C++ Programming

Exception Handling 异常处理

Zheng Guibin
(郑贵滨)

目录

目录

CONTENT

Exception Handling

异常处理

1 异常处理的基本思想

- 错误分类
  - 编译错
    - 编译时通不过，属语法错，最浅层次的错误。
  - 运行错
    - 调试时无法发现，运行时才出现，往往由系统环境引起，属可予料但不可避免。必须由语言的某种机制予以控制。
  - 逻辑错
    - 设计缺陷，编译器无法发现，只能靠人工分析跟踪排除，错误层次中等。

1 异常处理的基本思想

异常处理的指导思想：

- 对于可预料但不可避免的程序错误，不能眼睁睁看着它发生而无所作为（只会用exit()），要将消极等待变为积极预防，还要将预防的处理内容归纳整理，分门别类地作成类。
- 积极之意是，不能只凭编程经验这种个体性偶然性的做法，而要凭借人人都掌握的规范圆满的处理。
- 这规范是指在函数的处理中设下“陷阱”，一旦触发异常，定会被异常处理所收容，统一归口处理。

1 异常处理的基本思想

异常处理的三种方式

- 在出现异常时，直接调用abort() 或者exit();
- 通过函数的返回值来判断异常。

  但是如果一个函数有多个返回值的时候会比较麻烦，而且也会浪费不必要的判断返回值的时间开销；
- 通过try{ } catch() 的结构化异常处理来完成；

2 异常处理的实现机制

三个关键字：try throw catch

- 抛掷异常的程序段

```
……
throw 表达式;
……
```

- 捕获并处理异常的程序段

```
try
 {
     复合语句
}
catch（异常声明1）
异常处理复合语句
catch（异常声明2）
异常处理复合语句
  …
```

例1处理除零异常

```
#include <iostream>
using namespace std;
int Div(int x, int y) {
    if (y == 0)
        throw x;
    return x / y;
}
int main() {
    try {
        cout << "5 / 2 = " << Div (5, 2) << endl;
        cout << "8 / 0 = " << Div (8, 0) << endl;
        cout << "7 / 1 = " << Div (7, 1) << endl;
    } catch (int e) {
        cout << e << " is divided by zero!" << endl;
    }
    cout << "That is ok." << endl;
    return 0;
}
```

结果如下：

```
5 / 2 = 2
8 is divided by zero!
That is ok.
```

2 异常处理的实现机制

三个关键字作用：

- **try**设置了一个侦错范围，又叫保护段。其实是划定了一个跳跃的边界；
- **throw**负责抛掷异常对象；若放在函数声明中则又称为异常接口声明；
- **catch**负责处理捕获来的异常（包括继续抛掷异常）。

