

2023

# 面向对象程序设计

## 第九讲： 异常处理

李际军

lijijun@cs.zju.edu.cn



# 学习目标 / GOALS

---



- ◆ 理解异常、异常处理的概念；
- ◆ 掌握用 `try`、`throw` 和 `catch` 分别监视、指定和处理异常；掌握面向对象程序设计的特点；
- ◆ 掌握处理未捕获和未预料的异常；
- ◆ 理解标准异常层次结构。



**01**

异常处理概述

**02**

异常处理的实现

**03**

构造函数、析构函数与异常处理

**04**

异常匹配

**05**

标准异常及层次结构

# 01



## 异常处理概述

- 异常就是在程序运行中发生的难以预料的、不正常的事件而导致偏离正常流程的现象；
- 发生异常将导致正常流程不能进行，就需要对异常进行处理；
- 异常处理 (exception handling) 就是在运行时刻对异常进行检测、捕获、提示、传递等过程。
- 这里的异常是指软件异常。

- 原因：在一个大型软件中，由于函数之间有着明确的分工和复杂的调用关系，发现错误的函数往往不具备处理错误的能力。
- C++ 语言异常处理机制的基本思想是**将异常的检测与处理分离**。
- 过程 1：当在一个函数体中**检测到异常条件**存在，但却无法确定相应的处理方法时，该函数将**引发一个异常**，由函数的直接或间接调用者**捕获这个异常并处理这个错误**。
- 过程 2：如果程序始终没有处理这个异常，最终它会被传到 C++ 运行系统那里，运行系统捕获异常后，通常只是简单地终止这个程序。

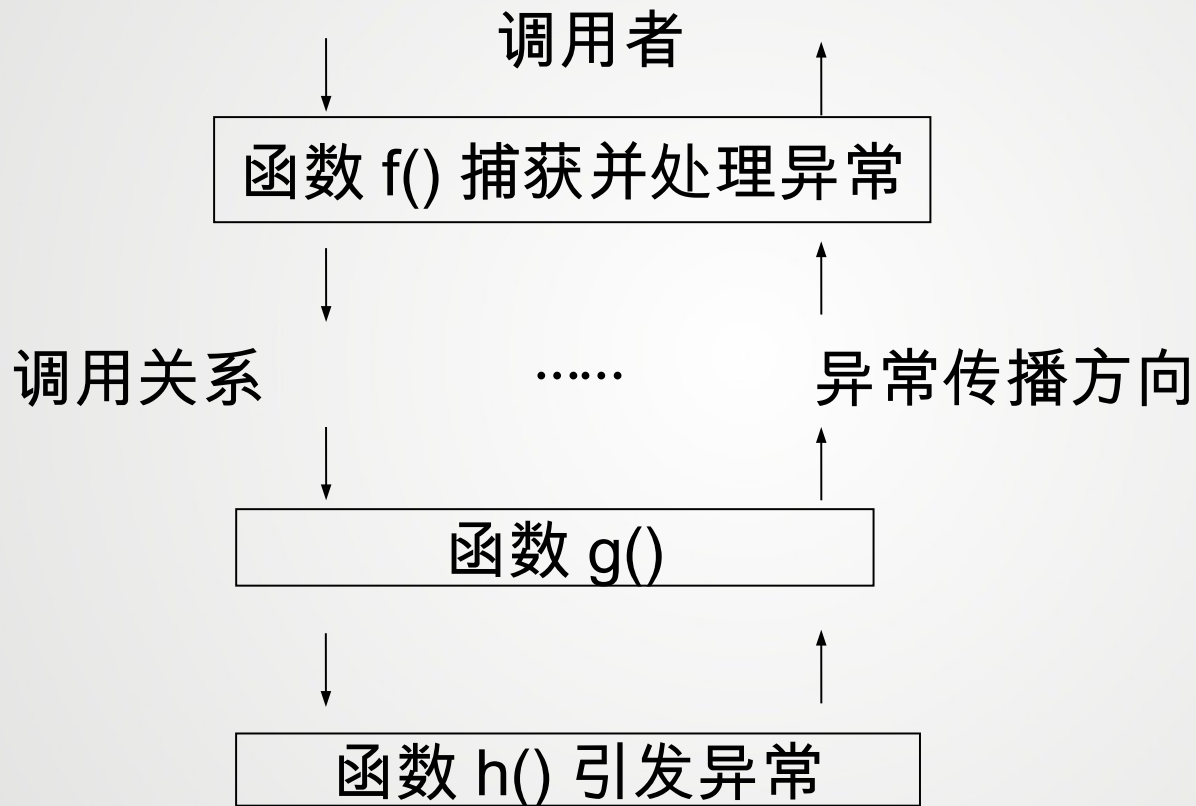
具有以下特点：

(1) 异常处理程序的编写不再繁琐。在错误有可能出现处写一些代码，并在后面的单独节中加入异常处理程序。如果程序中多次调用一个函数，在程序中加入一个函数异常处理程序即可。

(2) 异常发生不会被忽略。如果被调用函数需要发送一条异常处理信息给调用函数，它可向调用函数发送一描述异常处理信息的对象。如果调用函数没有捕捉和处理该错误信号，在后续时刻该调用函数将继续发送描述异常信息的对象，直到异常信息被捕捉和处理为止。

- 基本思想
  - 将异常检测与异常处理分离：异常检测部分检测到异常的存在时，抛出一个异常对象给异常处理代码，通过该异常对象，独立开发的异常检测部分和异常处理部分能够就程序执行期间所出现的异常情况进行通信。
- 由于异常处理机制使得异常的引发和处理不必在同一函数中；
  - 底层的函数可以着重解决具体问题而不必过多地考虑对异常的处理；
  - 上层调用者可以在适当的位置设计对不同类型异常的处理。





# 02



## 异常处理的实现

## 抛掷异常的程序段

.....

