第17章　异 常 处 理

错误修复技术的改进是提高代码健壮性的最有效方法之一。

但是,大多数程序设计人员在实际设计中往往忽略出错处理 ,似乎是在没有错误的状态下编程。毫无疑问，出错处理的繁琐及错误检查引起的代码膨胀是导致上述问题的主要原因。例如，虽然printf( )函数可返回打印参数的个数，但是实际程序设计中没有人检查该值。出错处理引起的代码膨胀将不可避免地增加程序阅读的困难，这对于程序设计人员来说是十分令人烦恼的。

C语言中实现出错处理的方法是将用户函数与出错处理程序紧密地结合起来，但是这将造成出错处理使用的不方便和难以接受。

异常处理是C++语言的一个主要特征，它提出了出错处理更加完美的方法。

1) 出错处理程序的编写不再繁琐，也不须将出错处理程序与“ 通常 ”代码紧密结合。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出错处理程序。如果程序中多次调用一个函数，在程序中加入一个函数出错处理程序即可。

2) 错误发生是不会被忽略的。如果被调用函数需发送一条出错信息给调用函数，它可向调用函数发送一描述出错信息的对象。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该出错信息的对象，直到该出错信息被捕捉和处理。

在这一章中我们将讨论C语言的出错处理方法，讨论为何该方法在C语言中不是很理想的，并且无法在C++中使用；然后学习try，throw和catch的用法，它们在C++中支持异常处理。

17.1　C语言的出错处理

本书在第8章以前使用C标准库的assert( )宏作为出错处理的方法。第8章以后assert( )被按照原先的设计目的使用：在开发过程中，使用它们，完成后用 #define NDEBUG使之失效，以便推出产品。为了在运行时检查错误， assert( )被allege( )函数和第8章中引入的宏所取代。通常我们会说：“ 对于出错处理我们必须面对复杂的代码，但是在这个例子中我们不必由此感到烦恼 ”。allege( )函数对一些小型程序很方便，对于复杂的大型程序，所编写的出错处理程序也将更加复杂。

在通过检查条件我们能确切地知道做什么的情况下，出错处理就变得十分明确和容易了，因为我们通过上下文得到了所有必要的信息。当然，我们只是在这一点上处理错误。这些都是十分普通的错误，不是这一章的主题。

若错误问题发生时在一定的上下文环境中得不到足够的信息，则需要从更大的上下文环境中提取出错处理信息，下面给出了C语言处理这类情况的三种典型方法。

1) 出错信息可通过函数的返回值获得。如果函数返回值不能用，则可设置一全局错误判断标志（标准C语言中errno( )和perror( )函数支持这一方法）。正如前文提到的，由于对每个函数调用都进行错误检查，这十分繁琐并增加了程序的混乱度。程序设计者可能简单地忽略这些出错信息，因为乏味而迷乱的错误检查必须随着每个函数调用而出现。另外，来自偶然出现异常的函数的返回值可能并不反映什么问题。

2) 可使用C标准库中一般不太熟悉的信号处理系统，利用signal()函数（判断事件发生的类型）和raise()函数（产生事件）。由于信号产生库的使用者必须理解和安装合适的信号处理系统，所以应紧密结合各信号产生库，但对于大型项目，不同库之间的信号可能会产生冲突。

3) 使用C标准库中非局部的跳转函数：setjmp( ) 和 longjmp( )。setjmp( ) 函数可在程序中存储一典型的正常状态，如果进入错误状态， longjmp( )可恢复setjmp( ) 函数的设定状态，并且状态被恢复时的存储地点与错误的发生地点紧密联系。