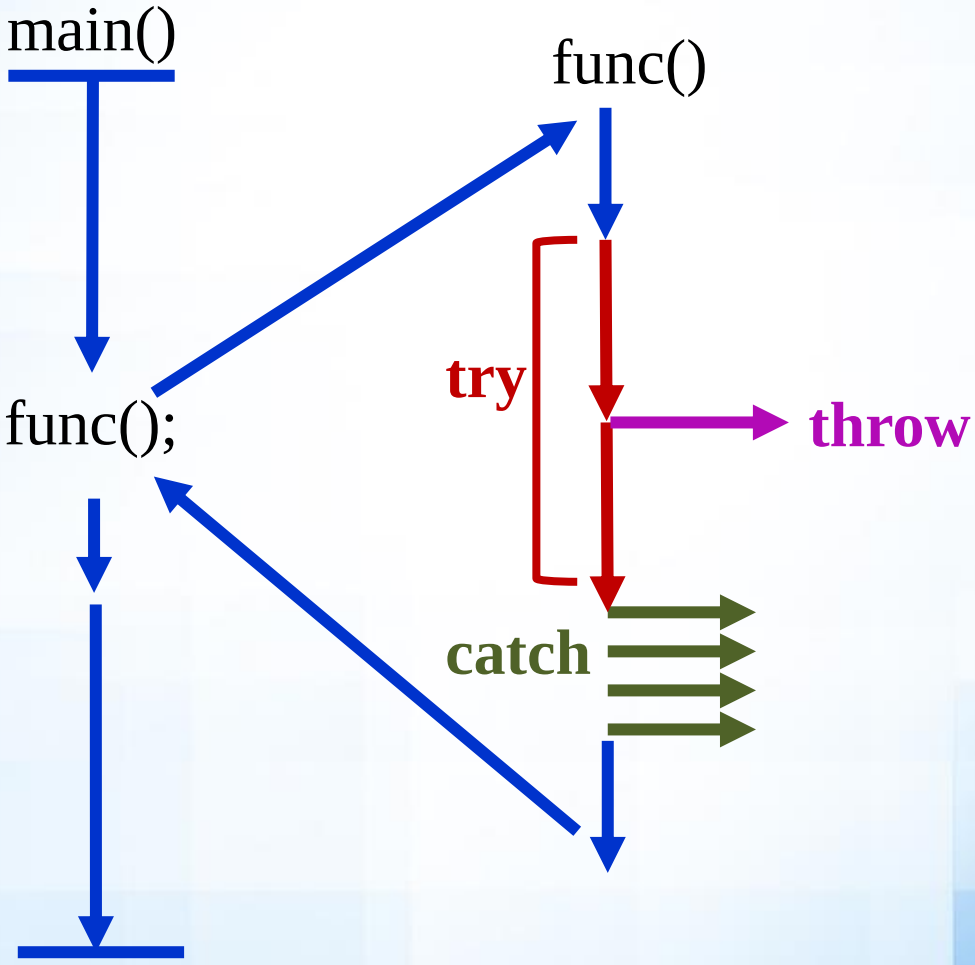
所抛掷的异常对象并非建在函数栈上，而是建在专用的异常栈上，故可以跨越函数而传递给上层。

2 异常处理的实现机制

- 每个catch()相当于一段函数代码；
- 每个throw则相当于一个函数调用；
- 每个try块至少跟一个catch()；
- 一个程序可设置个数不定的try 、throw 和catch。它们只有逻辑上的呼应，而无数量上的对应关系，且不受所在函数模块限制；
- 异常抛掷点往往距异常捕获点很远，它们可以不在同层模块中；
- 甚至有的throw我们看不到在哪，实际上在我们所调用的系统函数中，在标准库中；
- 程序中try块可以并列、可以嵌套；

2 异常处理的实现机制

- 异常处理的模式



2 异常处理的实现机制

异常处理的不唤醒机制

- 即一旦在保护段执行期间发生异常，则立即抛掷，由相应的catch子句捕获处理。抛掷——捕获间的代码被越过而不执行。程序从try块后跟随的最后一个catch子句后面的语句继续执行下去。
- 异常处理是将检测与处理分离，以便各司其职、灵活搭配地工作。它们的联系靠类型。

2 异常处理的实现机制

- 若有异常则通过throw操作创建一个异常对象并抛掷。
- 将可能抛出异常的程序段嵌在try块之中。控制通过正常的顺序执行到达try语句，然后执行try块内的保护段。
- 如果在保护段执行期间没有引起异常，那么跟在try块后的catch子句就不执行。程序从try块后跟随的最后一个catch子句后面的语句继续执行下去。
- catch子句按其在try块后出现的顺序被检查。匹配的catch子句将捕获并处理异常（或继续抛掷异常）。
- 如果匹配的处理器未找到，则运行库函数terminate将被自动调用，其缺省功能是调用abort终止程序。

2 异常处理的实现机制

```
try
{
…
}
catch(…)//catch all exception捕获所有异常;
{
   cout<<" Catch a excepton!"<<endl;
}
```

2 异常处理的实现机制

```
class AAA
{   class ERR_AAA{};
   public:
     void    throw_ex(){throw ERR_AAA();}
     void  show_msg();
};
void AAA::show_msg()
{
try{cout<<"msg from AAA"<<endl; throw_ex();}
    catch( ERR_AAA){    }
    catch(…){}
}
```