**throw** 表达式;  
( 异常抛出 )

.....

## 捕获并处理异常的程序段

**try**

复合语句 ( 保护段 )

**catch** ( 异常类型声明 )

复合语句 ( 处理段 )

**catch** ( 异常类型声明 )

复合语句 ( 处理段 )

.....



- 若有异常则通过 throw 操作创建一个异常对象并抛出；
- 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 块，然后执行 try 子块内的保护段；
- 如果在保护段执行期间没有引发异常，那么跟在 try 子块后的 catch 子句就不执行。程序继续执行紧跟在 try 块中最后一个 catch 子句后面的语句；
- catch 子句按其在 try 块后出现的顺序被检查。类型匹配的 catch 子句将捕获并处理异常（或继续抛出异常）；
- 如果找不到匹配的处理代码，则自动调用标准库函数 terminate，其默认功能是调用 abort() 终止程序。

- 异常处理的语法

throw 语句一般是由 throw 运算符和一个数据组成的，其形式为：

throw 表达式；

try-catch 的结构为

try

{ 被检查的语句 }

catch( 异常信息类型 [ 变量名 ] )

{ 进行异常处理的语句 }

### 2. try-catch 语句

```
try {  
    可能引发异常的语句序列;  
} // 受保护代码  
catch( 异常类型 1 异常变量 1){  
    处理代码 1;  
} // 异常处理器 1  
catch( 异常类型 2 异常变量 2){  
    处理代码 2;  
} // 异常处理器 2  
...  
catch(...){  
    处理代码;  
} // 异常处理器
```

- try 语句后的复合语句是代码的保护段。如果预料某段程序代码 ( 或对某个函数的调用 ) 有可能发生异常, 就将它放在 try 语句之后。如果这段代码 ( 或被调函数 ) 运行时真的遇到异常情况, 其中的 throw 表达式就会抛掷这个异常。
- catch 语句后的复合语句是异常处理程序, 捕获由 throw 表达式抛掷的异常。异常类型声明部分指明语句所处理的异常类型, 它与函数的形参相类似, 可以是某个类型的值, 也可以是引用。这里的类型可以是任何有效的数据类型, 包括 C++ 的类。当异常被抛掷以后, catch 语句便依次被检查。

### 3. 异常处理的执行过程

- ① 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。
- ② 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 语句** 就不执行，程序从异常被抛掷的 try 块后跟随的最后一个 catch 语句后面的语句继续执行下去。
- ③ 如果在保护段执行期间或在保护段调用的任何函数中（直接或间接的调用）有异常被抛掷，则从通过 **throw 创建的对象** 中创建一个异常对象（**隐含调用一个拷贝构造函数**），程序转到 **catch 处理段**。



这一点上，编译器能够处理抛掷类型的异常，在更高执行上下文中寻找一个 catch 语句（或一个能处理任何类型异常的 catch 处理程序）。catch 处理程序按其在 try 块后出现的顺序被检查。如果没有找到合适的处理程序，则继续检查下一个动态封闭的 try 块。此处理继续下去，直到最外层的封闭 try 块被检查完。

- ④ 如果匹配的 `catch` 处理器未找到，则 `terminate()` 将被自动调用，而函数 `terminate()` 的默认功能是调用 `abort` 终止程序。
- ⑤ 如果找到了一个匹配的 `catch` 处理程序，且它通过值进行捕获，则其形参通过拷贝异常对象进行初始化。如果它通过引用进行捕获，则参量被初始化为指向异常对象，在形参被初始化之后，“循环展开栈”的过程开始。这包括对那些在与 `catch` 处理器相对应的 `try` 块开始和异常丢弃地点之间创建的（但尚未析构的）所有自动对象的析构。



## 2. 异常处理的执行过程



简要说明

- **try** 块的重要性

- **try** 块包含了异常出现的语句。异常出现时，**try** 块提示编译器到哪里去查找 **catch** 块；

- 异常未出现时，几乎没有额外的运行成本。（异常的成本取决于编译器）

- **throw** 块的重要性

- 异常出现时，发出一个对象。（编译器初始化一个 **throw** 操作数的静态类型的临时对象。）



## 2. 异常处理的执行过程



简要说明

### • 编译器处理说明

- 编译器能够处理抛掷某种类型对象的异常，在更高执行上下文中寻找一个 **catch** 语句（或一个能处理任何类型异常的 **catch** 处理程序）。
- **catch** 处理程序按其在 **try** 块后出现的顺序被检查。如果没有找到合适的处理程序，则继续检查下一个外层动态封闭的 **try** 块。此处理继续下去，直到最外层的封闭 **try** 块被检查完。

- `catch` 处理程序的出现顺序很重要，因为在一个 `try` 块中，异常处理程序是按照它出现的顺序被检查的。只要找到一个匹配的异常类型，后面的异常处理都将被忽略。
- C++ 异常处理块中，比较特殊的是 `catch(...)`，它可以捕获任何异常，在它首发的情况下，其它的 `catch` 语句都不被检查。因此，`catch(...)` 应该放在最后。

```
void main()
{
    try {      // 异常可能被抛出的代码段      }
        catch(...) // 捕获所有异常
        { cout<<"exception of everything!"<<endl; }
        // 错误：后面的两个异常处理程序段不会被检查
    catch(const char* str)
        { cout<<"exception of:"<<str<<endl; }
    catch(int& e)
        { cout<<"exception of type:"<<e<<endl; }
}
```