考虑C++语言的出错处理方案时会存在另一个关键性问题：由于 C语言的信号处理技术和setjmp/longjmp函数不能调用析构函数，所以对象不能被正确地清除。由于对象不能被清除，它将被保留下来并且将不能再次被存取，所以存在这种问题时实际上是不可能有效正确地从异常情况中恢复出来。下面的例子将演示setjmp/longjmp的这一特点：

```
//: NONLOCAL.CPP -- setjmp & longjmp
#include <iostream.h>
#include <setjmp.h>

class rainbow {
public:
  rainbow() { cout << "rainbow()" << endl; }
  ~rainbow() { cout << "~rainbow()" << endl; }
};

jmp_buf kansas;

void OZ() {
  rainbow RB;
  for(int i = 0; i < 3; i++)
    cout << "there's no place like home\n";
  longjmp(kansas, 47);
}

main() {
  if(setjmp(kansas) == 0) {
    cout << "tornado, witch, munchkins...\n";
    OZ();
  } else {
    cout << "Auntie Em! "
         << "I had the strangest dream..."
         << endl;
  }
}
```

setjmp()是一个特别的函数，因为如果我们直接调用它，它就把当前进程状态的所有相关信息存放在jmp_buf中，并返回零。这样，它的行为象通常的函数。然而，如果使用同一个jmp_buf调用longjmp()，这就象再次从setjmp()返回，即正确地弹出setjmp()的后端。这时，返回值对于longjmp()是第二个参数，所以能发现实际上从longjmp()中返回了。可以想象，有多个

不同的jmp_buf，可以弹出程序的多个不同位置的信息。局部 goto（用标号）和这个非局部跳转的不同在于我们能通过setjmp/longjmp跳转到任何地方（一些限制不在这里讨论）。

在C++中的问题是，longjmp()不适用于对象，特别是，当它跳出范围时它不调用析构函数[1]。析构函数调用是必须的，所以这种方法在C++中不可行。

17.2 抛出异常

如果程序发生异常情况，而在当前的上下文环境中获取不到异常处理的足够信息，我们可以创建一包含出错信息的对象并将该对象抛出当前上下文环境，将错误信息发送到更大的上下文环境中。这称为异常抛出。如：

```
throw myerror ("something bad happened");
```

myerror是一个普通类，它以字符变量作为其参数。当进行异常抛出时我们可使用任意类型变量作为其参数（包括内部类型变量），但更为常用的办法是创建一个新类用于异常抛出。

关键字throw的引入引起了一系列重要的相关事件发生。首先是 throw调用构造函数创建一个原执行程序中并不存在的对象。其次，实际上这个对象正是 throw函数的返回值，即使这个对象的类型不是函数设计的正常返回类型。对于交替返回机制，如果类推太多有可能会陷入困境，但仍可看作是异常处理的一种简单方法，可通过抛出一个异常来退出普通作用域并返回一个值。

因为异常抛出同常规函数调用的返回地点完全不同，所以返回值同普通函数调用具有很小的相似性（异常处理器地点与异常抛出地点可能相差很远）。另外，只有在异常时刻成功创建的对象才被清除掉。（常规函数调用则不同，它使作用域内的所有对象均被清除。）当然，异常情况产生的对象本身在适当的地点也被清除。

另外，我们可根据要求抛出许多不同类型的对象。一般情况下，对于每种不同的错误可设定抛出不同类型的对象。采用这样的方法是为了存储对象中的信息和对象的类型，所以别人可以在更大的上下文环境中考虑如何处理我们的异常。

17.3 异常捕获

如果一个函数抛出一个异常，它必须假定该异常能被捕获和处理。正如前文所提到的，允许对一个问题集中在一处解决，然后处理在别处的差错，这也正是 C++语言异常处理的一个优点。

17.3.1 try块

如果在函数内抛出一个异常（或在函数调用时抛出一个异常），将在异常抛出时退出函数。如果不想在异常抛出时退出函数，可在函数内创建一个特殊块用于解决实际程序中的问题（和潜在产生的差错）。由于可通过它测试各种函数的调用，所以被称为测试块。测试块为普通作用域，由关键字try引导：

```
try {
    // code that may generate exceptions
}
```

[1] 当我们运行这个例子时会惊奇地发现 —— 一些C++编译器调用 longjmp( )函数清除堆栈中的对象。这是兼容性差的问题。

如果没有使用异常处理而是通过差错检查来探测错误，即使多次调用同一个函数，也不得不围绕每个调用函数重复进行设置和代码检测。而使用异常处理时不需做差错检查，可将所有的工作放入测试块中。这意味着程序不会由于差错检查的引入而变得混乱，从而使得程序更加容易编写，其可读性也大为改善。