2 异常处理的实现机制——异常接口声明

- 可以在函数的声明中列出这个函数可能抛掷的所有异常类型。
  - 例如：

    void fun() throw(A，B，C，D);
- 若无异常接口声明，则此函数可以抛掷任何类型的异常。
- 不抛掷任何类型异常的函数声明如下：

  void fun() throw();

2 异常处理的实现机制

C++的异常处理机制有何特色？

- C对出错的处理是将所有错误予以编码(返回码)，一码一错。程序员编程时，通过在主调函数中检查出现的返回码，给与相应的处理。这种方法繁琐又不规范。
- C++则是将形形色色的错误归结抽象为“类型错”，统统交给throw抛掷；将出错处理统一为catch 调用。这就为程序员编程制定了统一的规范，也就是运行协议。
- “异常处理”实际是个动态概念。其处理机制是靠类型匹配，这恰是运行时才能发生的事。

2 异常处理的实现机制

C对出错处理的返回码机制，与C++的异常处理机制有何不同？

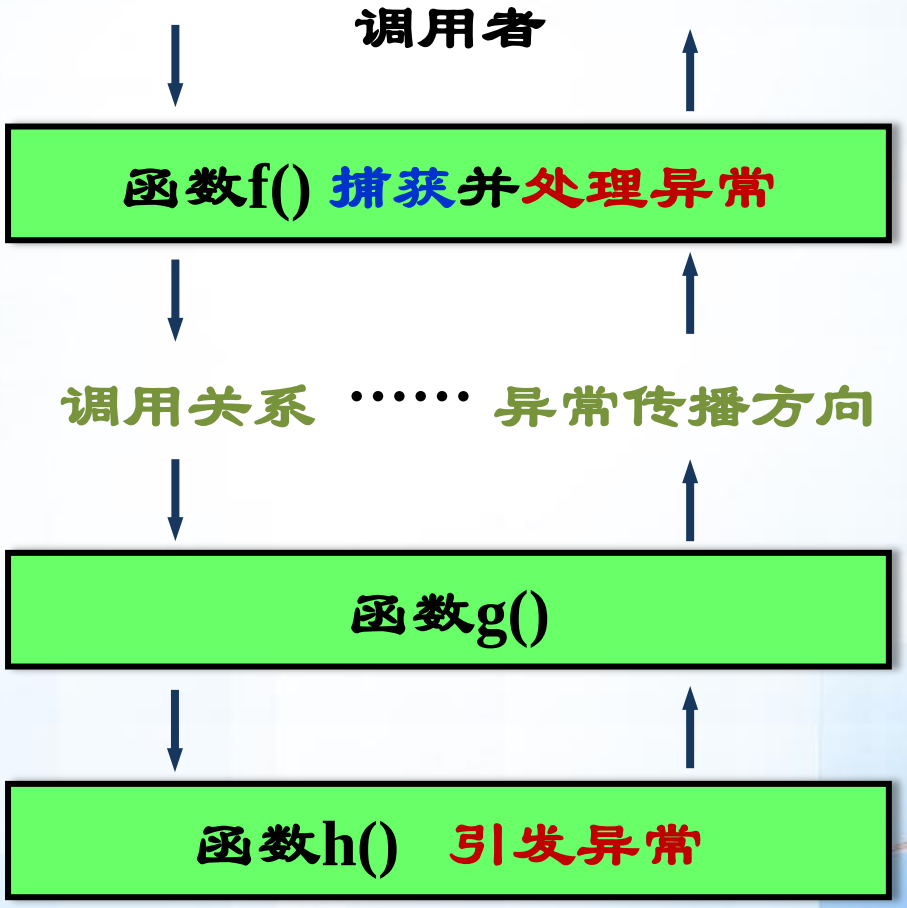
- 1、将正常逻辑与异常逻辑混在一起。每个函数调用后都要用正常逻辑（如if）加以判断，以过滤出异常。这样使子函数的错误扩散到了其所有的调用者中，形成了一个混杂的怪异链。
- 2、增加了用于处理很少出现的特殊情况的运行费用。尤其对于C++，其成员函数通常只有几行代码，但函数很多，若每个函数都使用返回码技术，则臃肿不堪。
- 3、类中还有构造、拷贝构造等无返回值的函数，它们出错处理的权力不应被剥夺。
- 4、返回码靠预定的数字代号来传递信息，所能携带的信息量少且呆板，远少于抛掷的对象所携带的信息量。

