指向的地址: " << pd1 << endl;
    Base2 *pb2 = &ok1;
    cout << "pb2指向的地址: " << pb2 << endl;
    cout << "------------\n";
    Ok * pok1 = static_cast<Ok*>(pb1);
    cout << pok1->b.bi << endl;
    //Ok * pok2 = dynamic_cast<Ok*>(pb1);
    //不行，dynamic_cast必须有虚函数
    Ok * pok3 = static_cast<Ok*>(pb2);
    cout << "pok3指向的地址: " << pok3 << endl;
    cout << pok3->b.bi << endl;
    Base2 base2;
    Ok * pok4 = static_cast<Ok*>(&base2);
    cout << "base2地址:      " <<&base2 << endl;
    cout << "pok4指向的地址:" << pok4 << endl;
    cout << pok4->b.bi << endl; //未定义,访问了未知内存
    return 0;
}
```

基类指针指向多继承派生类对象时：指向基类开始的地址

多继承：名字查找和类作用域

```
class Base1 {
public:
    void fun() { cout << "Base1::fun()\n"; }
    void fun_1(int i) { cout << "Base1::fun_1(int)\n"; }
    void print() { cout << "Base::print()\n"; }
};
class Base2 {
public:
    void fun() { cout << "Base2::fun()\n"; }
    void fun_1(const char*){cout<<"Base2::fun_1(const char*)\n";}
    void print(int i) { cout << "Base2::print(int)\n"; }
};
class D1 :public Base1 {
public:
    void fun_2(int) {}
    void fun_2(const char*) {}
};
class Ok :public D1, public Base2 {
public:
    void print(const char*){cout<<"Ok::Print(const char*)\n";}
};
```




```
int main() {
    Ok ok;
    //ok.fun(); //二义性错误
    //Ok的两个基类中：都有 fun()函数
    //ok.fun_1(1);    二义性错误
    //ok.fun_1("abc");二义性错误
    //虽然两个基类中的fun_1参数列表不同,但是不构成重载
    ok.fun_2(1);
    ok.fun_2("abc");
    //只有在同一个类中同名不同参数列表的才构成重载
    //ok.print();
    //ok.print(1); //基类中的print被派生类的shadow了。
    ok.print("abc");
    ok.Base2::print(1); //显式指明Base2中的print
    Base2 *pb = &ok; //基类指针指向派生类
    pb->fun();
    pb->fun_1("abc");
    pb->print(1);
    //pb->print("abc"); //错
    //pb只能调用其静态类型中有的成员。
    return 0;
}
```

```
Ok::Print(const char*)
Base2::print(int)
Base2::fun()
Base2::fun_1(const char*)
Base2::print(int)
请按任意键继续. . .
```

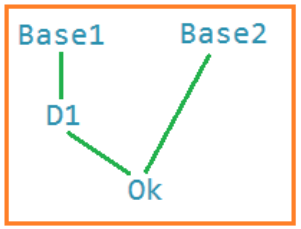
名字查找过程与单继承类似。
在多个基类中存在同名成员时，会产生二义性错误。

多继承：虚函数

```
struct A { int ai=1;};
struct B { int bi=2;};
class Base1 {
public:
    virtual void f0() { cout << "Base1::f0()\n"; }
    virtual void f1() { cout << "Base1::f1()\n"; }
    virtual void f2() { cout << "Base1::f2()\n"; }
    int i=3;
};

class Base2 {
public:
    virtual void f0() { cout << "Base2::f0()\n"; }
    virtual void f2() { cout << "Base2::f2()\n"; }
    virtual void f3() { cout << "Base2::f3()\n"; }
    int j=4;
};

class D1 :public Base1 {
public:
    virtual void f2()override { cout << "D1::f2()\n";}
    virtual void f4() { cout << "D1::f4()\n"; }
    A a;
};

class Ok :public D1, public Base2 {
public:
    virtual void f2()override { cout << "Ok::f2()\n";}
    virtual void f5() { cout << "Ok::f5()\n"; }
    B b;
};
```

```
int main() {
    Ok ok;
    Base1 *pb1 = &ok;
    pb1->f0(); //Base1::f0()
    pb1->f1(); //Base1::f1()
    pb1->f2(); //Ok::f2()
    Base2 *pb2 = &ok;
    pb2->f0(); //Base2::f0()
    pb2->f2(); //Ok::f2()
    pb2->f3(); //Base2::f3()
    D1 *pd1 = &ok;
    pd1->f0(); //Base1::f0()
    pd1->f1(); //Base1::f1()
    pd1->f2(); //Ok::f2()
    pd1->f4(); //D1::f4()

    //有虚函数,可以使用dynamic_cast
    Ok *pok1 = dynamic_cast<Ok*>(pb1);
    Ok *pok2 = dynamic_cast<Ok*>(pb2);
    cout << pok1 << " " << pok2 << endl;
    //dynamic_cast有安全保证
    Base2 base2;
    cout << "base2: " << &base2 << endl;
    Ok *pok3 = dynamic_cast<Ok*>(&base2);
    cout << pok3 << endl; // 0 说明转换失败
    Ok *pok4 = static_cast<Ok*>(&base2);
    cout << pok4 << endl; //非0,不安全
    cout << sizeof(Ok) << endl;
    //从原来的16变为24(32位编译)
    //多了8字节,猜测是2个虚函数表指针
    cout << "pb1: " << pb1 << endl;
    cout << "pb2: " << pb2 << endl;
    //地址差从8变为12(多了个虚函数表指针)
    return 0;
}
```

继承关系：