17.3.2 异常处理器

异常抛出信号发出后，一旦被异常器处理接收到就被销毁。异常处理器应具备接受任何一种类型的异常的能力。异常处理器紧随try块之后，处理的方法由关键字catch引导。

```
try {
// code that may generate exceptions
} catch(type1 id1) {
  // handle exceptions of type1
} catch(type2 id2) {
  // handle exceptions of type2
}
// etc...
```

每一个catch语句（在异常处理器中）就相当于一个以特殊类型作为单一参数的小型函数。异常处理器中标识符（id1、id2 等）就如同函数中的一个参数。如果异常抛出给出的异常类型足以判断如何进行异常处理，则异常处理器中的标识符可省略。

异常处理部分必须直接放在测试块之后。如果一个异常信号被抛出，异常处理器中第一个参数与异常抛出对象相匹配的函数将捕获该异常信号，然后进入相应的 catch语句，执行异常处理程序。catch语句与switch语句不同，它不需要在每个case语句后加入break用以中断后面程序的执行。

注意，在测试块中不同的函数的调用可能会产生相同的异常情况，但是，这时只需要一个异常处理器。

• 终止与恢复

在异常处理原理中含有两个基本模式：终止与恢复。假设差错是致命性的，当异常发生后将无法返回原程序的正常运行部分，这时必须调用终止模式（ C++支持）结束异常状态。无论程序的哪个部分只要发生异常抛出，就表明程序运行进入了无法挽救的困境，应结束运行的非正常状态，而不应返回异常抛出之处。

另一个为恢复部分。恢复意味着希望异常处理器能够修改状态，然后再次对错误函数进行检测，使之在第二次调用时能够成功运行。如果要求程序具有恢复功能，就希望程序在异常处理后仍能继续正常执行程序，这样，异常处理就更象一个函数调用 —— C ++程序中在需要进行恢复的地方如何设置状态（换言之就是使用函数调用，而非异常抛出来解决问题）。另外也可将测试块放入while循环中，以便始终装入测试块直到恢复成功得到满意的结果。

过去，程序员们使用的支持恢复性异常处理的操作系统最终被终止性模式所取代，它取消了恢复性模式。所以虽然恢复性模式初听起来是十分吸引人的，但在实际运用中却并非十分有效。其中一个原因可能是异常发生与异常处理相距较远的缘故。要终止相距较远的异常处理器，但是由于异常可能由很多地点产生，所以对于一个大型系统，从异常处跳转到异常处理器再跳转返回，这在概念上是十分困难的。

17.3.3 异常规格说明

可以不向函数使用者给出所有可能抛出的异常，但是这一般被认为是非常不友好的，因为这意味着他无法知道该如何编写程序来捕获所有潜在的异常情况。当然，如果他有源程序，他可寻找异常抛出的说明，但是库通常不以源代码方式提供。C++语言提供了异常规格说明语法，我们以可利用它清晰地告诉使用者函数抛出的异常的类型，这样使用者就可方便地进行异常处理。这就是异常规格说明，它存在于函数说明中，位于参数列表之后。

异常规格说明再次使用了关键字 throw，函数的所有潜在异常类型均随着关键字 throw而插入函数说明中。所以函数说明可以带有异常说明如下：

```
void  f ( )  throw ( toobig, toosmall, divzero) ;
```

而传统函数声明：

```
void  f ( );
```

意味着函数可能抛出任何一种异常。

如果是：

```
void  f ( )  throw ( );
```

这意味着函数不会有异常抛出。

为了得到好的程序方案和文件，为了方便函数调用者，每当写一个有异常抛出的函数时都应当加入异常规格说明。

1. unexpected( )