例2 异常嵌套处理

```
int Div(int x,int y)
{
    if(y==0) throw y;
    return x/y;
}
int Cal(int x,int y)
{
    return Div(x,y);
}
void main()
{
     try
     {
          cout<<"5/2="<<Cal(5,2)<<endl;
          cout<<"8/0="<<Cal(8,0)<<endl;
          cout<<"7/1="<<Cal(7,1)<<endl;
     }
     catch( int ){ cout<<"except of deviding zero.\n";}
     cout<<"that is ok.\n";
}
```

2 异常处理的实现机制

- 异常嵌套处理的基本思想



3 异常处理中的构造与析构

- 找到一个匹配的catch异常处理后
  - 初始化异常参数。
  - 将从对应的try块开始到异常被抛掷处之间构造（且尚未析构）的所有自动对象进行析构。
  - 从最后一个catch处理之后开始恢复执行。

例3使用带析构语义的类的C++异常处理

```
class MyException
 {
   public:
      MyException(const string &message) : message(message) {}
      ~MyException() {}
      const string &getMessage() const { return message; }
   private:
      string message;
};
class Demo
 {
   public:
      Demo() { cout << "Constructor of Demo" << endl; }
      ~Demo() { cout << "Destructor of Demo" << endl; }
};
```

例3使用带析构语义的类的C++异常处理

```
void func() throw (MyException)
{
    Demo d;
    cout << "Throw MyException in func()" << endl;
    throw MyException("exception thrown by func()");
}

int main()
{
    cout << "In main function" << endl;
    try {
        func();
    } catch (MyException& e) {
        cout << "Caught an exception: " << e.getMessage() << endl;
    }
    cout << "Resume the execution of main()" << endl;
    return 0;
}
```

```
In main function
Constructor of Demo
Throw MyException in func()
Destructor of Demo
Caught an exception: exception thrown by func()
Resume the execution of main()
```

课堂练习：读代码，分析结果

```
void f();
class T
{ public:
      T()
      {
          cout<<“constructor”<<endl;
          try{
                throw “exception 1”;
          }
          catch(char *)
          { cout<< “exception 1”<<endl; }
          throw “exception”;
      }
      ~T()
      { cout<<“destructor”; }
};
```

课堂练习：读代码，分析结果

```
void main()
{
   cout<<“main function”<<endl;
   try{
        f() ;
   }
   catch(char *)
   { cout<< “exception 2”<<endl; }

   cout<<“main function”<<endl;
}
void f()
{
  T t;
}
```

课堂练习：读代码，分析结果——答案

```
void f();
class T
{ public:
      T()
      {
          cout<<“constructor”<<endl;    //2
          try{
                throw “exception 1”;
          }
          catch(char *)
          { cout<< “exception 1”<<endl; }  //3
          throw “exception”;  //将被执行
      }
      ~T()
      { cout<<“destructor”; }//不执行
};
```