```
Base1::f0()
Base1::f1()
Ok::f2()
Base2::f0()
Ok::f2()
Base2::f3()
Base1::f0()
Base1::f1()
Ok::f2()
D1::f4()
0035F778 0035F778
base2: 0035F72C
00000000
0035F720
24
pb1: 0035F778
pb2: 0035F784
请按任意键继续. .
```

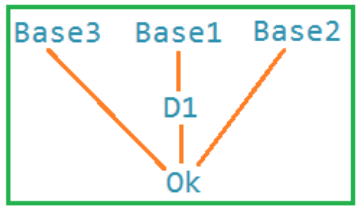
虚函数的调用与单继承类似。

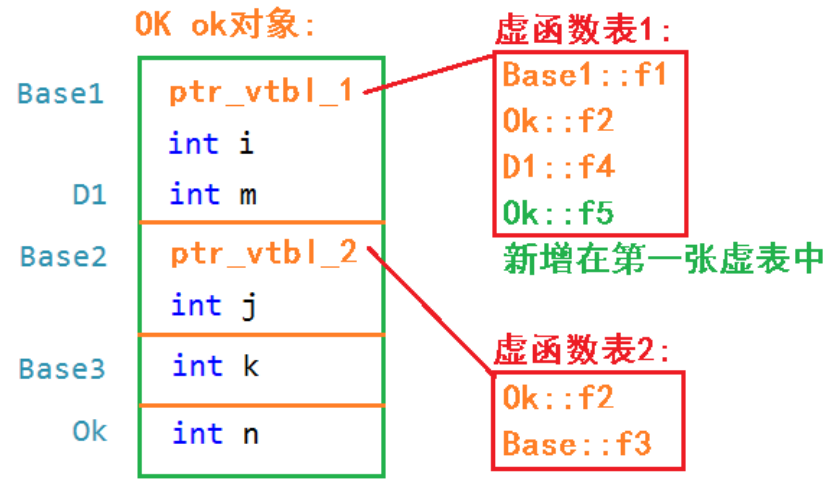
多继承：虚函数表

```
class Base1 {
public:
    Base1() { cout << "Base1\n"; }
    virtual void f1() {}
    virtual void f2() {}
    int i = 1;
};
class Base2 {
public:
    Base2() { cout << "Base2\n"; }
    virtual void f2() {}
    virtual void f3() {}
    int j = 2;
};
class Base3 { //没有虚函数
public:
    Base3() { cout << "Base3\n"; }
    void fun(){}
    int k = 3;
};
class D1 :public Base1 {
public:
    D1() { cout << "D1\n"; }
    virtual void f2()override {}
    virtual void f4() {}
    int m = 4;
};
```

```
class Ok :public Base3,public D1,public Base2{
public:
    Ok() { cout << "Ok\n"; }
    virtual void f2()override {}
    virtual void f5() {}
    int n = 5;
};
int main() {
    Ok ok;
    cout << "&ok: " << &ok << endl;
    cout << &ok.i << " " << &ok.j << " "<<&ok.k
        << " " << &ok.m << " " << &ok.n <<endl;
    Base1 *pb1 = &ok;
    Base2 *pb2 = &ok;
    Base3 *pb3 = &ok;
    cout << "pb1: " << pb1 << endl;
    cout << "pb2: " << pb2 << endl;
    cout << "pb3: " << pb3 << endl;
    cout << sizeof(ok) << endl;
    return 0;
}
```

```
Base3
Base1
D1
Base2
Ok
&ok: 0021FA30
0021FA34 0021FA40 0021FA44 0021FA38 0021FA48
pb1: 0021FA30
pb2: 0021FA3C
pb3: 0021FA44
28
请按任意键继续. . .
```





1. 多继承派生类对象内存布局:先放有虚函数的基类，接着放没有虚函数的基类，然后是派生类自己的数据成员。

2. 多继承下，有多少个有虚函数的基类就有多少个虚函数表指针

3. 派生类中新增的虚函数，添加在第一张虚函数表中。

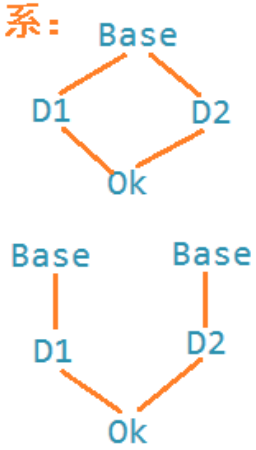
虚继承：引入

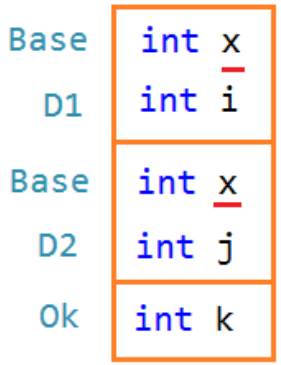
```
class Base {
public:
    int x = 0;
};
class D1 :public Base {
public:
    void set_x(int i) { x = i; }
    int i = 1;
};
class D2 :public Base {
public:
    int get_x()const {return x;}
    int j = 2;
};
class Ok :public D1, public D2 {
public:
    int k = 3;
};
int main() {
    Ok ok;
    ok.set_x(100);
    cout << ok.get_x() << endl;
    return 0;
}
```

0
请按

Base::x 出现了2次

继承关系：





加上虚继承：

```
class D1 :virtual public Base {···}
class D2 :virtual public Base {···}
int main() {
    Ok ok;
    ok.set_x(100);
    cout << ok.get_x() << endl;
    return 0;
}
```

100
请按任

在继承体系中，派生类有可能通过直接或间接的方式多次继承同一个基类，造成多份同名成员。虽然有时是必要的，但多数情况下我们不希望如此。（浪费空间，访问困难，造成歧义）

虚继承：class D1:virtual public Base
虚继承令某个类做出声明，承诺愿意共享它的基类。
其中，共享的基类称为虚基类。
此时：无论虚基类在继承体系中出现多少次，在派生类对象中都只包含一次虚基类。

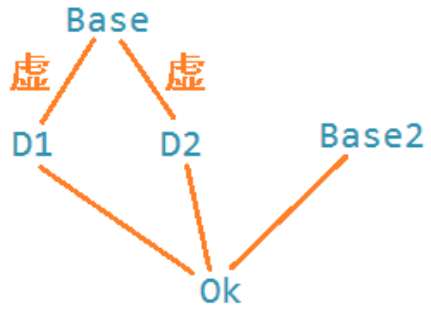
虚基类，需要设计与抽象；虚继承，是一种继承扩展

虚继承：构造与析构

```
class Base2 {
public:
    Base2() { cout << "Base2\n"; }
};
class Base {
public:
    Base(int i):x(i) { cout << "Base\n"; }
    int x;
};
class D1 :virtual public Base {
public:
    D1(int ii):Base(ii),i(ii) { cout << "D1\n"; }
    int i;
};
class D2 :virtual public Base {
public:
    D2(int jj):Base(jj),j(jj) { cout << "D2\n"; }
    int j;
};
class Ok :public D1, public D2,public Base2 {
public:
    Ok(int kk):D1(1),D2(2),Base(100),Base2() {
        cout << "Ok\n";
    }
    int k;
};
```

