#include <iostream>

#include <vector>

#include <tuple>

#include <bitset>

#include <regex>

#include <random>

using namespace std;

void exceptionTest();

int main()

{

    exceptionTest();

    cout << "enter key" << endl;

    while (cin.get() != EOF) // ctrl + z

    {

    }

    return 0;

}

void exceptionTest()

{

    try

    {

        int n = 0;

        if (n < 0)

            throw logic\_error("逻辑错误，n必须大于0");

    }

    catch (runtime\_error error) // 运行时错误

    {

    }

    catch (logic\_error error) // 逻辑错误，一般由程序员抛出

    {

        cout << error.what() << endl; // 打印错误消息

    }

    catch (exception error) // 错误

    {

    }

}

// 如果我们的函数不会抛出异常，那么我们应该指定 noexcept 或 noexcept(true)

// 如果指定我们的函数不会抛出异常，但函数却抛出了异常，那么该程序会终止

void exceptionfun1() noexcept {};

// 如果 exceptionfun1 不会抛出异常，那么 exceptionfun2 也承诺不会抛出异常

void exceptionfun2() noexcept(noexcept(exceptionfun1)) { exceptionfun1(); };

#ifndef Pack\_H

#define Pack\_H

#include <iostream>

#include <string>

// 指定命名空间 Sales

// 同一个命名空间可以定义在不同的文件

// 通常情况下，不将 #include 放在命名空间中

namespace Sales

{

class Pack

{

private:

    /\* data \*/

public:

};

void Fun1();

// 内联命名空间

inline namespace Sales\_Sub{

    void Fun2();

}

} // namespace Sale

// 命名空间外定义成员需要加命名空间前缀

void Sales::Fun1(){};

// 访问内联命名空间无需添加 Sales\_Sub 前缀

void Sales::Fun2(){};

// 头文件一般只在它的命名空间内或函数内使用 using 声明 或 指示

void Fun3(){

    using Sales::Fun1;          // using 声明

    using namespace Sales;      // using 指示

}

void Fun4(){

    // 考虑如下问题，operator>>属于std命名空间，但这里为什么不加前缀也可以？？

    // 函数匹配规则如下：

    // 1.在当前作用域中查找

    // 2.在当前所使用的命名空间查找

    // 3.如果参数是类类型，则在参数所在的命名空间中查找

    // 参数 s 定义在 string 文件的 std 命名空间中，而我们下面使用的 operator>> 也在 string 文件的 std 命名空间中，所有匹配成功

    std::string s = "aa";

    operator>>(std::cin, s);

}

#endif

#ifndef Programmer\_H

#define Programmer\_H

#include <string>

// 多继承

// 程序员

class Programmer

{

private:

    /\* data \*/

public:

};

// 前端

// 这里对子类使用虚继承，表示 FrontEnd 愿意共享 Programmer

// 虚基类在继承体系中只存在一个

// 如果这里不使用虚继承，那么 FrontEnd 将独自拥有其继承的 Programmer

class FrontEnd : public virtual Programmer

{

    public:

    std::string GetName(){ return "FrontEnd"; };

};

// 后端

class BackEnd : public virtual Programmer

{

    public:

    std::string GetName(){ return "BackEnd"; };

};

// 全栈

// 由于 FrontEnd，BackEnd 都使用虚继承 Programmer ，所以 Programmer 在 FullStack 的对象中只存在一份

class FullStack : public FrontEnd, public BackEnd

{

private:

public:

    // 显示初始化所有基类

    // 由于两个子类都虚继承 Programmer，所以 Programmer 的初始化由当前类执行，并且放在第一位，如果我们没有显示调用 Programmer，则会调用Programmer的默认构造函数

    FullStack() : Programmer(), FrontEnd(), BackEnd(){};

    // 基类都有 GetName 方法，所有为避免二义性，派生类应该也定义一个，隐藏基类的方法

    std::string GetName(){

        return FrontEnd::GetName() + "+" + BackEnd::GetName();

    };

};

#endif