课堂练习：读代码，分析结果——答案

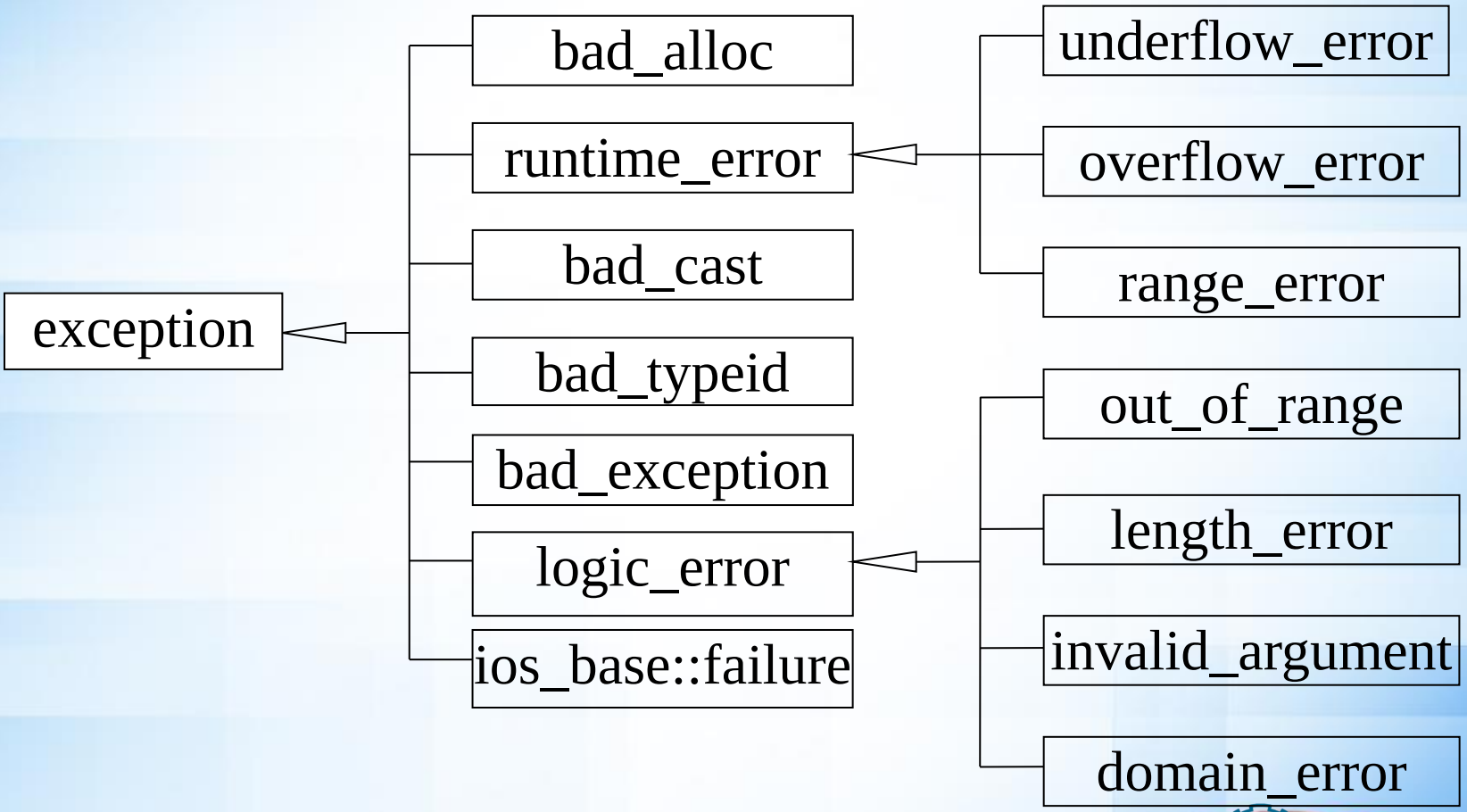
```
void main()
{
   cout<<“main function”<<endl;  //1
   try{
        f() ;
   }
   catch(char *)
   { cout<< “exception 2”<<endl; }//4

   cout<<“main function”<<endl; //5
}
void f()
{
  T t;
}
```

结论：

构造函数中含有异常处理，析构函数没有运行。

4 标准程序库异常处理


exception
bad_alloc
runtime_error
bad_cast
bad_typeid
bad_exception
logic_error
ios_base::failure
underflow_error
overflow_error
range_error
out_of_range
length_error
invalid_argument
domain_error

4 标准程序库的异常类

- **exception：标准程序库异常类的公共基类**
- **logic_error表示可以在程序中被预先检测到的异常**
  - 如果小心地编写程序，这类异常能够避免
- **runtime_error表示难以被预先检测的异常**