```
int main() {
    Ok ok(1);
    cout << ok.x << endl; //100
    return 0;
}
```

```
Base
D1
D2
Base2
Ok
100
请按任意
```

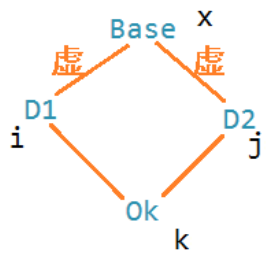


构造次序：首先按照虚基类的声明顺序调用虚基类的构造函数。（并以此为准）

然后按照非虚基类的声明顺序调用非虚基类的构造函数。

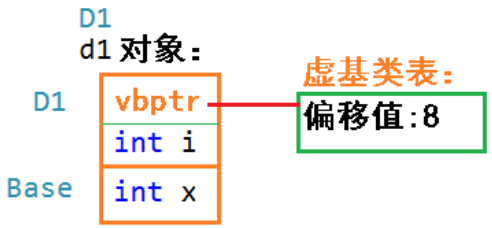
虚继承：对象内存分布

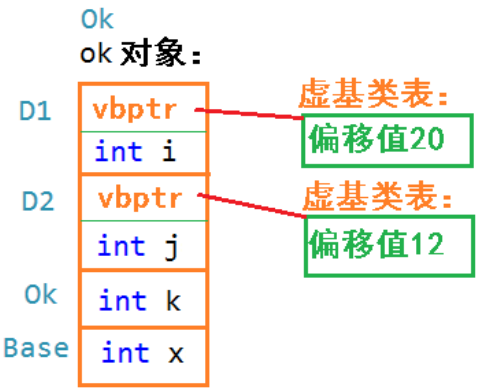
```
class Base {
public:
    int x=1;
};
class D1 :virtual public Base {
public:
    int i=2;
};
class D2 :virtual public Base {
public:
    int j=3;
};
class Ok :public D1, public D2 {
public:
    int k=4;
};
```




```
int main() {
    cout << "Base尺寸: " << sizeof(Base) << endl; //4
    cout << "  D1尺寸: " << sizeof(D1) << endl;   //12
    cout << "  D2尺寸: " << sizeof(D2) << endl;   //12
    cout << "  Ok尺寸: " << sizeof(Ok) << endl;   //24
    cout << "----------------\n";
    D1 d1;
    cout << "d1地址: " << &d1 << endl;
    cout << " i地址: " << &d1.i << endl;
    cout << " x地址: " << &d1.x << endl;
    //观察d1的最前面4个字节，指向虚基类表的指针 vbptr
    int *p = (int*)(&d1);
    int *p1 = (int*)(*p);
    cout << p1[0] << endl;
    cout << p1[1] << endl; //8 (偏移量)
    cout << "----------------\n";
    Ok ok;
    cout << "  ok地址: " << &ok << endl;
    Base *pb = &ok;
    cout << "Base地址: " << pb << endl;
    D1 *pd1 = &ok;
    cout << "  D1地址: " << pd1 << endl;
    D2 *pd2 = &ok;
    cout << "  D2地址: " << pd2 << endl;
    cout << " x地址: " << &ok.x << endl;
    cout << " i地址: " << &ok.i << endl;
    cout << " j地址: " << &ok.j << endl;
    cout << " k地址: " << &ok.k << endl;
    return 0;
}
```

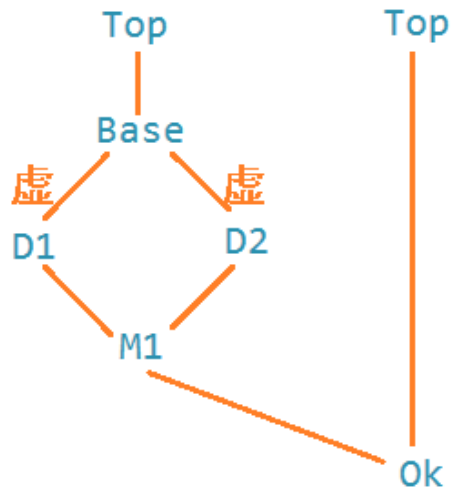
```
Base尺寸: 4
  D1尺寸: 12
  D2尺寸: 12
  Ok尺寸: 24
----------------
d1地址: 003CFA64
 i地址: 003CFA68
 x地址: 003CFA6C
0
8
----------------
  ok地址: 003CFA2C
Base地址: 003CFA40
  D1地址: 003CFA2C
  D2地址: 003CFA34
 x地址: 003CFA40
 i地址: 003CFA30
 j地址: 003CFA38
 k地址: 003CFA3C
请按任意键继续. . .
```

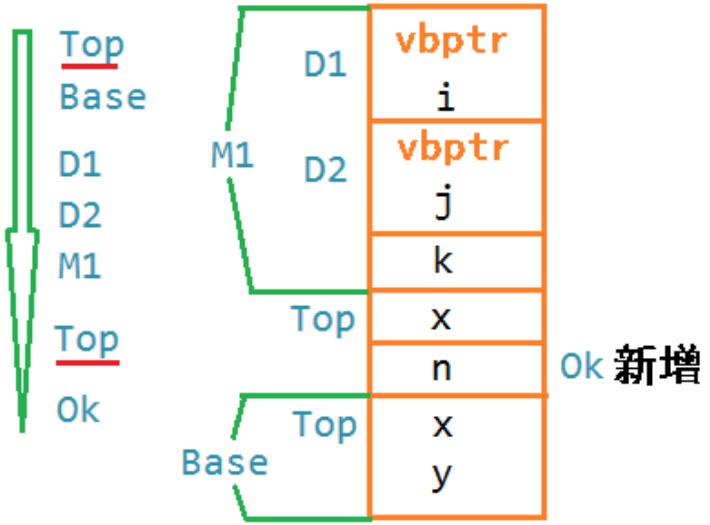




练习