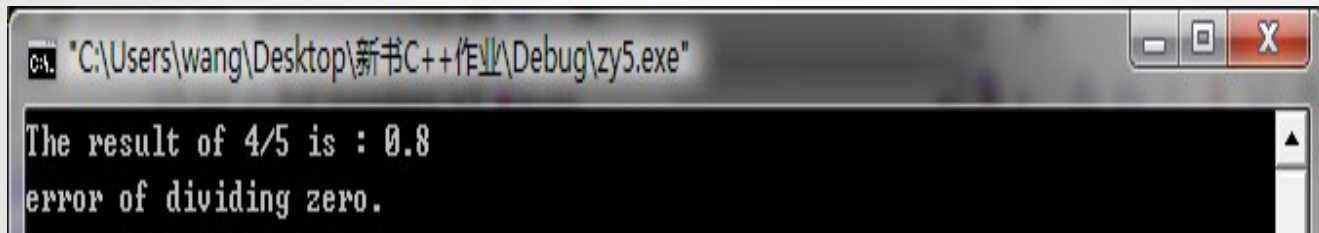
### 【例 9-1】处理除零异常

```
#include <iostream >
using namespace std;
double fun(double a, double b) // 定义除法函数
{
    if(b==0)
    { throw b; } // 除数为 0 , 抛出异常
    return a/b; // 否则返回两个数的商
}

int main()
{
    double res;
    try // 定义异常
    { res=fun(4,5);
      cout<<"The result of" <<4<<"/"<<5<<" is : "<<res<<endl;
      res=fun(6,0); // 出现异常, 函数内部会抛出异常
    }
}
```

```
catch(double)        // 捕获并处理异常
{ cerr<<"error of dividing zero.\n";
  exit(1);           // 异常退出程序
}
return 0;
}
```

程序运行结果：



```

C:\Users\wang\Desktop\新书C++作业\Debug\zy5.exe
The result of 4/5 is : 0.8
error of dividing zero.
```

从运行结果可以看出，当执行 `res=fun(6,0);` 语句时，在函数 `fun()` 中发生除零异常。



异常被抛出后，在 `main()` 函数中被捕获，异常处理程序输出有关信息后，程序流程跳转到主函数的 `catch` 子句，输出“error of dividing zero.”。 `catch` 处理程序的出现顺序很重要，因为在一个 `try` 块中，异常处理程序是按照它出现的顺序被检查的。只要找到一个匹配的异常类型，后面的异常处理都将被忽略。例如，在下面的异常处理块中，首先出现的是 `catch(...)`，它可以捕获任何异常，在任何情况下，其它的 `catch` 语句都不被检查。因此，`catch(...)` 应该放在最后。

### 【例 9-2】异常处理代码的搜索

```
#include <iostream>
using namespace std;
void f3( )
{
    double a=0;
    try
    {throw a;}          // 抛出 double 类型异常信息
    catch(float)
    {cout<<"OK3!"<<endl;}

    cout<<"end3"<<endl;
}
void f2( )
{
    // void f3( );
    try
    { f3( ); }          // 调用 f3( )
    catch(int)
    {cout<<"Ok2!"<<endl;}

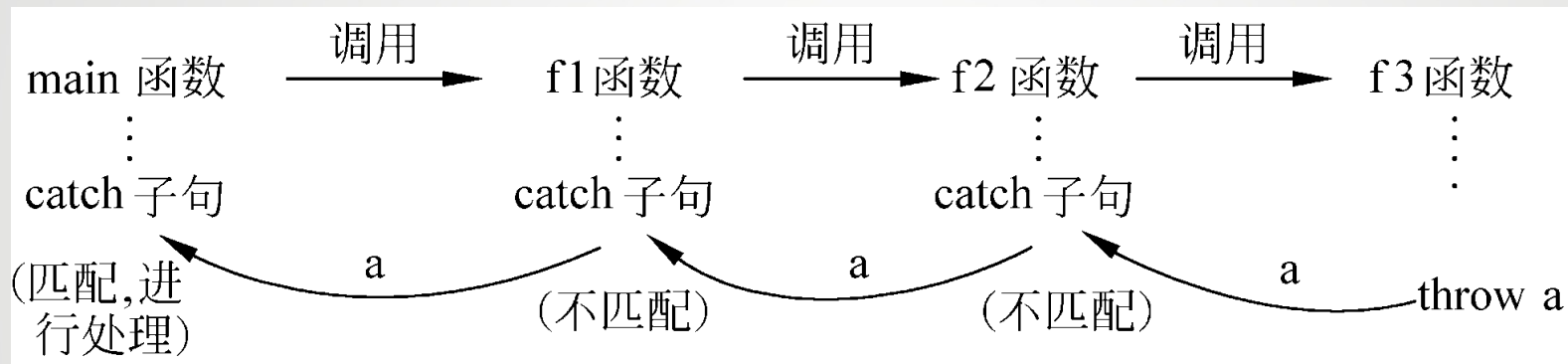
    cout<<"end2"<<endl;
}
```

```
void f1( )
{
    //    void f2( );
    try
    {f2( );}          // 调用 f2( )
    catch(char)
    { cout<<"OK1!"; }
    cout<<"end1"<<endl;
}
```

```
int main( )
{
    //    void f1( );
    try
    {f1( );}          // 调用 f1( )
    catch(double)
    {cout<<"OK0!"<<endl;}
    cout<<"end0"<<endl;
    return 0;
}
```

