



第8章 异常



本章学习要点

一个好的程序不仅要保证能实现所需要的功能，而且还应该有很好的容错能力。在程序运行过程中如果有异常情况出现，程序本身应该能解决这些异常，而不是死机。

本章主要介绍C++异常处理的语言机制，包括异常的**结构**、**捕捉**和**处理**，以及**异常类**。

通过本章的学习，掌握了C++异常处理的机制，我们就可以在编制程序时灵活地加以运用，从而使我们编制的程序在遇到异常情况时能摆脱大的影响，避免出现死机等现象。



程序调试与异常处理

- 在程序设计的过程中，不可避免出现各种错误。随着程序规模的扩大，程序中出现的错误也会越来越多。为此，各种程序设计语言都需要进行程序调试。此外，异常处理是C++的一个主要特征，它提供了完美的结果出错处理的方法。
- 使用任何一种语言设计的程序都或多或少存在程序错误，程序错误常常由于程序设计人员的疏忽产生。一般而言，程序错误主要包含编译错误、逻辑错误和运行错误三类。



1.1 编译错误

- 编译错误也称为语法错误，其定义如图-1所示。



- 对于初学者来说，编译错误是最容易犯的一类错误，但也是最容易排除的一类错误。

【示例-1】

- 下面程序在用户屏幕上输出“Welcome to C++”字样，代码如图-2所示。

```
#include<iostream>
using namespace std;
int main()
{
    cout<<"Welcome to C++"<<endl
    system("pause");
    return 0;
}
```

编译

```
#include<iostream>
using namespace std;
int main()
{
    cout<<"Welcome to C++"<<endl
    system("pause");
    return 0;
}
```

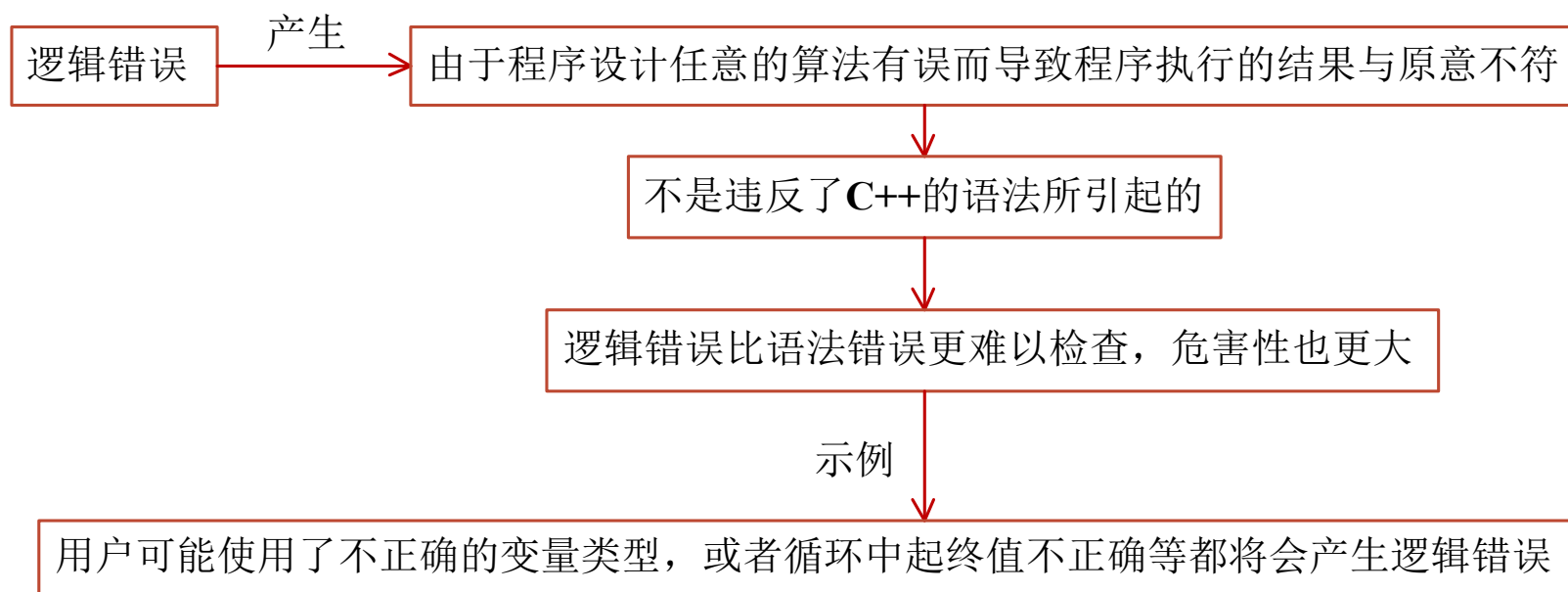
c.cpp(6) : error C2146: syntax error : missing ';' before identifier 'system'

上述程序因为在第5行语句结束时缺少了一个“;”，所以编译器给出编译错误的提示信息

此时，只需加上分号并重新编译即可解决该错误

1.2 逻辑错误

- 逻辑错误是指程序存在逻辑上的缺陷引起程序运行后，得不到所期望的结果。逻辑错误并不直接导致程序在编译期间和运行期间出现错误，因此不像编译错误那么容易发现。
- 逻辑错误的产生，如图-3所示。



- 一般来说，逻辑错误不会产生错误提示信息，需要用户仔细地分析程序。

【示例-2】

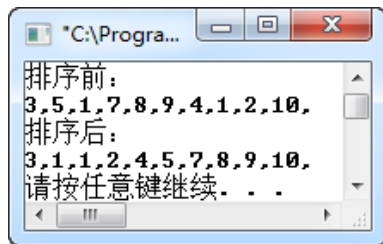
下面程序实现一个冒泡排序函数，对用户定义的一个数组进行升序排列后输出。其实现代码及结果如图-4所示。

```
#include <iostream>
using namespace std;
int sort(int array[], int n) 定义排序函数
{
    int temp;
    for (int i=1; i<n; i++)
    {
        for (int j=n-1; j>1; j--) 逻辑错误：下标取值不正确
        {
            if (array[j]<array[j-1]) 交换数组元素
            {
                temp=array[j-1];
                array[j-1]=array[j];
                array[j]=temp;
            }
        }
    }
    return 0;
}

int main()
{
    int array[10]={3, 5, 1, 7, 8, 9, 4, 1, 2, 10}; 定义一维数组并初始化
    int i;
    cout<<"排序前: "<<endl;
    for (i=0; i<10; i++)
        cout<<array[i]<<" ";
    cout<<endl;
    sort(array, 10);
    cout<<"排序后: "<<endl;
    for (i=0; i<10; i++)
        cout<<array[i]<<" ";
    cout<<endl;
    system("pause");
    return 0;
}
```