```
struct Top { int x = 1; };
struct Base : public Top { int y = 2; };
struct D1 :virtual public Base { int i = 3; };
struct D2 :virtual public Base { int j = 4; };
struct M1 :public D1, public D2 { int k = 5; };
struct Ok :public M1, public Top { int n = 6; };

int main() {
    Ok ok;
    //1.写出构造次序
    M1 *p_m1 = &ok;
    D1 *p_d1 = &ok;
    D2 *p_d2 = &ok;
    Base *p_base = &ok;
    //Top *p_top = &d;
    //2.上面为什么不行?
    //(有两个Top,二义性)
    //3.找一个编译器,尝试画出该情况下的对象内存图

    return 0;
}
```





多继承、虚继承：小结

1. 在派生类对象中，同名的虚基类只产生一个虚基类子对象，而同名的非虚基类则各产生一个非虚基类子对象。
2. 虚基类的子对象是由最后派生出来的类的构造函数通过调用虚基类的构造函数来初始化的。
3. 虚基类并不是在声明基类时声明的，而是在声明派生类时通过指定其继承该基类的方式来声明的。

虚继承、虚基类 ←没有关系→ 虚函数、纯虚函数、抽象基类

虚继承是解决C++多重继承问题的一种手段，从不同途径继承来的同一基类，会在子类中存在多份拷贝。
存在问题：浪费存储空间；二义性问题（多份拷贝的基类无法用指针或引用操作派生类对象）
虚继承底层实现与编译器相关，一般通过虚基类表指针和虚基类表实现，每个虚继承的派生类都有一个虚基类表指针和虚基类表（不占用类对象的存储空间）（虚基类依旧会在派生类里面存在拷贝，只是仅仅最多存在一份而已）；当虚继承的派生类被当做基类继承时，虚基类表指针也会被继承。

虚继承对比虚函数的实现原理：相似之处：都利用了虚指针（占用类对象的存储空间）和虚表（不占用类对象的存储空间）。
虚基类仍然存在继承类中，要占用对象存储空间；虚函数不占用对象存储空间。
虚基类表存储的是虚基类相对直接继承类的偏移；而虚函数表存储的是虚函数地址。

异常：C语言中的错误处理

C语言中的错误处理，一般采用返回值或者全局变量errno，导致处理错误代码过多。

异常：
除0错误；
打开的文件不存在；
网络连接失败；

BUG：
野指针引用；

```c
int my_fun1(int i) {
    if (i >= 0) {
        //do something...
        return 1;
    }
    //do something...
    return 0;
}

char* my_malloc1(int size) {
    char* p = (char*)malloc(size);
    return p;
}

int error_no = 1;
double triangle_area1(double a, double b, double c) {
    if (!(a + b > c && a + c > b && b + c > a)) {
        error_no = 0;
        return 0.0;
    }
    double p = (a + b + c) / 2;
    double area = sqrt(p*(p - a)*(p - b)*(p - c));
    error_no = 1;
    return area;
}
```

```c
int main() {
    //1.利用返回值来确定是否正确执行
    int ret = my_fun1(10);
    if (0 == ret) {
        //失败了，处理代码...
    }
    else {
        //成功了，处理代码...
    }
    //2.返回值用于判断是否成功，也有其本身作用
    char* pc1 = my_malloc1(100);
    if (!pc1) {
        //没成功，处理代码...
    }
    //成功了，继续执行...
    //3.用全局变量 error_no来判断错误
    double s1 = triangle_area1(3, 4, 5);
    if (0 == error_no)
        printf("三角形三边长度关系不正确！\n");
    else
        printf("面积=%f\n", s1);
    s1 = triangle_area1(2, 2, 4);
    if (0 == error_no)
        printf("三角形三边长度关系不正确！\n");
    else
        printf("面积=%f\n", s1);
    return 0;
}
```

异常：C++中异常处理

1.把可能发生异常的语句放在 try 语句中;
2.若未发生异常，catch 子语句不起作用，程序流转到 catch 子句的后面执行;
3.程序运行中发生异常，则通过 throw 抛出异常。若本层没有catch到该异常，则立即离开本函数，转到上一级函数。
4.在catch子语句中按次序匹配，匹配成功后执行catch子语句中的代码，然后跳出try-catch，继续执行后面的代码，若无匹配，则继续转到再上一级函数，若一直到顶也没有匹配的，则系统调用 terminate 终止程序。
5.throw 抛出数据的类型：既可以是基本数据类型，也可以是类类型。

try { 语句1; 语句2; … }
catch (类型1 参数名){ //… }
catch (类型2 参数名){ //… }　**注意：C++没有 finally{} 语句,使用RAII(资源分配即初始化)机制。**

```
double triangle_area(double a, double b, double c) {
    if (a + b > c && a + c > b && b + c > a) {
        double p = (a + b + c) / 2;
        double area = sqrt(p*(p - a)*(p - b)*(p - c));
        return area;
    }
    throw - 1;
    cout << "triangle_area(): 错误\n"; //永远执行不到
}
```

```
int main() {
    double s1 = 0.0;
    try {
        s1 = triangle_area(2, 2, 4);
    }
    catch (int e) {
        cout << "Main: " << e << endl;
    }
    cout << "Main: end\n";
    return 0;
}
```

```
Main: -1
Main: end
请按任意键
```

异常：栈展开

抛出异常后，程序立即开始寻找与异常匹配的catch子句。当throw出现在try语句块内，检查与该try块关联的catch子句，如匹配，则使用之处理异常。否则检查外层try块，若没有，则退出当前函数并在上层调用函数中继续查找。栈展开的过程沿着嵌套函数的调用链不断查找，直到找到匹配的catch，若一直无法找到，则调用标准库函数terminate，程序终止执行。

```
void f4() {
    cout << "f4():1\n";
    throw 3.14;
    throw "abc";
    throw false;
    throw Err();
    throw (short)1;
    cout << "f4():2\n"; //不执行
}
void f3() {
    cout << "f3():1\n";
    try {
        try {
            f4();
        }
        catch (int e) {
            cout << "f3(): err_int\n";
        }
        catch (const char* e) {
            cout << "f3(): err_char*\n";
        }
    }
    catch (short e) {
        cout << "f3(): err_short\n";
    }
    cout << "f3():2\n";
}
```

```
void f2() {
    cout << "f2():1\n";
    f3();
    cout << "f2():2\n";
}
void f1() {
    cout << "f1():1\n";
    try {
        f2();
    }
    catch (double e) {
        cout << "f1(): err_double\n";
    }
    catch (Err e) {
        cout << "f1(): err_Err\n";
    }
    cout << "f1():2\n";
}
```

```
int main() {
    f1();
    try {
        f4();
    }
    catch (...) {
        cout << "main: err_default\n";
    }
    return 0;
}
```

throw 3.14;

```
f1():1
f2():1
f3():1
f4():1
f1(): err_double
f1():2
f4():1
main: err_default
请按任意键继续. . .
```

throw "abc";