分3种情况分析运行情况：

(1) 执行上面的程序。图 为有函数嵌套时异常处理示意图。



程序运行结果如下：

OK0! (在主函数中捕获异常)

end0 (执行主函数中最后一个语句时的输出)

(2) 如果将 f3 函数中的 catch 子句改为 catch(double) , 而程序中其他部分不变, 则程序运行结果如下 :

OK3!( 在 f3 函数中捕获异常 )

end3 ( 执行 f3 函数中最后一个语句时的输出 )

end2 ( 执行 f2 函数中最后一个语句时的输出 )

end1 ( 执行 f1 函数中最后一个语句时的输出 )

end0 ( 执行主函数中最后一个语句时的输出 )

(3) 如果在此基础上再将 f3 函数中的 catch 块改为  
catch(double)

```
{cout<<"OK3!"<<endl;throw;}
```

程序运行结果如下 :

OK3!( 在 f3 函数中捕获异常 )

OK0! ( 在主函数中捕获异常 )

end0 ( 执行主函数中最后一个语句时的输出 )



## 例 3. 给出三角形的三边 $a, b, c$ , 求三角形的面



简要说明

只有  $a+b>c, b+c>a, c+a>b$  时才能构成三角形。设置异常处理, 对不符合三角形条件的输出警告信息, 不予计算。 1) 先写出没有异常处理时的程序:

```
#include <iostream>
#include <cmath>
using namespace std;
int main( )
{
    double triangle(double,double,double);
    double a,b,c;
    cin>>a>>b>>c;
    while(a>0 && b>0 && c>0)
    {
        cout<<triangle(a,b,c)<<endl;
        cin>>a>>b>>c;
    }
    return 0;
}
```



## 例 3. 给出三角形的三边 $a, b, c$ , 求三角形的面



简要说明

```
double triangle(double a,double b,double c)
{
    double area;
    double s=(a+b+c)/2;
    area=sqrt(s*(s-a)*(s-b)*(s-c));
    return area;
}
```

### 运行情况如下：

<u>6.54</u> ✓	( 输入 $a, b, c$ 的值 )
9.92157	( 输出三角形的面积 )
<u>11.52</u> ✓	( 输入 $a, b, c$ 的值 )
0.726184	( 输出三角形的面积 )
<u>12.1</u> ✓	( 输入 $a, b, c$ 的值 )
0	( 输出三角形的面积, 此结果显然不对, 因为不是三角形 )
<u>10.6</u> ✓	( 输入 $a, b, c$ 的值 )
( 结束 )	



## 例 3. 给出三角形的三边 $a, b, c$ , 求三角形的面



简要说明

2) 修改程序，在函数 `triangle` 中对三角形条件进行检查，如果不符合三角形条件，就抛出一个异常信息，在主函数中的 `try-catch` 块中调用 `triangle` 函数，检测有无异常信息，并作相应处理。修改后的程序如下：

```
#include <iostream>
#include <cmath>
using namespace std;
void main( )
{
    double triangle(double,double,double);
    double a,b,c;
    cin>>a>>b>>c;
    try          // 在 try 块中包含要检查的函数
    {
        while(a>0 && b>0 && c>0)
        {
            cout<<triangle(a,b,c)<<endl;
            cin>>a>>b>>c;
        }
    }
}
```





## 例 3. 给出三角形的三边 a,b,c , 求三角形的面



简要说明

```
catch(double)           // 用 catch 捕捉异常信息并作相应处理
{cout<<"a="<<a<<" ,b="<<b<<" ,c="<<c<<" ,that is not a triangle!"<<endl;}
cout<<"end"<<endl;
}

double triangle(double a,double b,double c) // 计算三角形的面积的函数
{
    double s=(a+b+c)/2;
    if (a+b<=c || b+c<=a || c+a<=b) throw a; // 当不符合三角形条件抛出异常信息
    return sqrt(s*(s-a)*(s-b)*(s-c));
}
```

程序运行结果如下：

6.54✓ ( 输入 a,b,c 的值 )

9.92157 ( 计算出三角形的面积 )

11.52✓ ( 输入 a,b,c 的值 )

0.726184 ( 计算出三角形的面积 )

12.1✓ ( 输入 a,b,c 的值 )

a=1,b=2,c=1, that is not a triangle! ( 异常处理 )

end



现在结合程序分析怎样进行异常处理：

- (1) 首先把可能出现异常的、需要检查的语句或程序段放在 **try** 后面的花括号中。
- (2) 程序开始运行后，按正常的顺序执行到 **try** 块，开始执行 **try** 块中花括号内的语句。  
如果在执行 **try** 块内的语句过程中没有发生异常，则 **catch** 子句不起作用，流程转到 **catch** 子句后面的语句继续执行。
- (3) 如果在执行 **try** 块内的语句（包括其所调用的函数）过程中发生**异常**，则 **throw** 运算符抛出一个异常信息。**throw** 抛出异常信息后，流程立即离开本函数，转到其上一级的函数（main 函数）。**throw** 抛出什么样的数据由程序设计者自定，可以是**任何类型的数据**。



## 例 3. 分析 2/2

- (4) 这个异常信息提供给 **try-catch 结构**，系统会寻找与之匹配的 **catch 子句**。
- (5) 在进行异常处理后，程序并不会自动终止，继续执行 **catch 子句**后面的语句。