如果函数实际抛出的异常类型与我们的异常规格说明不一致，将会产生什么样的结果呢？这时会调用特殊函数unexpected( )。

2. set_unexpected( )

unexpected( )是使用指向函数的指针而实现的，所以我们可通过改变指针的指向地址来改变相对应的运算。这些可通过类似于 set_new_handler( )的函数set_unexpected( )来实现，set_unexpected( ) 函数可获取不带输入和输出参数的函数地址和 void返回值。它还返回unexpected指针的先前值，这样我们可存储unexpected( )函数的原先指针值，并在后面恢复它。为了使用set_unexpected( )函数，我们必须包含头文件EXCEPT.H。下面给出一实例展示本章所讨论的各个特点的简单使用：

```
//: EXCEPT.CPP -- Basic exceptions
// Exception specifications & unexpected()
#include <except.h>
#include <iostream.h>
#include <stdlib.h>
#include <string.h>

class up {};
class fit {};
void g();

void f(int i) throw (up, fit) {
  switch(i) {
    case 1: throw up();
    case 2: throw fit();
```

```
  }
  g();
}

// void g() {} // version 1
void g() { throw 47; } // Version 2
// (can throw built-in types)

void my_unexpected() {
  cout << "unexpected exception thrown";
  exit(1);
}

main() {
  set_unexpected(my_unexpected);
  // (ignores return value)
  for(int i = 1; i <=3; i++)
    try {
      f(i);
    } catch(up) {
      cout << "up caught" << endl;
    } catch(fit) {
      cout << "fit caught" << endl;
    }
}
```

作为异常抛出类，up 和fit分别被创建。通常异常类均是小型的，但有时它们包含许多额外信息，这样异常处理器可通过查询它们来获得辅助信息。

f( )函数在它的异常规格说明中声明函数的异常抛出只能是类 up 和 fit，并且函数体的定义同函数的异常规格说明是一致的。函数 g( ) （version1）被函数f( )调用，但并不抛出异常，因此这也是可行的。当函数g( ) （version1）被修改以后得g( ) （version2），g( ) （version2）仍是f( )的调用函数，但其具有异常抛出功能。函数g( )修改以后f( )函数具有了新的异常抛出，但最初创建的f( )函数对于这些却未加声明，这样就违反了异常规格说明。

my_unexpected( )函数可以没有输入或输出参数， 它是按照定制的unexpected( )函数的正确格式编写的。它仅仅打出一条有关异常的信息就退出，所以一旦被调用，我们就可以观察到这条信息。新函数unexpected( )不必有返回值（可以按照这种方法编写程序，但这是错误的）。然而它却可抛出另一个异常（也可使它抛出同一个异常），或者调用函数exit( )或abort( )。如果函数unexpected( )抛出一个异常，异常处理器将在异常抛出时开始搜寻 unexcepted异常。（这种特点对于unexcepted （）来说是独特的）

虽然new_handler( )函数的指针可为空，但unexpected( )函数的指针却不能为空。它的缺省值指向terminate( )（后面将会介绍）函数，但是，只要我们使用异常抛出和异常规格说明，我们就应该编写自己的unexpected( )函数，用于记录或者再次抛出异常及抛出新的异常或终止程序运行。

在主程序中，为了对所有的潜在异常进行检测，测试块被放入 for循环中。注意这里提到的

实现方法很象前文介绍的恢复模式，将测试块放入 for, while, do 或 if 的循环语句中，并利用每一个异常来试图消除差错问题；然后再一次的调用测试块对潜在异常进行检测。

由于程序中f( ) 的函数声明引入了up和fit两类异常，因此只有该两类异常可被抛出。因为f( ) 的函数声以后要抛出的整型，所以修改后的 g( )（version2）会使得函数my_unexpected( ) 被调用。（我们可使用任意的异常类型，包括内部类型。）

函数set_unexcepted( )被调用后，它的返回值可被忽略，但也可以被保存为函数指针，并在随后用于恢复 unexcepted( )的原先指针。

17.3.4 更好的异常规格说明

我们可能觉得在前面介绍的已存在的异常规格说明规则并非十分可靠，并且

```
void f ( ) ;
```

应该意味着函数没有异常抛出，但按照前面的规则这正好相反，它表示可抛出任意类型的异常。如果程序员要抛出任意类型的异常，我们可能会想他应该说明如下

```
void  f ( )  throw (. . .) ;  //  not  in  C++
```