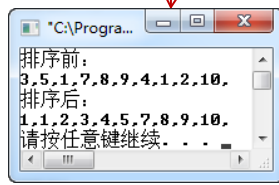
程序的运行结果并没有按照读者的意图对数组中的元素进行升序排列

说明冒泡排序函数存在逻辑错误



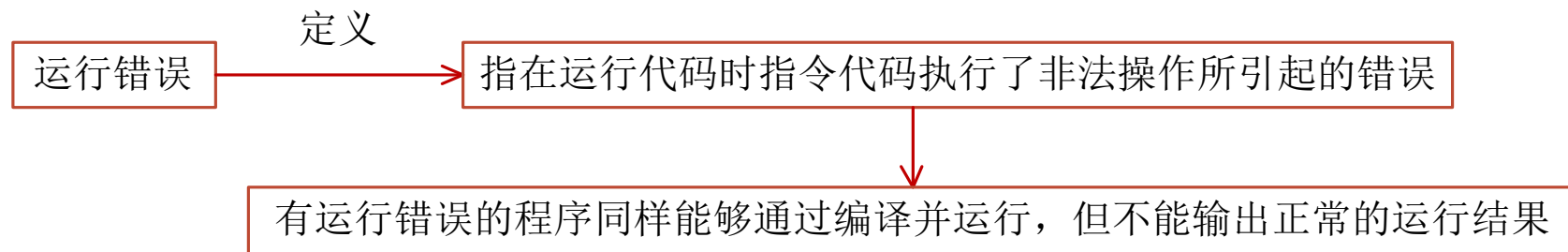
- 提示：逻辑错误的排除需要读者仔细理解算法，一定要注意数组的下标越界、内存溢出等问题。

逻辑错误的原因在于第8行的for循环语句中变量j的终值应为1，即将语句j>1改为j>=1



1.3 运行错误

- 运行错误可以认为是逻辑错误的一种特殊情况，其定义如图-5所示。



- 一般来说，编译器对许多错误将以运行错误的形式体现，如除法时分母为零，被操作的驱动器未准备好或磁盘读写有错等错误，导致程序不能继续执行。

【示例-3】

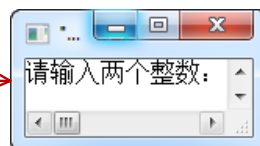
- 下面程序对一个除数为0的变量进行了除法运算，这是不允许的。C++将以运行错误的形式体现该错误，实现代码及结果如图-6所示。

```
#include <iostream>
using namespace std;
int main()
{
    int a,b;
    cout<<"请输入两个整数: "<<endl;
    cin>>a>>b;
    cout<<"a/b= "<<a/b<<endl;
    system("pause");
    return 0;
}
```

定义整型变量

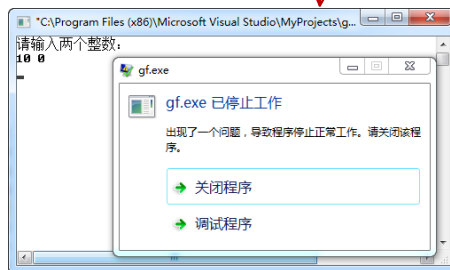
输出a/b的运算结果

编译并运行程序，弹出用户界面接收输入




该程序没有错误，能够通过编译也能够正常运行

当用户在运行该程序后输入测试数据10和0，返回运行错误



- 因此，在判断一个程序是否有运行错误时不能单靠C++语法的检查，还需要加强程序健壮性检查，从而避免出现运行错误。



1.4 程序调试

- 程序调试是指发现程序的错误及其对错误的改正。程序调试不仅仅依赖于编译器发现错误，更重要的是程序设计人员对算法和C++语法规则的理解。
- 一般来说，C++源程序出现编译错误后，编译器将自动显示出错行和错误提示，读者根据提示进行修改即可，编译错误一般不需要进行逐条语句调试。



1.断言函数（assert）

- 断言就是判断。assert有两种，assert和ASSERT。其中，assert是标准C++中的宏，assert宏的原型定义在<assert.h>中，ASSERT是MFC中的宏，它们的使用方法一样。使用格式如下：
- assert(expression);
- 执行时先测试逻辑表达式expression，若expression的值为0，那么它先打印一条出错信息，然后通过调用 abort 来终止程序运行;否则执行assert后的语句。

- 用法总结:

1)在函数开始处检验传入参数的合法性, 如:

- `int resetBufferSize(int nNewSize)`
 {
 //功能:改变缓冲区大小,
 //参数:nNewSize 缓冲区新长度
 //返回值:缓冲区当前长度
 //说明:保持原信息内容不变 nNewSize<=0表示清除缓冲区
 assert(nNewSize >= 0);
 assert(nNewSize <= MAX_BUFFER_SIZE);

 ...
 }

2)每个**assert**只检验一个条件,因为同时检验多个条件时,如果断言失败,无法直观的判断是哪个条件失败

- 不好: **assert**(nOffset>=0 && nOffset+nSize<=m_nInfomationSize);
- 好: **assert**(nOffset >= 0);
 assert(nOffset+nSize <= m_nInfomationSize);

3)不能使用改变环境的语句,因为**assert**只在DEBUG个生效,如果这么做,会使用程序在真正运行时遇到问题

- 错误: **assert(i++ < 100)**

这是因为如果出错,比如在执行之前*i*=99,那么这条语句就不会执行,那么*i++*这条命令就没有执行。

- 正确: **assert(i < 100);**
i++;

4)**assert**和后面的语句应空一行,以形成逻辑和视觉上的一致感

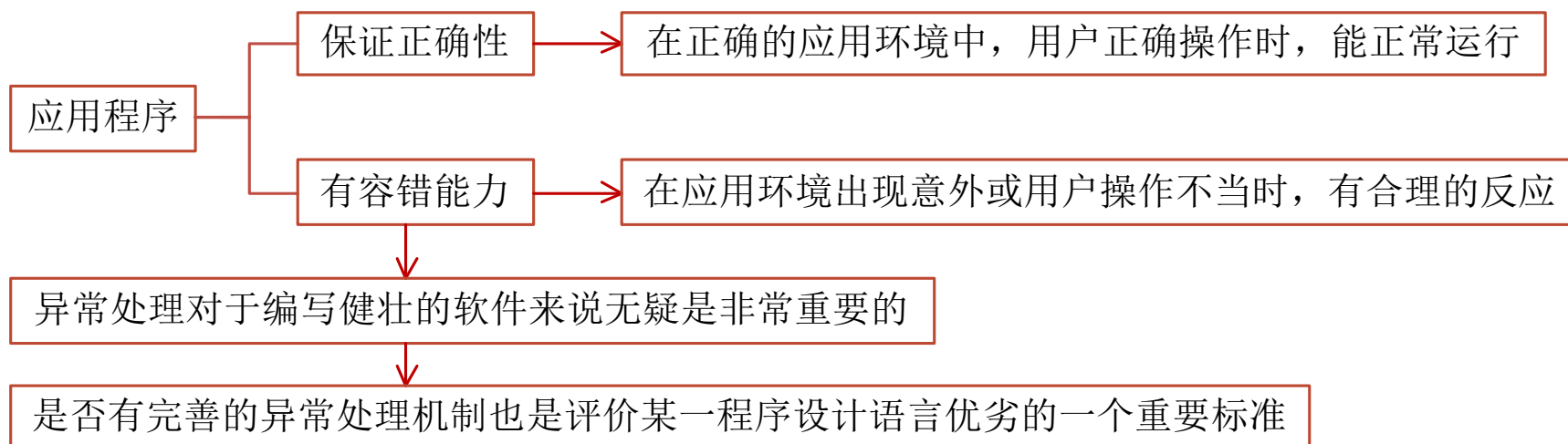
- **ASSERT**只有在Debug版本中才有效，如果编译为Release版本则被忽略掉。（在C中，**ASSERT**是宏而不是函数），使用**ASSERT**“断言”容易在debug时输出程序错误所在。
- 而**assert()**的功能类似，它是ANSI C标准中规定的函数，它与**ASSERT**的一个重要区别是可以用在Release版本中。
- 使用**assert**的缺点是，频繁的调用会极大的影响程序的性能，增加额外的开销。