由于 **catch 子句**是用来处理异常信息的，往往被称为 **catch 异常处理块**或 **catch 异常处理器**。

- C++ 语言提供了异常接口声明语法，异常接口声明也称为异常接口声明，利用它可以清晰地告诉使用者异常抛出的类型，异常接口声明再次使用关键字 `throw`，语法如下：

函数返回值类型 函数名（形参列表） `throw`（类型列表）；

例如：

```
double triangle(double,double,double) throw(double);
```

表示 `triangle` 函数只能抛出 `double` 类型的异常信息。如果写成

```
double triangle(double,double,double) throw(int,double,float,char);
```

则表示 `triangle` 函数可以抛出 `int`, `double`, `float` 或 `char` 类型的异常信息。异常指定是函数声明的一部分，必须同时出现在函数声明和函数定义的首行中，否则在进行函数的另一次声明时，编译系统会报告“类型不匹配”。

- 如果在声明函数时未列出可能抛出的异常类型，则该函数可以抛出任何类型的异常信息。如例 1 中第 2 个程序中所表示的那样。
- 如果想声明一个不能抛出异常的函数，可以写成以下形式：

`double triangle(double,double,double) throw();`//throw 无参数

这时即使在函数执行过程中出现了 throw 语句，实际上也并不执行 throw 语句，并不抛出任何异常信息，程序将非正常终止。

# 03



## 构造函数、析构函数与 异常处理

## >>> 1. 异常处理中的构造和析构函数

39

- C++ 异常处理的真正能力不仅在于能处理各种不同类型异常，还在于它具有在异常抛掷前为构造的所有局部对象自动调用析构函数的能力。
- 在程序中，找到一个匹配的 `catch` 异常处理后，如果 `catch` 语句的异常类型声明是一个值参数，则其初始化方式是复制被抛掷的异常对象；如果 `catch` 语句的异常类型声明是一个引用，则其初始化方式是使该引用指向异常对象。
- 当 `catch` 语句的异常类型声明参数被初始化后，栈的展开过程便开始了。这包括从对应的 `try` 块开始到异常被抛掷处之间对构造（且尚未析构）的所有自动对象进行析构。析构的顺序与构造的顺序相反。然后程序从最后一个 `catch` 处理之后开始恢复执行。

## 1. 异常处理中的构造和析构函数

40

构造函数中发生异常后，异常处理遵从以下规则：

- (1) 如果对象有成员函数，且如果在外层对象构造完成之前有异常抛出，则在发生异常之前，执行构造成员对象的析构函数。
- (2) 如果异常发生时，对象数组被部分构造，则只调用已构造的数组元素的析构函数。
- (3) 异常可能跳过通常释放资源的代码，从而造成资源泄漏。解决的方法是，请求资源时初始化一个局部对象，发生异常时，调用析构函数并释放资源。
- (4) 要捕捉析构函数中的异常，可以将调用析构函数的函数放入 try 块，并提供相应类型的 catch 处理程序块。抛出对象的析构函数在异常处理程序执行完毕后执行。



```
#include <iostream>
#include <memory>
#include <string>
using namespace std;

class DemoClass {
public:
    DemoClass(const string objname) : name(objname)
    {
        cout << "construcing DemoClass object..." << endl;
    }
    ~DemoClass()
    {
        cout << "destructing DemoClass object: " << name << endl;
    }
    string who()
    {
        return name;
    }
private:
    string name;
};
```

```
void f()
{
    // 定义一个 auto_ptr 对象, 用该对象指向一个动态创建的 DemoClass 对象
    auto_ptr<DemoClass> dcPtr1(new DemoClass("dcobj"));
    cout << "name of the DemoClass object constructed: "
         << dcPtr1 -> who() << endl;
    // 创建另一个 auto_ptr 对象, 将 dcPtr1 复制给该对象
    auto_ptr<DemoClass> dcPtr2(dcPtr1);
    cout << "name of the DemoClass object to which dcPtr2 points: "
         << (*dcPtr2).who() << endl;
    throw 8;    // 抛出一个 int 型异常
}

int main()
{
    try {
        f();    // 调用有可能产生异常的函数 f
    }
    catch (int) { // 捕获 int 型异常
        cout << "an int exception occurred!" << endl;
    }
    cout << "end of main" << endl;
    return 0;
}
```

程序执行结果：

construcing DemoClass object...

name of the DemoClass object constucted: dcobj

name of the DemoClass object to which dcPtr2 points: dcobj

destructing DemoClass object: dcobj

an int exception occurred!

end of main



用 new 操作创建的  
DemoClass 对象被撤销



## 例 5. 带析构类的异常处理

44

```
#include <iostream.h>
void MyFunc(void);
class Expt
{
public:
    Expt(){ };
    ~Expt(){ };
    const char* ShowReason() const
    {
        return "Expt 类异常。 ";
    }
};
```

## 例 5. 带析构类的异常处理

```
class Demo
{
public:
    Demo() {    cout<<" 构造 Demo 。 "<<endl;    }
    ~Demo(){    cout<<" 析构 Demo 。 "<<endl;    }
};

void MyFunc()
{
    Demo D;
    cout<<" 在 MyFunc() 中抛掷 Expt 类异常。" <<endl;
    throw Expt();
}
```