```
f1():1
f2():1
f3():1
f4():1
f3(): err_char*
f3():2
f2():2
f1():2
f4():1
main: err_default
请按任意键继续. . .
```

throw false;

```
f1():1
f2():1
f3():1
f4():1
```

throw Err();

```
f1():1
f2():1
f3():1
f4():1
f1(): err_Err
f1():2
f4():1
main: err_default
请按任意键继续. . .
```

throw (short)1;

```
f1():1
f2():1
f3():1
f4():1
f3(): err_short
f3():2
f2():2
f1():2
f4():1
main: err_default
请按任意键继续. . .
```

异常：栈展开过程中对象被自动销毁

在栈展开（stack unwinding）的过程中，位于调用链上的语句块可能会提前退出。栈展开过程中退出了某个块，编译器将负责确保在这个块中创建的对象能正确地销毁（类对象一定会调用其析构函数）

```
void f2() {
    A a1;
    A a2;
    B* pb1 = new B;
    cout << "f2(): before throw..\n";
    throw 1;
    delete pb1; //不会执行到
}
void f1() {
    A a3;
    B* pb2 = new B;
    cout << "--------------\n";
    try {
        f2();
    }
    catch (double e) {
        cout << "f1(): err_double\n";
    }
    catch (const char* e) {
        cout << "f1(): err_char*\n";
    }
    cout << "f1():2\n";
    delete pb2; //这里有没有被执行到？
}
```

```
struct A {
    A() { cout << "A 构造\n"; }
    ~A() { cout << "A 析构\n"; }
};
struct B {
    B() { cout << "B---构造\n"; }
    ~B() { cout << "B---析构\n"; }
};

int main() {
    try {
        f1();
    }
    catch(int e){
        cout << "main(): err_int\n";
    }
    return 0;
}
```

```
A 构造
B---构造
--------------
A 构造
A 构造
B---构造
f2(): before throw.
A 析构
A 析构
A 析构
main(): err_int
请按任意键继续. . .
```

异常：catch匹配规则

异常抛出表达式会对异常对象进行拷贝初始化。
catch子句中的异常声明，有点像函数形参列表。参数类型可以是非引用类型(值或指针),也可以是引用类型。
注意：**不要抛出一个指向局部对象的指针**。
catch语句按其顺序逐一匹配，绝大多数类型转换不被允许：
允许从非常量类型向常量转换；允许从派生类向基类转换。所以：越是专门的catch越应该放在前面。

```
class A {
public:
    A() { cout << "A\n"; }
    ~A() { cout << "~A\n"; }
    /*explicit*/ A(const A&) { cout << "拷贝构造\n"; }
    A(A&&) { cout << "移动构造\n"; }
    int a = 100;
};
void f1() {
    try {
        A a;
        throw a; // 有 explicit 的拷贝构造就不行了
                 //上面throw a;抛出异常表达式,会执行拷贝构造(或移动构造)
                 //构造出来的异常对象,位于一个特殊空间内(编译器管理)
                 //异常处理完毕后，该对象销毁。
    }
    catch (A e) {
        cout << "f1():----------\n";
    }
}
```

```
void f2() {
    try {
        A a;
        throw &a; //错误！抛出了指向局部对象的指针
    }
    catch (A* pe) {
        //此处pe所指向的对象已经销毁,结果是未定义的
        cout << "f2():----------\n";
    }
}

void f3() {
    try {
        throw 1;
    }
    catch (double e) { //不会匹配(没有隐式转换)
        cout << "f3(double):----\n";
    }
    catch (int e) { //精确匹配
        cout << "f3(int):-------\n";
    }
}

int main() {
    f1(); f2(); f3(); return 0;
}
```

```
A
移动构造
拷贝构造
~A
f1():----------
~A
~A
A
~A
f2():----------
f3(int):-------
请按任意键继续.
```

异常：重新抛出异常

在单个catch语句不能完整处理某个异常时，可先处理部分，然后再抛出。
空throw语句只能出现在catch语句或catch语句调用的函数中，否则直接调用std::terminate()

```
struct Base { int b_data = 10; };
struct Derived:Base { int d_data = 20; };
void f1() {
    try {
        Derived d;
        throw d;
    }
    catch (Derived e) {
        e.b_data = 100; throw;
    }
}
void f2() {
    try {
        Derived d;
        throw d;
    }
    catch (Derived &e) {
        e.b_data = 100; throw;
    }
}
```

```
void f3() {
    try { Derived d; throw d; }
    catch (Base &e) {//会捕获到派生类类型的异常对象
        cout << "f3(Base):---\n";
        e.b_data = 100;
        //e.d_data = 200; //错,派生类向基类转换,截断
    }
    catch (Derived &e) { cout << "f1(Base):---\n"; }
    catch (...) { cout << "f1(...):---\n"; }
}

int main() {
    try { f1(); }
    catch (Derived e) {
        cout << e.b_data << endl; //还是10,没有改变
    }
    try { f2(); }
    catch (Derived &e) {
        cout << e.b_data << endl; //100,改变了
    }
    f3();
    return 0;
}
```

```
10
100
f3(Base):---
请按任意键继
```

异常：noexcept异常说明

明确告诉编译器某个函数不会抛异常，可以加上noexcept异常说明。(有助于编译器更好地优化代码)

noexcept运算符，判断表达式是不是会抛异常。(不会运算表达式)

```
void f1() noexcept { } //不会抛异常
void f2() noexcept(true){}//不会抛异常
void f3() noexcept(false) {}//可能抛异常
void f4() {} //可能抛异常
void f5() throw() {} //不会抛异常
void f6() throw(int,double){} //只会抛int和double类型的异常
void f7() noexcept { //不会抛异常
    throw 1; //实际上抛了,编译ok,实际调用std::terminate
}
void f8() noexcept {
    try { throw 1; }
    catch (int e) { cout << "异常处理！\n"; }
    //异常处理了，ok,没问题。
}
```

```
int main() {
    f8(); //正常运行
    //noexcept运算符：不会抛异常返回true,会抛异常返回false
    cout << noexcept(f1()) << endl; //1
    cout << noexcept(f2()) << endl; //1
    cout << noexcept(f3()) << endl; //0
    cout << noexcept(f4()) << endl; //0
    cout << noexcept(f5()) << endl; //1
    cout << noexcept(f6()) << endl; //0
    cout << noexcept(f7()) << endl; //1
    cout << noexcept(f8()) << endl; //1
    return 0;
}
```