例4 三角形面积计算

- 编写一个计算三角形面积的函数，函数的参数为三角形三边边长*a*、*b*、*c*，可以用Heron公式计算：

- 设 $p=\dfrac{a+b+c}{2}$，则三角形面积

$$S=\sqrt{p(p-a)(p-b)(p-c)}$$

例4（续）

```
#include <iostream>
#include <cmath>
#include <stdexcept>
using namespace std;
//给出三角形三边长，计算三角形面积
double area(double a, double b, double c)  throw (invalid_argument)
{
//判断三角形边长是否为正
    if (a <= 0 || b <= 0 || c <= 0)
        throw invalid_argument("the side length should be positive");
//判断三边长是否满足三角不等式
    if (a + b <= c || b + c <= a || c + a <= b)
        throw invalid_argument("the side length should fit the triangle
    inequation");
//由Heron公式计算三角形面积
    double s = (a + b + c) / 2;
    return sqrt(s * (s - a) * (s - b) * (s - c));
}
```

例4（续）

```
int main() {
    double a, b, c;      //三角形三边长
    cout << "Please input the side lengths of a triangle: ";
    cin >> a >> b >> c;
    try {
        double s = area(a, b, c);       //尝试计算三角形面积
        cout << "Area: " << s << endl;
    } catch (exception &e) {
        cout << "Error: " << e.what() << endl;
    }
    return 0;
}
```

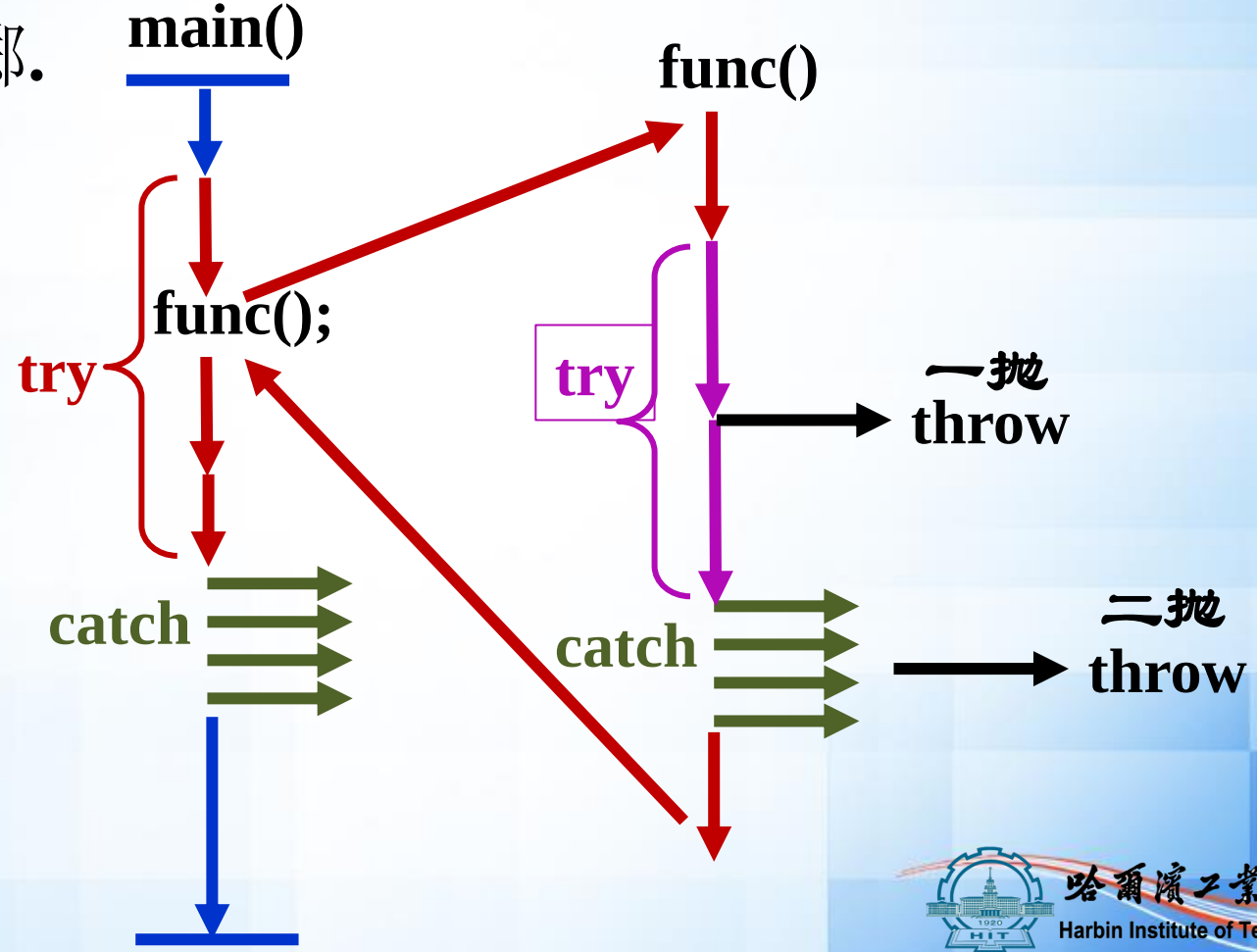
```
Please input the side lengths of a triangle: 3 4 5
Area: 6
```

```
Please input the side lengths of a triangle: 0 5 5
Error: the side length should be positive
```

```
Please input the side lengths of a triangle: 1 2 4
Error: the side length should fit the triangle inequation
```