2 异常处理

- 异常是指程序在运行过程中，由于使用环境的变化及用户的操作而产生的错误，因此异常可以认为是错误的狭义概念。异常处理是在程序设计过程中，针对可预测的异常编制相应的预防代码或处理代码，以便防止异常发生后造成严重后果。
- 在运行没有异常处理的程序时，如果运行情况出现异常，由于程序本身不能处理，程序只能终止运行。如果在程序中设置了异常处理，则在运行情况出现异常时，由于程序本身已设定了处理的方法，于是程序的流程就转到异常处理代码段处理。
- 需要说明的是：只要程序运行时出现与人们期望的情况不同，都可以认为是异常，并对它进行异常处理。因此，所谓异常处理是指对程序运行时出现的差错以及其他例外情况的处理。

- 一个应用程序，既要保证其正确性，还应有容错能力。具体地说，如图-7所示。



C++的异常处理机制能将异常检测与异常处理分离开来，当异常发生时能自动调用异常处理程序进行错误处理。

异常处理机制增加了程序的清晰性和可读性，使程序员能够编写出**清晰、健壮、容错**能力更强的程序，适用于大型软件开发。

8.1 异常处理概述

商业软件常会提供许多错误处理代码，以便对各种可能出现错误的程序代码进行检测和处理，增加程序的健壮性。

处理程序错误的代码可能出现在源码的任何地方，比如判定`new`是否成功分配内存、数据下标是否越界、运算溢出、除数0、无效参数等。

错误处理代码“污染”了程序源码、大量的错误处理代码使原本简单的程序变得晦涩难懂。

2、传统的异常处理方法

传统程序处理异常的典型方法是不断测试程序继续运行的必要条件，并对测试结果进行处理。

形式如下伪码所示：

执行任务1

if 任务1未能被正确执行

执行错误处理程序

执行任务2

if 任务2未能正确执行

执行错误处理程序

执行任务3

缺点

错误处理代码分布在整个程序的各个部分，使程序受到了错误处理代码的“污染”。

3、C++异常处理思想

其基本思想是将异常发生和异常处理分别放在不同的函数中，产生异常的函数不需要具备处理异常的能力。

当一个函数出现异常时，它可以抛出一个异常，然后由该函数的调用者捕获并处理这个异常，如果调用者不能处理，它可以将该异常抛给其上一级的调用者处理。

一般而言，异常处理的基本思想是：在底层发生的问题，逐级上报，直到有能力可以处理异常的那级为止。具体地说，如图所示。

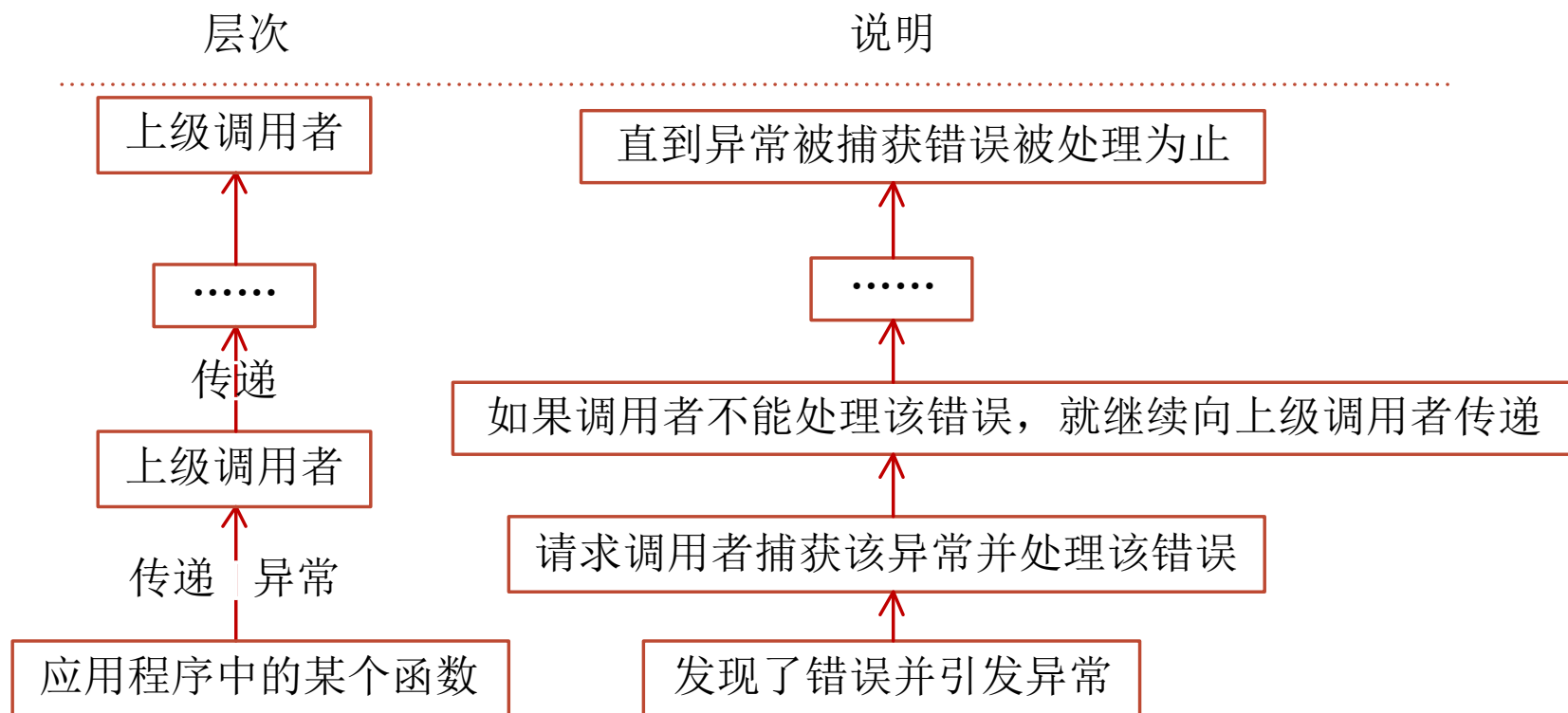


图17-8 异常处理的基本思想

C++异常处理的目的是在异常发生时，尽可能地减少破坏，使其不影响或尽量少影响程序其他部分的运行。

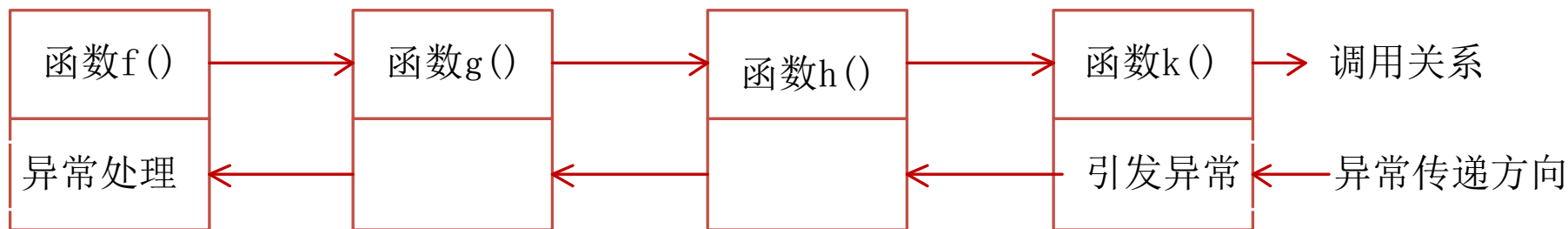


图17-9 异常的传递方向

4、C++的异常处理的作用

改善程序的可读性和可维护性，将异常处理代码与主程序代码分离，适合团队开发大型项目。

有力的异常检测和可能的异常恢复，以统一方式处理异常。
在异常引起系统错误之前处理异常。

处理由库函数引起的异常。

在出现无法处理的异常时执行清理工作，并以适当的方式退出程序。

在动态调用链中有序地传播异常处理。

8.2 异常处理基础

8.2.1 异常处理的结构

C++处理异常的机制由3个部分组成：检查（try）、抛出（throw）和捕捉（catch）。把需要检查的语句放在try块中，throw用来当出现异常时抛出一个异常信息，而catch则用来捕捉异常信息，如果捕捉到了异常信息，就处理它。try-throw-catch构成了C++异常处理的基本结构，形式如下：

try{

.....