因为函数声明应当更加清晰，所以这是一个改进。但不幸的是，我不能总是通过查看程序代码来知道函数是否有异常抛出 —— 例如，函数的异常抛出发生在存储分配过程中。较为糟糕的是由于调用了在异常处理之前引入的函数而出现非有意的异常抛出。（函数可能与一个新版本的异常抛出相连接）所以采用不明确的描述，如：

```
void   f ( ) ;
```

表示有可能有异常抛出，也可能没有。这种不明确的描述对于避免阻碍程序执行是十分必要的。

17.3.5 捕获所有异常

前面论述过，如果函数没有异常规格说明，任何类型的异常都有可能被函数抛出。为了解决这个问题，应创建一个能捕获任意类型的异常的处理器。这可以通过将省略号加入参数列表(á la C)中来实现这一方案。

```
catch (. . . ) {
    cout <<  "an exception was thrown" <<endl;
}
```

为了避免漏掉异常抛出，可将能捕获任意异常的处理器放在一系列处理器之后。

在参数列表中加入省略号可捕获所有的异常，但使用省略号就不可能有参数，也不可能知道所接受到的异常为何种类型。

17.3.6 异常的重新抛出

有时需要重新抛出刚接收到的异常，尤其是在我们无法得到有关异常的信息而用省略号捕获任意的异常时。这些工作通过加入不带参数的 throw就可完成：

```
catch (. . .) {
    cout <<  "an exception was thrown "<<endl;
    throw;
}
```

如果一个catch句子忽略了一个异常，那么这个异常将进入更高层的上下文环境。由于每个异常抛出的对象是被保留的，所以更高层上下文环境的处理器可从抛出来自这个对象的所有信息。

17.3.7 未被捕获的异常

如果测试块后面的异常处理器没有与某一异常相匹配，这时内层对异常的捕获失败，异常将进入更高层的上下文环境中（高层测试块一般不最先进行异常接收），这个过程一直进行直到在某个层次异常处理器与该异常相匹配，这时这个异常才被认为是被捕获了，进一步的查询也将停止。

假如任意层的处理器都没有捕获到这个异常，那么这个异常就是“未捕获的”或“未处理的”。如果已存在的异常在被捕获之前又有一个新的异常产生将造成异常不能被获取，最常见的这种情况的产生原因是异常对象的构造函数自身会导致新的异常。

1. terminate( )

如果异常未能被捕获，特殊函数terminate( )将自动被调用。如同函数unexception( )终止函数一样，它实际上也是一个指向函数的指针。在C标准库中它的缺省值为指向函数abort( )的指针，abort( )函数可以不用调用正常的终止函数而直接从程序中退出（这意味着静态全局函数的析构函数不用被调用）。

如果一个异常未被捕获，析构函数不会被调用，则异常将不会被清除。含有未捕获的异常将被认为是程序错误。我们可将程序（如果有必要，包括main( )的所有代码）封装在一个测试块中，这个测试块由各异常处理器按序组成，并可以捕获任意异常的缺省处理器(catch(. . .))结束。如果我们不将程序按上述方法封装，将使我们的程序十分臃肿。一个未能被捕获的异常可看成是一个程序错误。

2. set_terminate( )

我们可以使用标准函数set_ terminate( )来安装自己的终止函数terminate( )，set_ terminate( )返回被替代的terminate( )函数的指针，这样就可存贮该指针并在需要时进行恢复。定做的终止函数terminate( )必须不含有输入参数，其返回值为 void。另外所安装的任何终止处理器terminate( )必须不返回或抛出异常，但是作为替换将调用一些程序终止函数。在实际中如果函数terminate( )被调用就意味着问题将无法被恢复。

如同函数unexpected( )一样，函数terminate( )的指针不能为零。

这儿给出一实例用以展示set_ terminate( )的使用。例中set_ terminate( )函数被调用后返回函数terminate( )的原先的指针，存储该指针并为以后的恢复做准备，这样可通过函数terminate( )为判断未捕获的异常在程序中何处发生提供帮助：

```
//: TRMNATOR.CPP -- Use of set_terminate()
// Also shows uncaught exceptions
#include <except.h>
#include <iostream.h>
#include <stdlib.h>

void terminator() {
  cout << "I'll be back!" << endl;
  abort();
}
```

```
void (*old_terminate)()
  = set_terminate(terminator);

class botch {
public:
  class fruit {};
  void f() {
    cout << "botch::f()" << endl;
    throw fruit();
  }
  ~botch() { throw 'c'; }
};

main() {
  try{
    botch b;
    b.f();
  } catch(...) {
    cout << "inside catch(...)" << endl;
  }
}
```

old_terminate的定义初看上去有些令人费解：该语句不仅创建了一个指向函数的指针old_terminate，而且将其初始化为函数set_ terminate( )的返回值。虽然我们可能比较熟悉在函数指针后面加分号的定义方法，但例中所给出是另一种变量并可在定义时进行初始化。