二次抛掷的图示

- 镶嵌在catch函数中的不带表达式的throw语句叫二次抛掷.



练习二：分析其中的调用及抛掷关系(将…改成cout)

```
void f() {....
throw 10;
}
void g() {.....
throw 3.2;
}
void h() { ....
    throw 65.5;
    try{ ....
    throw 'a';
    f(); g(); }
    catch( float )
    { .....
    throw;
    }
}
```

```
void k()
{....
    try{ ....
    h(); }
    catch( float )
    { .....
    throw 10;
    }
    catch( int )
    { .....
    }
}
void m()
{....
k();
}
```

```
void main()
{ ....
    try
    { ....
      m();
    }
    catch( int )
    { ..... }
    catch( float )
    { ..... }
    catch( ... )
    { .....
    }
}
```

如果把float换成double呢？

5 编写异常安全程序的原则

- 明确哪些操作绝对不会抛掷异常
  - 这些操作是异常安全编程的基石
  - 例：基本数据类型的绝大部分操作，针的赋值、算术运算和比较运算，STL容器的swap函数
- 尽量确保析构函数不抛掷异常

5 编写异常安全程序的原则

- 避免异常发生时的资源泄漏
  - 一个函数，必须在有异常向外抛出前，释放应由它负责释放的资源。
  - 通常的解决方案
    - 把一切动态分配的资源都包装成栈上的对象，利用抛掷异常时自动调用对象析构函数的特性来释放资源。
    - 对于必须在堆上构造的对象，可以用智能指针auto_ptr加以包装。

6 智能指针auto_ptr

- C++标准库的一个类模板
  - 在memory头文件中定义
  - 有一个类型参数X，表示智能指针指向数据的类型
  - 每个智能指针对象关联一个普通指针
- 构造函数：
  - `explicit auto_ptr(X *p = 0) throw();`
- 获得与智能指针对象关联的指针：

  X *get() const throw();
  - 由于auto_ptr的“*”与“->”运算符已被重载，对一个auto_ptr的对象使用“*”和“->”，等价于对它所关联的指针使用相应运算符。

6 智能指针auto_ptr（续）

- 更改智能指针对象关联的指针

  void reset(X *p = 0) throw();

  - 原指针所指堆对象会被删除
- 解除与当前指针的关联

  X* release() throw();
- 注意事项
  - 智能指针对象执行赋值和拷贝构造时，原对象的关联指针会被解除