if err1 **throw** xx1

.....

if err2 **throw** xx2

.....

if errn **throw** xxn

}

catch(type1 arg){.....}

catch(type2 arg){.....}

catch(typem arg){.....}

.....

//try程序块

//异常类型1错误处理

//异常类型2错误处理

//异常类型m错误处理

try-throw-catch异常处理的执行逻辑如下

- 1、当程序执行过程中遇到try块时，将进入try块并按正常的程序逻辑顺序执行其中的语句
- 2、如果try块的所有语句都被正常执行，没有发生任何异常，那么try块中就不会有异常被throw。在这种情况下，程序将忽略所有的catch块，顺序执行那些不属于任何catch块的程序语句，并按正常逻辑结束程序的执行，就像catch块不存在一样。

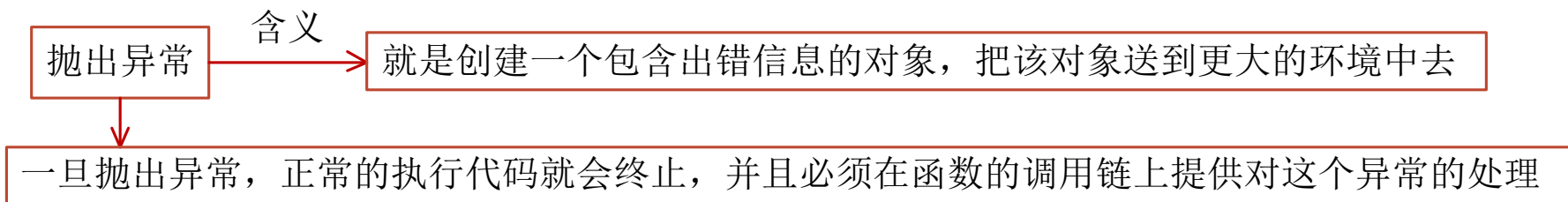
- 3、如果在执行try块的过程中，某条语句产生错误并用throw抛出了异常，则程序控制流程将自此throw子句转移到catch块，try块中该throw语句之后的所有语句都不会再被执行了。
- 4、C++将按catch块出现的次序，用异常的数据类型与每个catch参数表中指定的数据类型相比较，如果两者类型相同，就执行该catch块，同时还将把异常的值传递给catch块中的形参arg（如果该块有arg形参）。只要有一个catch块捕获了异常，其余catch块都将被忽略。
- 5、如果没有任何catch能够匹配该异常，C++将调用系统默认的异常处理程序处理该异常，其通常做法是直接终止该程序的运行。

【例8-1】 异常处理的简单例程。

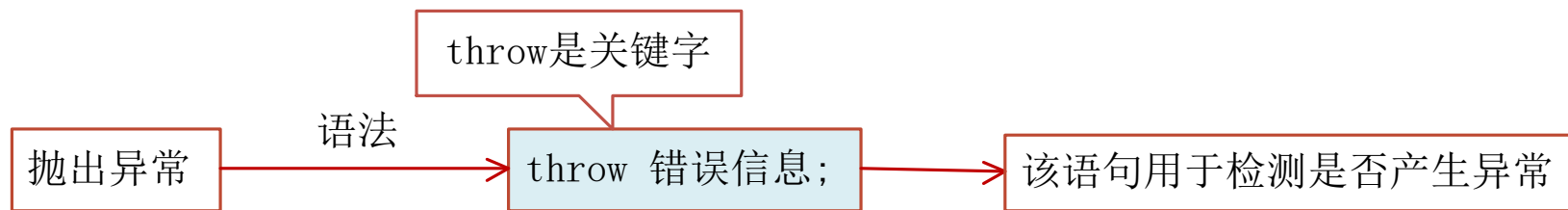
```
//CH8-1.cpp
#include<iostream>
using namespace std;
void main(){
    cout<<"1--befroe try block..."<<endl;
    try{
        cout<<"2--Inside try block..."<<endl;
        throw 10;
        cout<<"3--After throw ...."<<endl;
    }
    catch(int i) {
        cout<<"4--In catch block1 ... exception..errcode is.."<<i<<endl;
    }
    catch(char * s) {
        cout<<"5--In catch block2 ... exception..errcode is.."<<s<<endl;
    }
    cout<<"6--After Catch...";
}
```

8.2.2 抛出异常

如果程序发生异常情况，而在当前环境中获取不到异常处理的足够信息，可以将异常抛出。抛出异常的含义，如图所示。



当一段程序中发现错误数据，但是该程序不知道如何处理时，可以抛出异常。在C++中，使用`throw`来抛出异常。抛出异常的语法如图所示。



将上述除数为0的程序处理，当用户输入除数为0时抛出异常，其实现代码及结果如图所示。

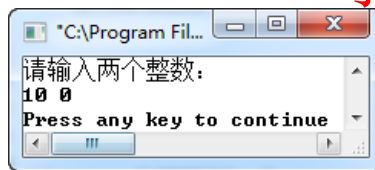
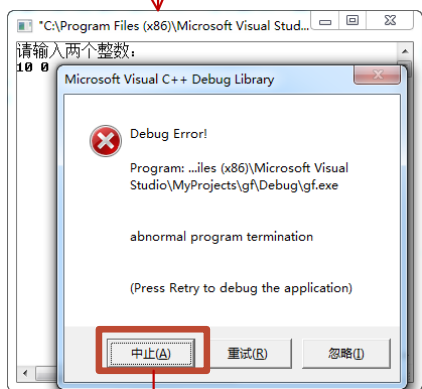
```
#include <iostream>
using namespace std;
int main()
{
    int a,b;
    cout<<"请输入两个整数："<<endl;
    cin>>a>>b;
    if (b==0)
    {
        throw b;
    }
    else
    {
        cout<<"a/b= " <<a/b<<endl;
        system("pause");
        return 0;
    }
}
```

判断除数是否为0

抛出异常

输出a/b的结果

输入两测试数据10和0，程序将退出执行



当用户使用throw语句抛出异常时，可以避免程序出现运行错误。但是，如果只有异常的抛出，程序将中途退出，这也是具体程序设计中不允许的，为此，C++引入了异常的捕获和处理机制。