## 例 5. 带析构类的异常处理

```
void main()
{
    cout<<" 在 main() 函数中。 "<<endl;
    try {
        cout<<" 在 try 块中，调用 MyFunc()。 "<<endl;
        MyFunc();    }
    catch(Expt E)
    {
        cout<<" 在 catch 异常处理程序中。 "<<endl;
        cout<<" 捕获到 Expt 类型异常： ";
        cout<<E.ShowReason()<<endl;
    }
    catch(char* str) {
        cout<<" 捕获到其它的异常： "<<str<<endl;    }
        cout<<" 回到 main() 函数。从这里恢复执行。 "<<endl;
    }
```



## 例 5. 带析构类的异常处理

- 程序运行结果为：  
在 main() 函数中  
在 try 块中，调用 MyFunc()  
构造 Demo  
在 MyFunc() 中抛掷 Expt 类异常  
析构 Demo  
在 catch 异常处理程序中  
捕获到 Expt 类型异常： Expt 类异常  
回到 main() 函数，从这里恢复执行

## 例 5. 带析构类的异常处理

- 注意：

本例中 catch 处理器都有异常参量 (catch 后的参量)

```
catch(Expt E)      { // ... }
```

```
catch(char* str)   { // ... }
```

其实，也可以不说明这些参量 (E 和 str)。在很多情况下，只要通知处理程序有某个特定类型的异常已经产生就足够了。但是在需要访问异常对象时就要说明参量，否则，将无法访问 catch 处理程序语句中的那个对象。例如：

```
catch(Expt)
```

```
{    // 在这里不能访问 Expt 异常对象    }
```



## 例 5. 带析构类的异常处理

- 用不带操作数的 throw 表达式可将当前正被处理的异常再次抛掷；
- 这样的表达式只能出现在一个 catch 处理程序中或 catch 处理程序内部调用的函数中。

再次抛掷的异常对象是源异常对象（不是拷贝）。例如：

```
try
{
    throwCSomeOtherException();
}
catch(...) // 处理所有异常
{
    // 对异常作出响应（也许仅仅是部分的）
    //...
    throw; // 将异常传给某个其它处理器
}
```



## 异常处理中的构造和析构函数

说明：

- (1) 被检测的函数必须放在 try 块中，否则不起作用。
- (2) try 块和 catch 块作为一个整体出现，catch 块是 try-catch 结构中的一部分，必须紧跟在 try 块之后，不能单独使用，在二者之间也不能插入其他语句。但是在一个 try-catch 结构中，可以只有 try 块而无 catch 块。即在本函数中只检查而不处理，把 catch 处理块放在其他函数中。
- (3) try 和 catch 块中必须有用花括号括起来的复合语句，即使花括号内只有一个语句，也不能省略花括号。
- (4) 一个 try-catch 结构中只能有一个 try 块，但却可以有多个 catch 块，以便与不同的异常信息匹配。

## 1. 异常处理中的构造和析构函数

(5) `catch` 后面的圆括号中，一般只写异常信息的类型名， 如

`catch(double)`

`catch` 只检查所捕获异常信息的类型，而不检查它们的值。因此如果需要检测多个不同的异常信息，应当由 `throw` 抛出不同类型的异常信息。

异常信息可以是 C++ 系统预定义的标准类型，也可以是用户自定义的类型（如结构体或类）。如果由 `throw` 抛出的信息属于该类型或其子类型，则 `catch` 与 `throw` 二者匹配，`catch` 捕获该异常信息。

`catch` 还可以有另外一种写法，即除了指定类型名外，还指定变量名，如

`catch(double d)`

## 1. 异常处理中的构造和析构函数

此时如果 **throw** 抛出的异常信息是 `double` 型的变量 `a`，则 **catch** 在捕获异常信息 `a` 的同时，还使 `d` 获得 `a` 的值，或者说 `d` 得到 `a` 的一个拷贝。什么时候需要这样做呢？有时希望在捕获异常信息时，还能利用 **throw** 抛出的值，如

**catch(double d)**

```
{cout<<"throw "<<d;}
```

这时会输出 `d` 的值（也就是 `a` 值）。当抛出的是类对象时，有时希望在 `catch` 块中显示该对象中的某些信息。这时就需要在 `catch` 的参数中写出变量名（类对象名）。

(6) 如果在 **catch** 子句中没有指定异常信息的类型，而用了删节号“...”，则表示它可以捕捉任何类型的异常信息，如

# 1. 异常处理中的构造和析构函数

`catch(...)`

```
{cout<<"OK"<<endl;}
```

它能捕捉所有类型的异常信息，并输出“OK”。

这种 `catch` 子句应放在 `try?catch` 结构中的最后，相当于“其他”。如果把它作为第一个 `catch` 子句，则后面的 `catch` 子句都不起作用。

(7) `try?catch` 结构可以与 `throw` 出现在同一个函数中，也可以不在同一函数中。当 `throw` 抛出异常信息后，首先在本函数中寻找与之匹配的 `catch`，如果在本函数中无 `try?catch` 结构或找不到与之匹配的 `catch`，就转到离开出现异常最近的 `try?catch` 结构去处理。

## 1. 异常处理中的构造和析构函数

(8) 在某些情况下，在 `throw` 语句中可以不包括表达式，如

`throw;`

表示“我不处理这个异常，请上级处理”。

(9) 如果 `throw` 抛出的异常信息找不到与之匹配的 `catch` 块，那么系统就会调用一个系统函数 `terminate`，使程序终止运行。

## 2. 在异常处理中处理析构函数

如果在 **try 块** ( 或 try 块中调用的函数 ) 中定义了类对象，在建立该对象时要调用构造函数。在执行 **try 块** ( 包括在 try 块中调用其他函数 ) 的过程中如果发生了异常，此时流程立即离开 try 块。这样流程就有可能离开该对象的作用域而转到其他函数，因而应当事先做好结束对象前的清理工作，C++ 的异常处理机制会在 **throw** 抛出异常信息被 **catch** 捕获时，对有关的局部对象进行析构 ( 调用类对象的析构函数 )，析构对象的顺序与构造的顺序相反，然后执行与异常信息匹配的 **catch 块** 中的语句。