```
异常处理!
1
1
0
0
1
0
1
1
请按任意键
```

异常：标准库异常类

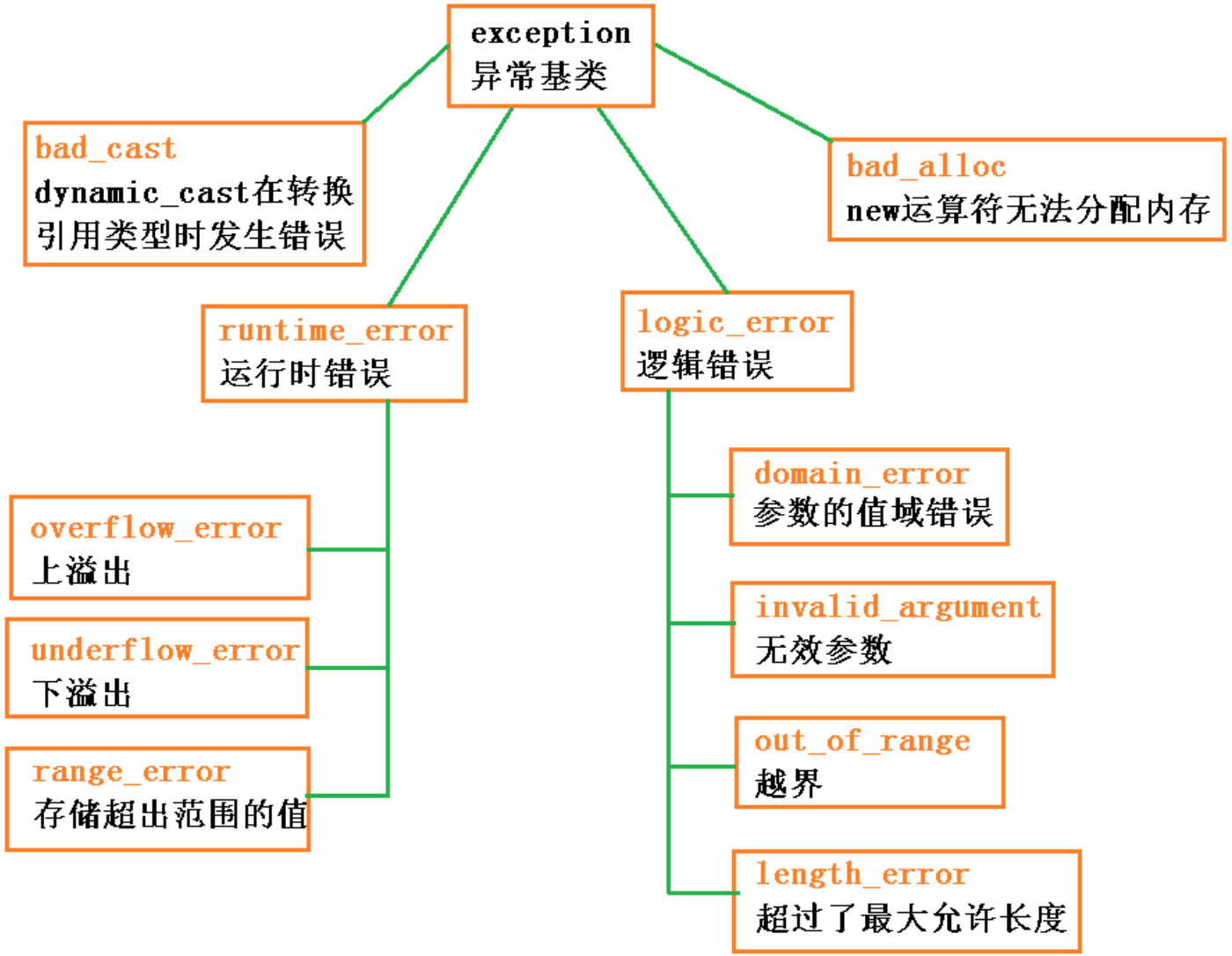
类exception定义了默认构造、拷贝构造、赋值重载、虚析构和一个what()的虚函数(返回const char*);

exception、bad_cast、bad_alloc有默认构造；runtime_error、logic_error没有默认构造，有接受字符串为参数的构造。

what()函数可以自己override。

除0，常规的下标运算符[]越界，c++都不会抛异常…

效率优先。



异常：标准库异常类

```
int main() {
    int a, b;
    cout << "输入2个整数: ";
    while (cin >> a >> b) {
        try {
            if (b == 0) throw runtime_error("divisor is 0");
            cout << static_cast<double>(a) / b << endl;
            cout << "输入2个整数: ";
        }
        catch (runtime_error err) {
            cout << err.what();
            cout << "\nTry Again? Enter y or n:" << endl;
            char c;
            cin >> c;
            if (!cin || c == 'n' || c == 'N')
                break;
            else
                cout << "输入2个整数: ";
        }
    }
    return 0;
}
```

```
输入2个整数: 20 0
divisor is 0
Try Again? Enter y or n:
y
输入2个整数: 1 2
0.5
输入2个整数: 3 0
divisor is 0
Try Again? Enter y or n:
n
请按任意键继续. . .
```

异常：自定义异常类

```
#include <iostream>
#include <cassert>
#include <vector>
using namespace std;
class MyException :public logic_error {
public:
    explicit MyException(int i,const char* s):logic_error(s),index(i){}
    int index;
};
class A {
public:
    A(int n) {
        //assert(n >= 0); 直接断言
        if (n < 0) throw invalid_argument("错！初始化n>=0");
        vec_data.resize(n, 100);
    }
    int & operator[](int i) {
        //assert(i >= 0 && i < vec_data.size()); 效率优先
        if (i < 0 || i >= vec_data.size())
            throw MyException(i, "越界操作！");
        return vec_data[i];
    }
private:
    vector<int> vec_data;
};
```