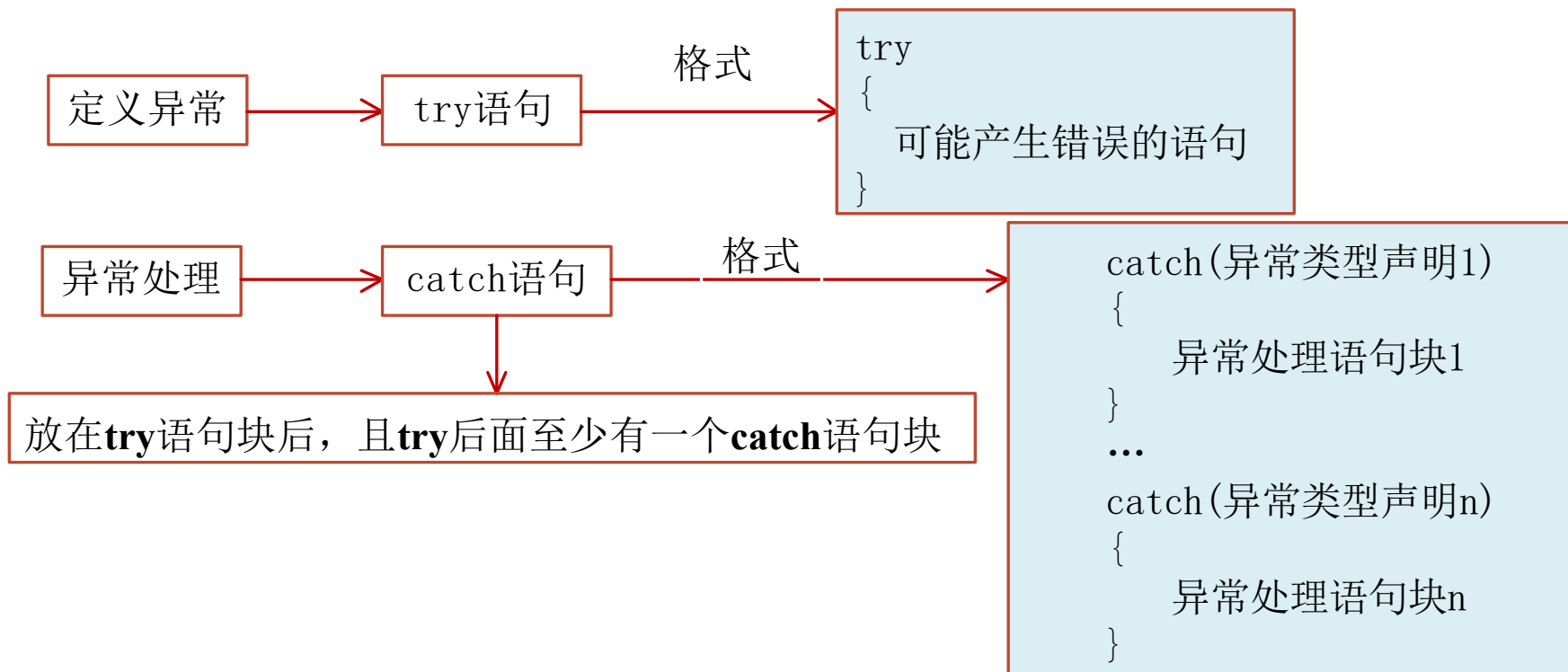
代码只简单的在函数中抛出异常，不进行异常处理，因此程序会中途退出

8.2.3 捕获异常

捕获异常由`catch`完成，`catch`必须紧跟在与之对应的`try`块后面。如果异常被某`catch`捕获，程序将执行该`catch`块中的代码，之后将继续执行`catch`块后面的语句；如果异常不能被任何`catch`块捕获，它将被传递给系统的异常处理模块，程序将被系统异常处理模块终止。

`catch`根据异常的数据类型捕获异常，如果`catch`参数表中异常声明的数据类型与`throw`抛出的异常的数据类型相同，该`catch`块将捕获异常。

注意：`catch`在进行异常数据类型的匹配时，**不会进行数据类型的默认转换**，只有与异常的数据类型精确匹配的`catch`块才会被执行。



【例8-2】 `catch`捕获异常时，不会进行数据类型的默认转换。

```
//CH8-2.cpp
```

```
#include<iostream>
```

```
using namespace std;
```

```
void main(){
```

```
    cout<<"1--befroe try block..."<<endl;
```

```
    try{
```

```
        cout<<"2--Inside try block..."<<endl;
```

```
        throw 10;
```

```
        cout<<"3--After throw ...."<<endl;
```

```
    }
```

```
    catch(double i) {                //仅此与例8.1不同
```

```
        cout<<"4--In catch block1 .. an int type is.."<<i<<endl;
```

```
    }
```

```
    cout<<"5--After Catch...";
```

```
}
```

8.3.1 在函数中处理异常

异常处理可以局部化为一个函数，当每次进行该函数的调用时，异常将被重置。

【例8-3】 `temperature`是一个检测温度异常的函数，当温度达到冰点或沸点时产生异常。

```
#include<iostream>
using namespace std;
void temperature(int t)
{
    try{
        if(t==100) throw "沸点！ ";
        else if(t==0) throw "冰点！ ";
        else cout<<"the temperature is OK..."<<endl;
    }
    catch(int x){cout<<"temperatore="<<x<<endl;}
    catch(char *s){cout<<s<<endl;}
}

void main(){
    temperature(0);           //L1
    temperature(10);          //L2
    temperature(100);          //L3
}
```


8.3.2 在函数调用中完成异常处理

将产生异常的程序代码放在一个函数中，将检测处理异常的函数代码放在另一个函数中，能让异常处理更具灵活性和实用性。

【例8-4】 异常处理从函数中独立出来，由调用函数完成。

```
#include<iostream>
using namespace std;
void temperature(int t) {
    if(t==100) throw "沸点! ";
    else if(t==0) throw "冰点! ";
    else cout<<"the temperature is ..."<<t<<endl;
}
void main(){
    try{
        temperature(10);
        temperature(50);
        temperature(100);
    }
    catch(char *s){cout<<s<<endl;}
}
```

增补(选讲) 限制函数异常

限制异常的方法

- 1、 当一个函数声明中不带任何异常描述时，它可以抛出任何异常。例如：

int f(int,char); //函数f可以抛出任何异常

- 2、 在函数声明的后面添加一个**throw**参数表，在其中指定函数可以抛出的异常类型。例如：

int g(int,char) throw(int,char); //只允许抛出int和char异常。

- 3、 指定**throw**限制表为不包括任何类型的空表，不允许函数抛出任何异常。如：

int h(int,char) throw();//不允许抛出任何异常

【例8-12】 设计函数Errhandler，限制它只能抛出int、char和double类型的异常。

//eg8.cpp

```
#include<iostream>
```

```
using namespace std;
```

```
void Errhandler(int n)throw(int,char,double) {
```

```
    if(n==1) throw n;
```

```
    if(n==2) throw 'x';
```

```
    if(n==3) throw 1.1;
```

```
}
```

```
void main(){
```

```
    cout<<"Before Errhander..."<<endl;
```

```
    try{
```

```
        Errhandler(1);
```

```
    }
```

```
    catch(int i){ cout<<"catch an integer..."<<endl;}
```

```
    catch(char c){cout<<"catch a char..."<<endl;}
```

```
    catch(double d){cout<<"catch a double..."<<endl;}
```

```
}
```

8.4 异常处理的几种特殊情况

8.4.1 捕获所有异常

在多数情况下，`catch`都只用于捕获某种特定类型的异常，但它也具有捕获全部异常的能力。其形式如下：

```
catch(...) {  
    .....           //异常处理代码  
}
```