类botch不仅在函数f( )内部会抛出异常，而且在它的析构函数内也会抛出异常。从主程序中可见，这是调用函数terminate( )的一种情况。虽然异常处理器中使用了catch(...)函数，从表面上看它似乎可以捕获所有的异常，避免函数terminate( )的调用，但是当处理一个异常需清除堆栈中的对象时，在这一过程中将调用类botch的析构函数，由此产生了第二个异常，这将迫使函数terminate( )被调用。因此析构函数中含有异常抛出或引起异常抛出都将是一个设计错误。

17.4 清除

异常处理的部分难度就在于异常抛出时从正常程序流转入异常处理器中。如果异常抛出时对象没有被正确地清除，这一操作将不会很有效。 C++的异常处理器可以保证当我们离开一个作用域时，该作用域中所有结构完整的对象的析构函数都将被调用，以清除这些对象。

这里给出一个例子，用以演示当对象的构造函数不完整时其析构函数将不被调用，它也用来展示如果在被创建对象过程中发生异常抛出时将出现什么结果，如果 unexpected( )函数再次抛出意外的异常时将出现什么结果：

```
//: CLEANUP.CPP -- Exceptions clean up objects
#include <fstream.h>
#include <except.h>
#include <string.h>

ofstream out("cleanup.out");
```

```
class noisy {
  static int i;
  int objnum;
  enum { sz = 40 };
  char name[sz];
public:
  noisy(const char* nm="array elem") throw(int){
    objnum = i++;
    memset(name, 0, sz);
    strncpy(name, nm, sz - 1);
    out << "constructing noisy " << objnum
      << " name [" << name << "]" << endl;
    if(objnum == 5) throw int(5);
    // Not in exception specification:
    if(*nm == 'z') throw char('z');
  }
  ~noisy() {
    out << "destructing noisy " << objnum
      << " name [" << name << "]" << endl;
  }
  void* operator new[](size_t sz) {
    out << "noisy::new[]" << endl;
    return ::new char[sz];
  }
  void operator delete[](void* p) {
    out << "noisy::delete[]" << endl;
    ::delete []p;
  }
};

int noisy::i = 0;

void unexpected_rethrow() {
  out << "inside unexpected_rethrow()" << endl;
  throw; // Rethrow same exception
}
main() {
  set_unexpected(unexpected_rethrow);
  try {
    noisy n1("before array");
    // Throws exception:
    noisy* array = new noisy[7];
    noisy n2("after array");
  } catch(int i) {
    out << "caught " << i << endl;
  }
```

```
  out << "testing unexpected:" << endl;
  try {
    noisy n3("before unexpected");
    noisy n4("z");
    noisy n5("after unexpected");
  } catch(char c) {
    out << "caught " << c << endl;
  }
}.
```

类noisy可跟踪对象的创建，所以可通过它跟踪程序的运行。类 noisy中含有静态整数变量i用以记录创建对象的个数，整数变量 objnum用以记录特殊对象的个数，字符缓冲器 name用以保存字符标识符。该缓冲器首先设置为零，然后把构造函数的参数拷贝给它。（注意这里用缺省的字符串参数表明所创建的为数组元素，所以该构造函数实际上充当了缺省构造函数。）因为C标准库中函数strncpy( )在它的第三个参数指定的字符数出现或零终结符出现时，将终止字符的复制，所以被复制字符的数肯定小于缓冲器的大小，并且最后一个字符始终为零，因此打印语句将决不会超出缓冲器。