```
int main() {
    try {
        A a(-1);
    }
    catch (invalid_argument &e) {
        cout << e.what() << endl;
    }
    catch (MyException &e) {
        cout << e.what() << " index=" << e.index << endl;
    }

    try {
        A a(10);
        cout << a[11];
    }
    catch (invalid_argument &e) {
        cout << e.what() << endl;
    }
    catch (MyException &e) {
        cout << e.what() << " index=" << e.index << endl;
    }

    return 0;
}
```

```
错！初始化n>=0
越界操作！ index=11
请按任意键继续. . .
```

异常：析构函数与异常

析构函数不应该抛出不能被它自己处理的异常。(通常在析构函数后面加上noexcept表示不抛异常)

析构函数总是会被执行，通常用来释放资源。
1.假如在析构函数中出现不能被它自己处理的异常，则可能导致释放资源的代码没有运行。
2.栈展开的过程中，析构函数中有异常逃离的话，会造成同时有多个未处理的异常并存，程序会退出。

一般来说：析构函数只是释放资源，所以不太可能抛异常。标准库类型都确保它们的析构函数不抛异常。

```
class A {
public:
    A() { cout << "A构造\n"; }
    ~A() noexcept(false) {  //允许它抛异常
        throw 1; cout << "A析构\n";
    } //析构函数中有异常逃离
};
void f() {
    A a1;
    A a2;
    throw 2;
}
```

有些编译器会默认加上noexcept

```
int main() {
    try { f(); }
    catch (...) { }
    return 0;
}
```

栈展开过程中，多个未处理的异常并存，会导致程序结束

异常：构造函数与异常

构造函数中的异常，导致对象创建失败，编译器会自动逐个析构对象中的数据成员。

```
struct A1 {
    A1() { cout << "A1()\n"; }
    ~A1() { cout << "~A1()\n"; }
};
struct A2 {
    A2() { cout << "A2()\n"; }
    ~A2() { cout << "~A2()\n"; }
};
struct A3 {
    A3() { throw 1; cout << "A3()\n"; }
    ~A3() { cout << "~A3()\n"; }
};

void* operator new(std::size_t size) {
    cout << "malloc memory!\n";
    void* p = malloc(size);
    if (p) return p;
    throw bad_alloc();
}
void operator delete(void* p) {
    cout << "free memory!\n";
    free(p);
}
```

```
class Object {
public:
    Object() { cout << "Object()\n"; }
    ~Object() { cout << "~Object()\n"; }
private:
    A1 a1; A2 a2; A3 a3;
};

int main() {
    try { //obj构造失败,异常发生在构造过程中
        Object obj;
    }
    catch (...) { cout << "error\n"; }
    cout << "------------\n";
    try {
        Object* pobj = new Object;
    }
    catch (...) { cout << "error\n"; }
    return 0;
}
```

```
A1()
A2()
~A2()
~A1()
error
------------
malloc memory!
A1()
A2()
~A2()
~A1()
free memory!
error
请按任意键继续
```

异常：构造函数与异常

```
struct A1 {
    A1() { cout << "A1()\n"; }
    ~A1() { cout << "~A1()\n"; }
};
struct A2 {
    A2() { cout << "A2()\n"; }
    ~A2() { cout << "~A2()\n"; }
};
struct A3 {
    A3() { throw 1; cout << "A3()\n"; }
    ~A3() { cout << "~A3()\n"; }
};

class Object {
public:
    Object() :a2(nullptr),a3(nullptr){
        cout << "Object()\n";
        a2 = new A2;
        a3 = new A3;
    }
    ~Object() {
        cout << "~Object()\n";
        delete a2; delete a3;
    }

private:
    A1 a1; A2 *a2;  A3 *a3;
};
```

```
int main() {
    try {
        Object obj;
    }
    catch(...){cout << "error\n";}
    return 0;
}
```

~A2()没调用到！
a2指向的内存泄漏

```
A1()
Object()
A2()
~A1()
error
请按任意
```

```
Object() :a2(nullptr),a3(nullptr){
    cout << "Object()\n";
    try {
        a2 = new A2;
        a3 = new A3;
    }
    catch (...) {
        delete a2;
        delete a3;
        throw;
        //确保构造失败时，不会泄露内存
    }
}
```

```
A1()
Object()
A2()
~A2()
~A1()
error
请按任意
```

异常：构造函数与异常

构造函数的初始值列表，也有可能会抛异常，可以使用函数try语句块。

带初始值列表的
构造函数:

```
Object ::Object() try
      :a2(new A2()) {
      //somecode...
}
catch (...) {
    //somecode...
}
```

```
class Object {
    ...
private:
    A1 *a1;
    A2 *a2;
    A3 *a3;
};
```

```
Object::Object() try :a1(new A1()), a2(new A2()), a3(new A3()) {
    //somecode may be throw...
}   这样写法会有什么问题？
catch (...) { delete a1; delete a2; delete a3; throw; }
```

```
Object::Object() try :a1(cr_A1()), a2(cr_A2()), a3(cr_A3()) {
    //somecode may be throw...
}   cr_A1, cr_A2, cr_A3要怎么处理？
catch(...){ /*so smoething*/ throw; }
```

```
Object::Object() :a1(nullptr), a2(nullptr), a3(nullptr) {
    try {
        a1 = new A1(); a2 = new A2(); a3 = new A3();
        //somecode may be throw...
    }
    catch (...) { delete a1; delete a2; delete a3; throw; }
}
```

改用智能指针:

```
class Object {
    ...
private:
    shared_ptr<A1> a1;
    shared_ptr<A2> a2;
    shared_ptr<A3> a3;
};
```

```
Object::Object() try
    :a1(make_shared<A1>()), a2(new A2()), a3(new A3()){
    //somecode may be throw...
}  不需要手工处理 delete a1, a2, a3
catch (...) { /*do something...*/ throw; }
```

异常：异常安全1

异常安全函数（异常安全代码）：

1. 如果异常抛出，不会泄露资源。
2. 如果异常抛出，程序状态不改变。如果函数成功，就是完全成功；如果函数失败，程序回复到“调用函数之前”的状态。

```
#include <iostream>
#include <memory>
#include <mutex>
#include <thread>
using namespace std;
class Image {
public:
    Image(int i) :data(i) {
        if (i % 4 == 1) throw "Cr_Img error";
    }   //1,5,9...抛异常
private:
    int data;
};
class ImgMenu {
public:
    ImgMenu(int i = 0)
        :img(new Image(i)),img_changes(0) {}
    ~ImgMenu() { delete img; }
    void set_img(int image); //更改图片
private:
    std::mutex mtx; //互斥体
    Image *img;
    int img_changes; //图片变换的次数
};
```