【例8-10】 改写前面的Errhandler函数，使之能够捕获所有异常。

//CH.cpp

```
#include<iostream>
using namespace std;
void Errhandler(int n)throw(){
    try{
        if(n==1) throw n;
        if(n==2) throw "dx";
        if(n==3) throw 1.1;
    }
    catch(...){cout<<"catch an exception..."<<endl;}
}
void main(){
    Errhandler(1);
    Errhandler(2);
    Errhandler(3);
}
```

8.4.2 再次抛出异常

如是`catch`块无法处理捕获的异常，它可以将该异常再次抛出，使异常能够在恰当的地方被处理。再次抛出的异常不会再被同一个`catch`块所捕获，它将被传递给外部的`catch`块处理。要在`catch`块中再次抛出同一异常，只需在该`catch`块中添加不带任何参数的`throw`语句即可。

【例8-11】 在异常处理块中再次抛出同一异常。

```
//eg8.cpp
#include<iostream>
using namespace std;
void Errhandler(int n)throw()
{
    try{
        if(n==1) throw n;
        cout<<"all is ok..."<<endl;
    }
    catch(int n){
        cout<<"catch an int exception inside..."<<n<<endl;
        throw;           //再次抛出本catch捕获的异常
    }
}

void main(){
    try{Errhandler(1); }
    catch(int x){ cout<<"catch an int exception in main..."<<x<<endl; }
    cout<<"....End..."<<endl;
}
```

8.4.3 异常的嵌套调用

`try`块可以嵌套，即一个`try`块中可以包括另一个`try`块，这种嵌套可能形成一个异常处理的调用链。

【例8-5】 嵌套异常处理示例。

`//CH8-5.cpp`

`#include<iostream>`

`using namespace std;`


```

void fb(){
    int *q=new int[10];
    try{
        fc();
        cout<<"return form fc()"<<endl;
    }
    catch(...){
        delete []q;
        throw;
    }
}

void fa()
{
    char *p=new char[10];
    try{
        fb();
        cout<<"return from fb()"<<endl;
    }
    catch(...){
        delete []p;
        throw;
    }
}

```

```

void main() {
    try {
        fa();
        cout<<"return from fa"<<endl;
    }
    catch(...) {cout<<"in main"<<endl;}
    cout<<"End"<<endl;
}

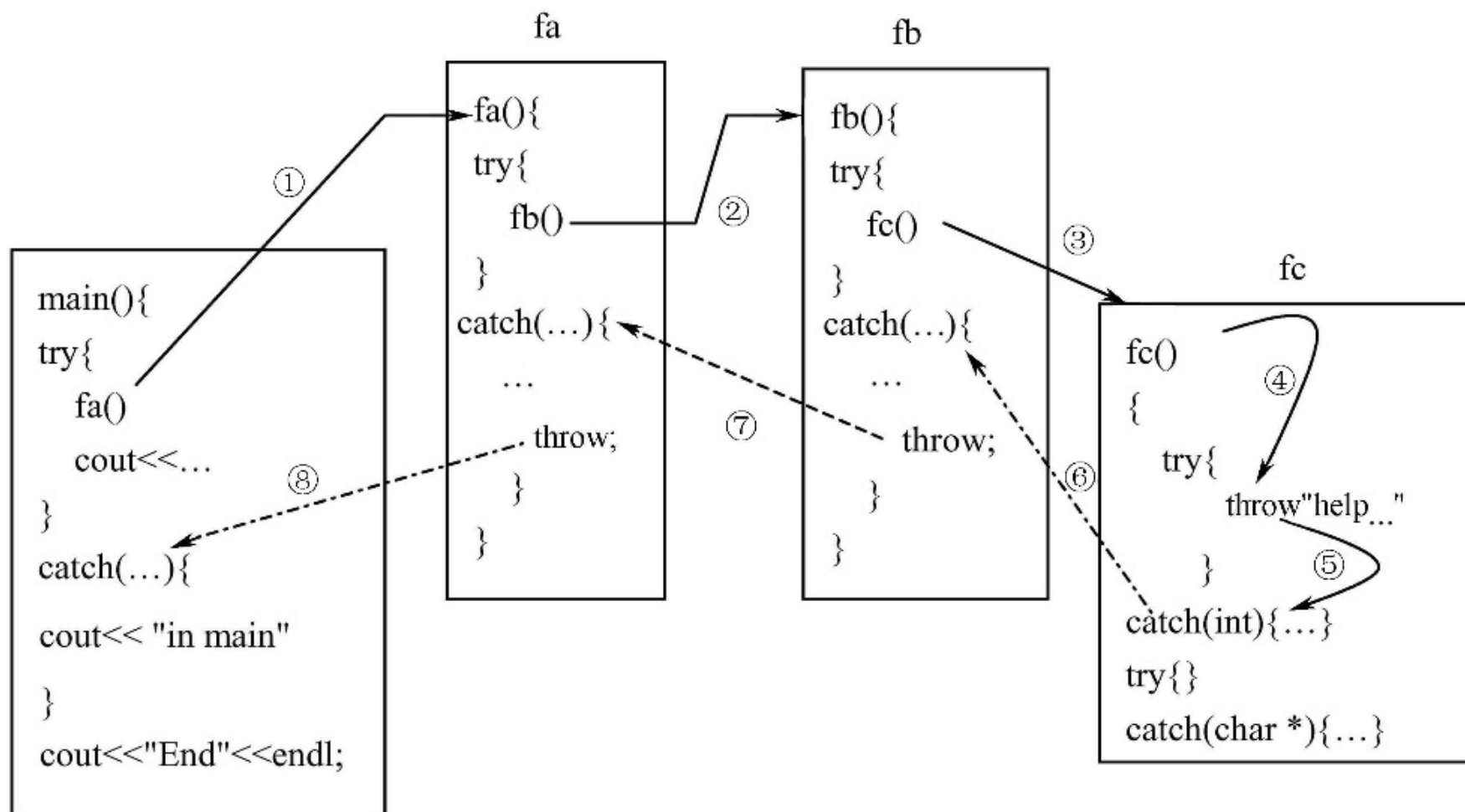
```

```

void fc(){
    try{throw "help...";}
    catch(int x){cout<<"in fc..int
hanlder"<<endl;}
    try{cout<<"no error
handle..."<<endl;}
    catch(char *px){cout<<"in
fc..char* hanlder"<<endl;}
}

```

例8-5的调用过程



8.5 异常和类

8.5.1 构造函数与异常

由于构造函数没有返回类型，在执行构造函数过程中若出现异常，传统处理方法可能是：

1. 返回一个处于错误状态的对象，外部程序可以检查该对象状态，以便判定该对象是否被成功构造。
2. 设置一个全局变量保存对象构造的状态，外部程序可以通过该变量值判断对象构造的情况。
3. 在构造函数中不做对象的初始化工作，而是专门设计一个成员函数负责对象的初始化。

C++中异常处理机制能够很好地处理构造函数中的异常问题，当构造函数出现错误时就抛出异常，外部函数可以在构造函数之外捕获并处理该异常。

8.5.2 异常类

1、关于异常类

异常可以是任何类型，包括自定义类。用来传递异常信息的类就是异常类。

异常类可以非常简单，甚至没有任何成员；也可以同普通类一样复杂，有自己的成员函数、数据成员、构造函数、析构函数、虚函数等，还可以通过派生方式构成异常类的继承层次结构。