## 例 6. 在异常处理中处理析构函数

这是一个为说明在异常处理中调用析构函数的示例，为了清晰地表示流程，程序中加入了一些 cout 语句，输出有关的信息，以便对照结果分析程序。

```
#include <iostream.h>
#include <string.h>
class Student
{
public:
    Student(int n,char* nam) {
        cout<<"constructor-"<<n<<endl;
        num=n;name=nam;
    }
    ~Student() {cout<<"destructor-"<<num<<endl;} // 定义析构函数
    void get_data( ); // 成员函数声明
private:
    int num;
    char* name;
};
```



## 例 6. 在异常处理中处理析构函数

```
void Student::get_data( )           // 定义成员函数
{
    if(num==0)
        throw num;                 // 如 num=0, 抛出 int 型变量 num
    else
        cout<<num<<" "<<name<<endl;    // 若 num≠0 , 输出 num,name
        cout<<"in get_data()"<<endl;    // 输出信息, 表示目前在 get_data 函数中
}

void fun( )
{
    Student stud1(1101,"Tan");      // 建立对象 stud1
    stud1.get_data( );              // 调用 stud1 的 get_data 函数
    Student stud2(0,"Li");          // 建立对象 stud2
    stud2.get_data( );              // 调用 stud2 的 get_data 函数
}
```

## 例 6. 在异常处理中处理析构函数

```
int main( )
{
    cout<<"main begin"<<endl;           // 表示主函数开始了
    cout<<"call fun( )"<<endl;          // 表示调用 fun 函数
    try
    {
        fun( );
    } // 调用 fun 函数
    catch(int n)
    {
        cout<<"num="<<n<<"error!"<<endl;
    } // 表示 num=0 出错
    cout<<"main end"<<endl;             // 表示主函数结束
    return 0;
}
```

## 例 6. 在异常处理中处理析构函数

### 程序运行结果如下：

main begin

call fun( )

constructor-1101

1101 tan

in get\_data()

constructor-0

destructor-0

destructor-1101

num=0,error!

main end

# 04



## 异常匹配

从基类可以派生各种异常类，当一个异常抛出时，异常处理器会根据异常处理顺序找到“最近”的异常类型进行处理。如果 `catch` 捕获了一个指向基类类型异常对象的指针或引用，那么它也可以捕获该基类所派生的异常对象的指针或引用。相关错误的多态处理是允许的。

# 05



## 标准异常及层次结构

- C++ 标准提供了标准库异常及层次结构。标准异常以基类 `exception` 开头 ( 在头文件 `<exception>` 中定义 )，该基类提供了函数 `what()`，每个派生类中重定义发出相应的错误信息。
- 由基类 `exception` 直接派生的类 `runtime_error` 和 `logic_error`( 均定义在头文件 `<stdexcept>` 中 )，分别报告程序的逻辑错误和运行时错误信息。
- I/O 流异常类 `ios::failure` 也由 `exception` 类派生而来。