```
void ImgMenu::set_img(int image) {
    mtx.lock();  //加锁,申请资源
    delete img; img = nullptr;
    img_changes++;
    img = new Image(image);
    cout << img_changes << endl;
    mtx.unlock(); //解锁,释放资源
}
```

有什么问题？

```
ImgMenu img_menu;
void do_thread(int image) { //线程函数
    try {
        img_menu.set_img(image);
    }
    catch (const char * e) {
        cout << e << endl;
    }
}
int main() {
    std::thread threads[8];
    for (int i = 0; i < 8; i++)
      threads[i]=std::thread(do_thread,i);
    for (auto &th : threads) th.join();
    return 0;
}
```

异常：异常安全2

```
class My_lock_guard {
public:
    explicit My_lock_guard(std::mutex& _mtx):mtx(_mtx){
        mtx.lock(); //构造完成时,资源已经可以使用
    }
    ~My_lock_guard() { mtx.unlock(); } //析构时释放
private:
    std::mutex& mtx; //引用
};
```

```
class Image {
public:
    Image(int i) :data(i) {
        if (i % 4 == 1) throw "Cr_Img error";
};   ...
```

```
class ImgMenu {
public:
    ...
    void set_img(int image); //更改图片
private:
    std::mutex mtx; //互斥体
    Image *img;
};  int img_changes; //图片变换的次数
```

```
void ImgMenu::set_img(int image) {
    //std::lock_guard<std::mutex> lck(mtx);
    My_lock_guard lck(mtx); //代替了1.2
    //1. mtx.lock();  //加锁,申请资源
    delete img; img = nullptr;
    img_changes++;
    img = new Image(image);
    cout << img_changes << endl;
    //2. mtx.unlock(); //解锁,释放资源
}
```

通过RAII，用对象来管理资源。
或者通过智能指针来管理资源。

可以有效避免资源的泄漏。

异常：异常安全3

如果异常抛出，程序状态不改变。数据一致性，数据完整性

如果函数成功，就是完全成功；如果函数失败，程序回复到“调用函数之前”的状态。

```
class ImgMenu {
public:
    ImgMenu(int i = 0)
        :img(new Image(i)),img_changes(0) {}
    ~ImgMenu() { /*delete img;*/ }
    void set_img(int image); //更改图片
private:
    std::mutex mtx;
    //Image *img;
    shared_ptr<Image> img; //改用智能指针
    int img_changes;
};
```

```
void ImgMenu::set_img(int image) {
    My_lock_guard lck(mtx);

    //delete img;    img = nullptr;
    //img_changes++;
    //img = new Image(image);

    //new成功才会执行 reset
    //失败的话,不会delete原来的img内容
    img.reset(new Image(image));
    img_changes++;

    cout << img_changes << endl;
}
```

异常：异常安全4

如果异常抛出，程序状态不改变。

常用方法：copy & swap，先复制一个副本，在副本上修改，成功则与副本交换，不成功没有影响。

当然，这样的做法有些时候代价较大，选择合适的写法保证异常安全。

```
struct ImgMenu_Impl {
    ImgMenu_Impl(int i=0)
        :img(make_shared<Image>(i)),img_changes(0){}
    shared_ptr<Image> img;
    int img_changes;
};
class ImgMenu {
public:
    ImgMenu(int i=0):pImpl(make_shared<ImgMenu_Impl>(i)){}
    void set_img(int image); //更改图片
private:
    std::mutex mtx;
    shared_ptr<ImgMenu_Impl> pImpl;
};
```

```
void ImgMenu::set_img(int image) {
    My_lock_guard lck(mtx);
    //1.复制一个副本
    shared_ptr<ImgMenu_Impl> pNew(new ImgMenu_Impl(*pImpl));
    //2.修改副本
    pNew->img.reset(new Image(image));
    pNew->img_changes++;
    //3.和副本交换
    std::swap(pImpl, pNew);

    cout << pImpl->img_changes << endl;
}
```

异常：练习

```
#include <iostream>
#include <memory>

using namespace std;
class A {
public:
    A(int i):p_data(new int(i)){
        if (i % 2) throw "A_Error";
    } //奇数初始化会抛异常
    int get_data()const {return *p_data;}
    void swap(A& rhs) {
        std::swap(p_data, rhs.p_data);
    }
private:
    unique_ptr<int> p_data;
};
```

```
class Test {
public:
    Test(int i, int j) :a1(i), a2(j) {}
    void print_a1_a2()const {
        cout << "a1=" <<a1.get_data()
            << " a2=" <<a2.get_data()<<endl;
    }
    void set_a1_a2(int i, int j) {
        A tmp_a1(i);
        tmp_a1.swap(a1);
        A tmp_a2(j);
        tmp_a2.swap(a2);
    }
private:
    A a1;
    A a2;
};
```

有没有问题？
有的话如何修改？

```
int main() {
    Test t(2, 4);
    t.print_a1_a2();
    t.set_a1_a2(100, 200);
    t.print_a1_a2();
    cout << "----------\n";
    //当前是 100,200
    try {
        t.set_a1_a2(100, 201);
    }
    catch (const char * e) {
        cout << e << endl;
    }
    t.print_a1_a2();
    //仍然是100,200
    //异常安全，操作失败
    return 0;   没有改变原始值
}
```

异常：小结

1. C++的异常是有代价(记录工作、代码优化等)的，如果使用普通的处理方式(assert,return等)已经足够，就不要使用异常处理机制。频繁执行的关键代码段应该避免使用C++异常处理机制。当然，在你确认你的所有代码没有使用异常，可告知编译器以更好地优化代码。

2. 通过抛出对象，能够获取异常信息，可以清晰的展示出错误原因，给用户更好的提示。不像返回错误码那么模糊。

3. 把可能出现异常的代码和异常处理代码隔离开，结构更清晰。把内层错误的处理直接转移到适当的外层来处理，化简了处理流程。传统的手段是通过一层层返回错误代码把错误转移到上层，再转移到上上层，当层数过多时需要非常多的判断。

4. 出现异常时，通过栈展开自动销毁栈对象来确保栈对象析构函数会被正确调用。

5. 由于异常处理机制本身带来的开销，建议只在重要事件上使用异常处理，也就是不要滥用异常处理。

6. 使用异常处理机制，会打断执行流，函数有可能不在该返回的地方返回，这样使得代码的管理和调试变得困难。

7. 编写异常安全的代码通常要使用RAII和不同编码方式(如copy&swap)来实现，增加了编写难度。