【例8-7】 设计一个堆栈，当入栈元素超出了堆栈容量时，就抛出一个栈满的异常；如果栈已空还要从栈中弹出元素，就抛出一个栈空的异常。

//CH8-7.cpp

```
#include <iostream>
```

```
using namespace std;
```

```
const int MAX=3;
```

```
class Full{}; //L1 堆栈满时抛出的异常类
```

```
class Empty{}; //L2 堆栈空时抛出的异常类
```

```
class Stack{
```

```
private:
```

```
    int s[MAX];
```

```
    int top;
```

```
public:
```

```
    void push(int a);
```

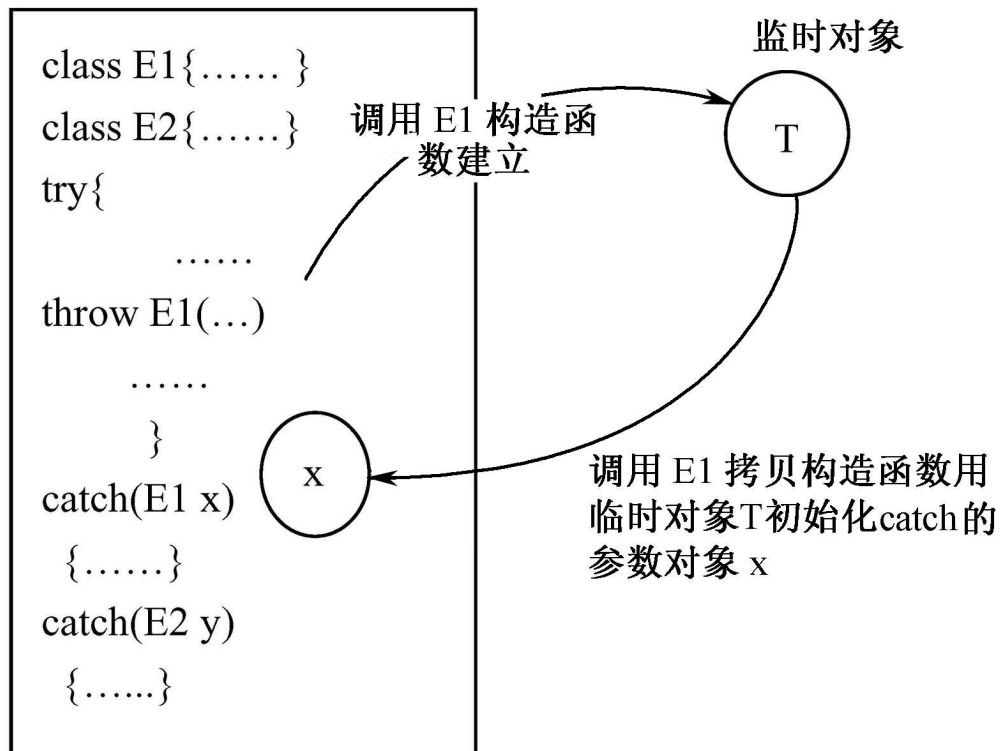
```
    int pop();
```

```
    Stack(){top=-1;}
```

```
};
```

```
void Stack::push(int a){
    if(top>=MAX-1) throw Full();
    s[++top]=a;
}
int Stack::pop(){
    if(top<0) throw Empty();
    return s[top--];
}
void main(){
    Stack s;
    try{
        s.push(10);    s.push(20);    s.push(30);
        //s.push(40);    //L5 将产生栈满异常
        cout<<"stack(0)= "<<s.pop()<<endl;
        cout<<"stack(1)= "<<s.pop()<<endl;
        cout<<"stack(2)= "<<s.pop()<<endl;
        cout<<"stack(3)= "<<s.pop()<<endl;           //L6
    }
    catch(Full){ cout<<"Exception: Stack Full"<<endl; }
    catch(Empty){ cout<<"Exception: Stack Empty"<<endl; }
}
```

8.5.2 异常类



2. 异常对象

由异常类建立的对象称为异常对象。

异常类的处理过程实际上就是异常对象的生成与传递过程。如右图

【例8-8】 修改例8-7的Full异常类，修改后的Full类具有构造函数和成员函数，还有一个数据成员。利用这些成员，可以获取异常发生时没有入栈的元素信息。

//CH8-8.cpp

#include <iostream>

using namespace std;

const int MAX=3;

class Full{

int a;

public:

Full(int i):a(i){}

int getValue(){return a;}

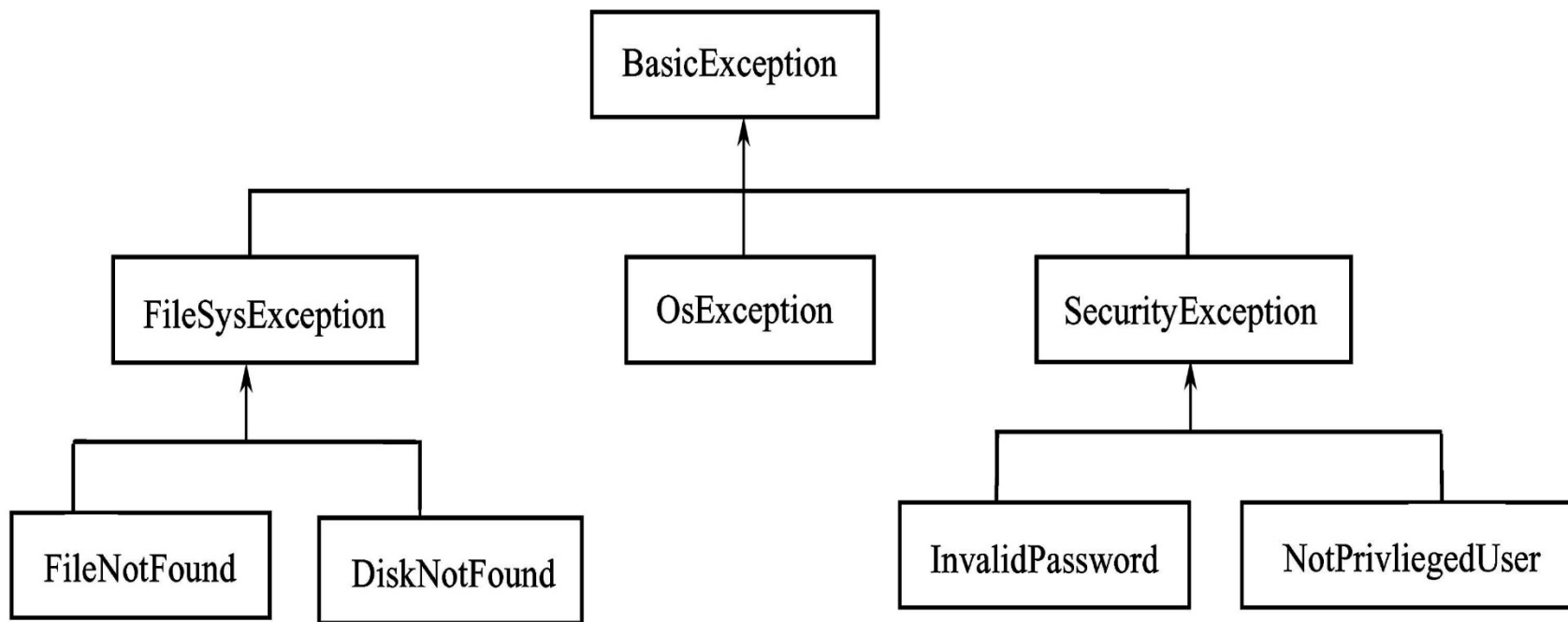
};

class Empty{};


```
class Stack{
private:
    int s[MAX];
    int top;
public:
    Stack(){top=-1;}
    void push(int a){
        if(top>=MAX-1)
            throw Full(a);
        s[++top]=a;
    }
    int pop(){
        if(top<0)
            throw Empty();
        return s[top--];
    }
};
```

```
void main(){
    Stack s;
    try{
        s.push(10);
        s.push(20);
        s.push(30);
        s.push(40);
    }
    catch(Full e){
        cout<<"Exception: Stack
            Full..."<<endl;
        cout<<"The value not push in stack: "
            <<e.getValue()<<endl;
    }
}
```