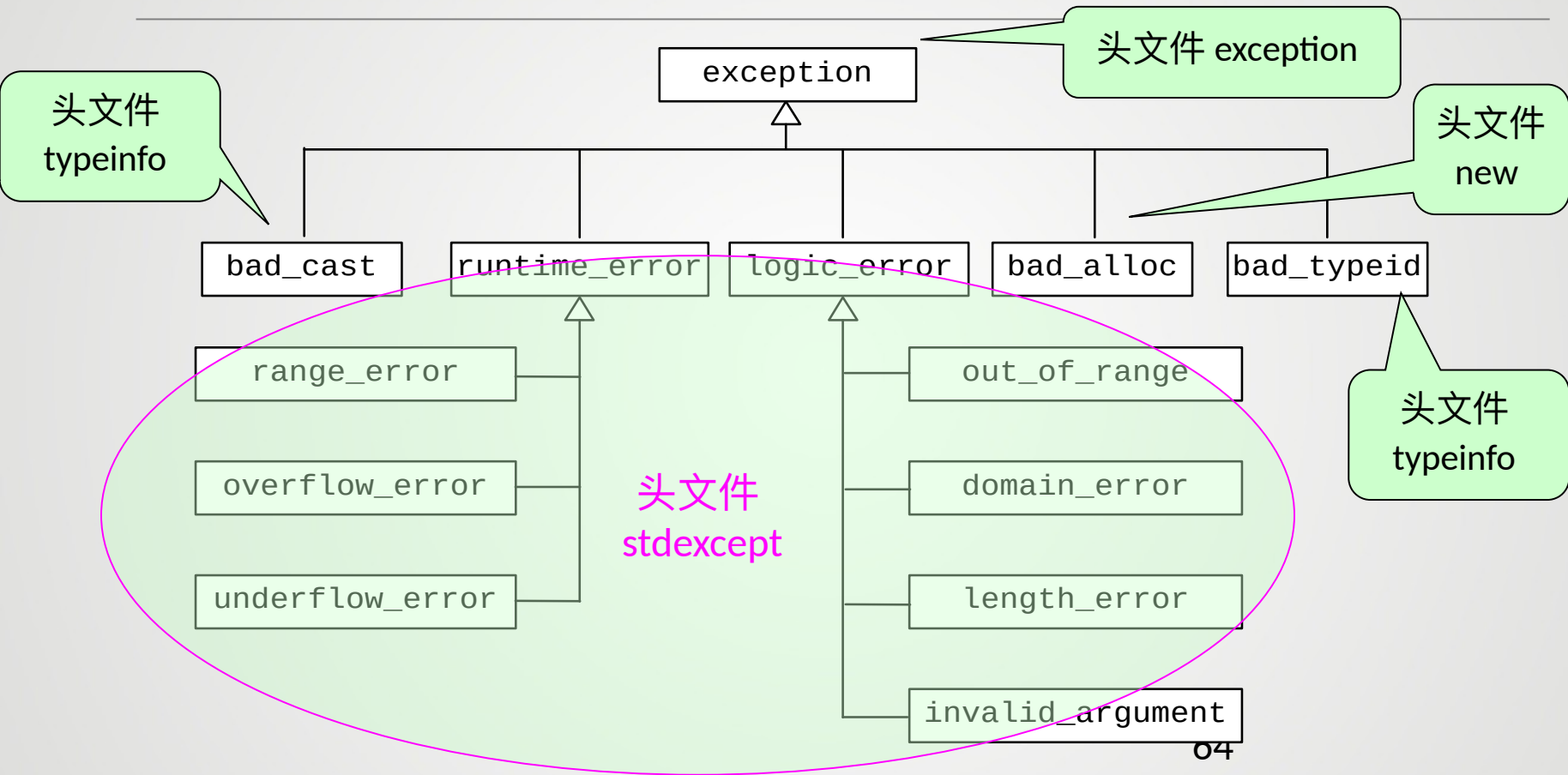
注意：

异常处理不能用于处理异步情况，如磁盘 I/O 完成、网络消息到达、鼠标单击等。



# • 用于报告标准库中的函数和类在程序运行时产生的异常

64







- exception 类位于 <exception> 头文件中，它被声明为：

```
class exception
{
public:
    exception () throw(); // 构造函数
    exception (const exception&) throw(); // 拷贝构造函数
    exception& operator= (const exception&) throw(); // 运算符重载
    virtual ~exception() throw(); // 虚析构函数
    virtual const char* what() const throw(); // 虚函数
}
```

- 这里需要说明的是 what() 函数。what() 函数返回一个能识别异常的字符串，正如它的名字“what”一样，可以粗略地告诉你这是什么异常。不过 C++ 标准并没有规定这个字符串的格式，各个编译器的实现也不同，所以 what() 的返回值仅供参考。

- exception 类的直接派生类：

异常名称	说 明
logic_error	逻辑错误。
runtime_error	运行时错误。
bad_alloc	使用 new 或 new[] 分配内存失败时抛出的异常。
bad_typeid	使用 typeid 操作一个 NULL 指针，而且该指针是带有虚函数的类，这时抛出 bad_typeid 异常。
bad_cast	使用 dynamic_cast 转换失败时抛出的异常。
ios_base::failure	io 过程中出现的异常。
bad_exception	这是个特殊的异常，如果函数的异常列表里声明了 bad_exception 异常，当函数内部抛出了异常列表中没有的异常时，如果调用的 unexpected() 函数中抛出了异常，不论什么

- `logic_error` 的派生类：

异常名称	说 明
<code>length_error</code>	试图生成一个超出该类型最大长度的对象时抛出该异常，例如 <code>vector</code> 的 <code>resize</code> 操作。
<code>domain_error</code>	参数的值域错误，主要用在数学函数中，例如使用一个负值调用只能操作非负数的函数。
<code>out_of_range</code>	超出有效范围。
<code>invalid_argument</code>	参数不合适。在标准库中，当利用 <code>string</code> 对象构造 <code>bitset</code> 时，而 <code>string</code> 中的字符不是 0 或 1 的时候，抛出该异常。



- `runtime_error` 的派生类：

异常名称	说 明
<code>range_error</code>	计算结果超出了有意义的值域范围。
<code>overflow_error</code>	算术计算上溢。
<code>underflow_error</code>	算术计算下溢。



- 作用：在函数原型中对函数是否会抛出异常以及会抛出何种



- 作用：在函数原型中对函数是否会抛出异常以及会抛出何种



- 作用：在函数原型中对函数是否会抛出异常以及会抛出何种



- 作用：在函数原型中对函数是否会抛出异常以及会抛出何种





- 作用：在函数原型中对函数是否会抛出异常以及会抛出何种

- 作用：在函数原型中对函数是否会抛出异常以及会抛出何种异常进行说明，帮助程序员了解标准库或第三方库中的函数（或类操作）所抛出异常的类型
- 形式：  
    `throw` （异常类型列表）  
    放于函数原型中形参表的后面  
    — 例如：  
        `int f1 () throw (logic_error);`
- 异常类型列表为空，表示该函数不抛出任何异常
- 不带异常说明的函数可以抛出任意类型的异常
- `const` 成员函数的异常说明放在保留字 `const` 之后
- 基类虚函数的异常列表是派生类中对应虚函数的异常列表的超集

- C++ 中异常处理的目标是简化大型可靠程序的创建，用尽可能少的代码，使系统中没有不受控制的错误。
- 异常处理设计用来处理同步情况，作为程序执行的结构，而不能用于处理异步情况；异常处理通常用于发现错误部分与处理错误部分处于不同位置（不同范围）时；异常处理不应作为具体的控制流机制。