8.5.3 派生异常类的处理



一个进行远程登录访问程序的异常类层次结构如图。

【例8-9】 设计图8-3所示异常继承体系中从BasicException到FileSysException部分的异常类。

//CH8-9.cpp

```
#include<iostream>
```

```
using namespace std;
```

```
class BasicException{
```

```
public:
```

```
    char* Where(){return "BasicException...";}
```

```
};
```

```
class FileSysException:public BasicException{
```

```
public:
```

```
    char *Where(){return "FileSysException...";}
```

```
};
```

```
class FileNotFound:public FileSysException{
```

```
public:
```

```
    char *Where(){return "FileNotFound...";}
```

```
};
```

```
class DiskNotFound:public FileSysException{
```

```
public:
```

```
    char *Where(){return "DiskNotFound...";}
```

```
};
```

```
void main(){
    try{
        ..... //程序代码
        throw FileSysException();
    }
    catch(DiskNotFound p){cout<<p.Where()<<endl;}
    catch(FileNotFound p){cout<<p.Where()<<endl;}
    catch(FileSysException p){cout<<p.Where()<<endl;}
    catch(BasicException p){cout<<p.Where()<<endl;}
    try{
        ..... //程序代码
        throw DiskNotFound();
    }
    catch(BasicException p){cout<<p.Where()<<endl;}
    catch(FileSysException p){cout<<p.Where()<<endl;}
    catch(DiskNotFound p){cout<<p.Where()<<endl;}
    catch(FileNotFound p){cout<<p.Where()<<endl;}
}
```

8.5.3 派生异常类的处理

用多态性处理异常

异常类继承结构也可以用多态实现，多态可以简化异常的捕获。

例8-9的多态实现程序如下，其中省略的代码与例8-9完全相同。

```
#include <iostream>
using namespace std;
class BasicException{
public:
    virtual char* Where(){return "BasicException...";}
};
.....
void main(){
try{
//.....  程序代码
throw FileSysException();
}
catch(BasicException &p){cout<<p.Where()<<endl;}
try{
//.....  程序代码
throw DiskNotFound();
}
catch(BasicException &p){cout<<p.Where()<<endl;}
}
```

标准异常及其头文件

头文件

异常类型

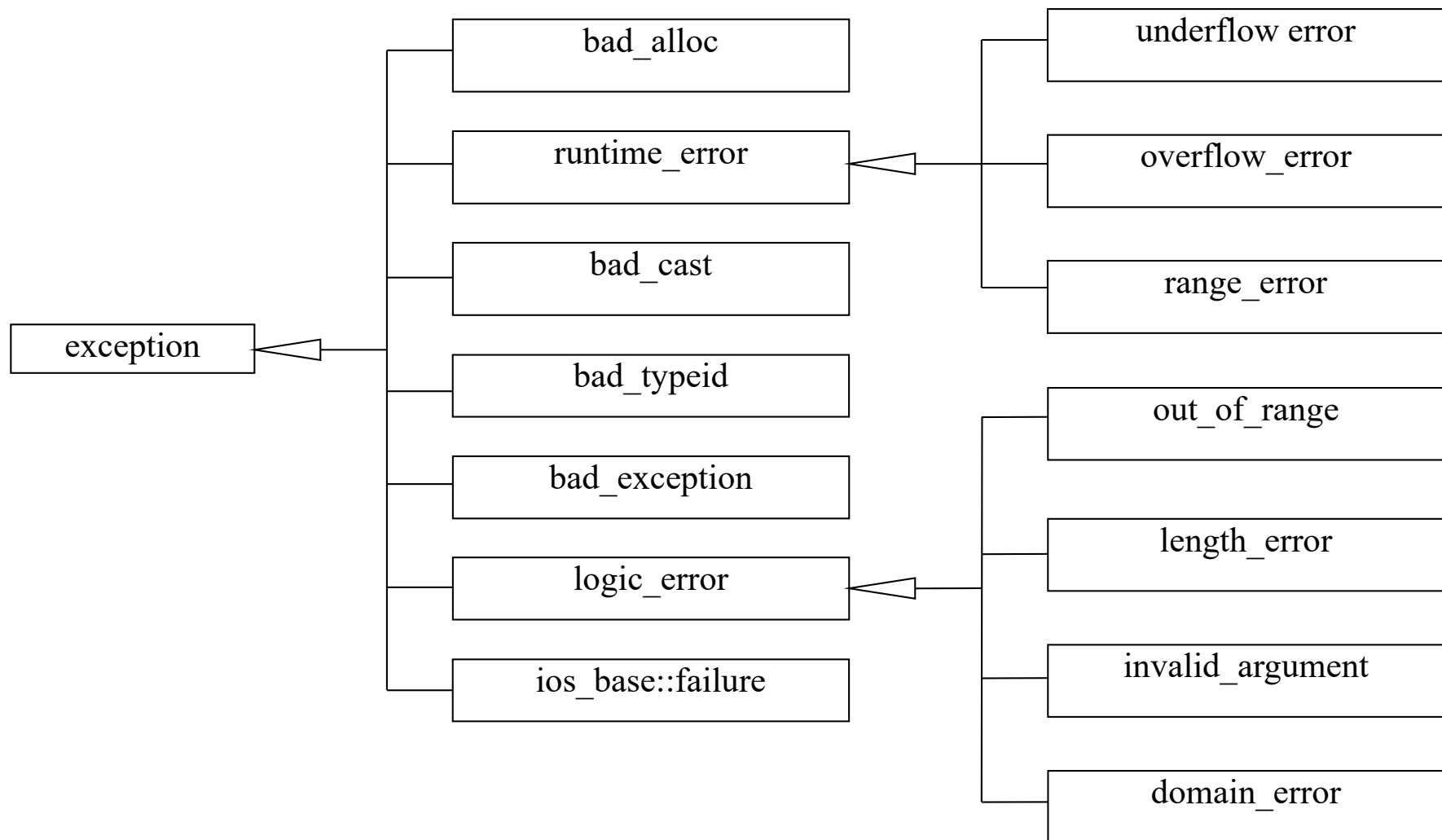
| | |
|-------------|--------------------------|
| <exception> | exception, bad exception |
|-------------|--------------------------|

| | |
|-------|---------------|
| <new> | bad exception |
|-------|---------------|

| | |
|------------|----------------------|
| <typeinfo> | bad_cast, bad_typeid |
|------------|----------------------|

| | |
|-------------|--|
| <stdexcept> | Logic_error, runtime_error, domain_error, invalid_argument, length_error, out_of_range, range_error, overflow_error, underflow_error |
|-------------|--|

标准程序库异常处理



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



谢谢!