构造函数在两种情况下会发生异常抛出。第一种情况是当第五个对象被创建时（这只是为了显示在对象数组创建中发生异常，而不是真正的异常条件），这种异常将抛出一个整数，并且函数在异常规格说明中已引入了整数类型。第二种情况当然也是特意设计的，当参数字符串的第一个字符为“ z ”时将抛出一字符型异常。由于异常规格说明中不含有字符型，所以这类异常将调用unexpected( )函数。

函数new 和 delete可对类进行重载，其功能可见其函数调用。

函数unexpected_rethrow( ) 打印一条信息，并且再次抛出同一个异常。在主程序 main( )的第一行中，它充当unexpected( )函数被安装。在测试块中将创建一些noisy对象，但是在对象数组的创建中有异常抛出，所以对象n2将不会被创建。这些在程序输出结果中可以见到：

```
constructing noisy 0 name [before array]
noisy::new[]
constructing noisy 1 name [array elem]
constructing noisy 2 name [array elem]
constructing noisy 3 name [array elem]
constructing noisy 4 name [array elem]
constructing noisy 5 name [array elem]
destructing noisy 4 name [array elem]
destructing noisy 3 name [array elem]
destructing noisy 2 name [array elem]
destructing noisy 1 name [array elem]
noisy::delete[]
destructing noisy 0 name [before array]
caught 5
testing unexpected:
constructing noisy 6 name [before unexpected]
constructing noisy 7 name [z]
inside unexpected_rethrow()
destructing noisy 6 name [before unexpected]
```

```
caught z
```

程序成功地创建了四个对象数组单元，但在构造第五个对象时发生异常抛出。由于第五个对象的构造函数未完成，因此异常在清除对象时只有1~4的析构函数被调用。

全局函数new的一次调用所产生的对象数组的存储分配是分离的。注意，即使程序中没有明确地调用函数delete，但异常处理系统仍不可避免地调用delete函数来释放存储单元。只有在使用规范的new函数形式时才会出现上述情况。如果使用第12章介绍的语法，异常处理机构将不会调用delete函数来清除对象，因为它只适用于清除不是堆结构的存储区。

最终对象n1将被清除，而对象n2由于没被创建所以也不存在被清除的问题。

在测试函数unexpected_rethrow( )的程序段中，对象n3已被创建，对象n4的构造函数已开始创建对象。但是在它创建完成之前已有异常抛出。该异常为字符型，不存在于函数的异常规格说明中，所以函数unexpection( )将被调用（在此例中为函数unexpected_rethrow( )）。由于函数unexpected_rethrow( )可抛出所有类型的异常，所以该函数将再次抛出与已知类型完全相同的异常。当对象n4的构造函数被调用抛出异常后，异常处理器将进行查找并捕获该异常（在成功创建的对象n3被清除之后）。这样函数unexpected_rethrow( )的作用就是接收任意的未加说明的异常，并作为已知异常再次抛出；使用这种方法该函数可为我们提供一滤波器，用以跟踪意外异常的出现并获取该异常的类型。

17.5 构造函数

当编写的程序出现异常时，我们总会问：“当异常出现时，这能被合理地清除掉吗？”这是十分重要的。对于大多数情况，程序是相当安全的；但是如果构造函数中出现异常，这将产生问题：如果异常抛出发生在构造函数创建对象时，对象的析构函数将无法调用其相应的对象。这意味着在编写构造函数的程序时必须十分谨慎。

构造函数进行存储资源分配时存在普遍的困难。如果构造函数在运行时有异常抛出，析构函数将无法收回这些存储资源。这些问题大多数发生在未加保护的指针上。例如：

```
//: NUDEP.CPP -- Naked pointers
#include <fstream.h>
#include <stdlib.h>
ofstream out("nudep.out");

class bonk {
public:
  bonk() { out << "bonk()" << endl; }
  ~bonk() { out << "~bonk()" << endl; }
};

class og {
public:
  void* operator new(size_t sz) {
    out << "allocating an og" << endl;
    throw int(47);
    return 0;
  }
  void operator delete(void* p) {
```

```
    out << "deallocating an og" << endl;
    ::delete p;
  }
};

class useResources {
  bonk* bp;
  og* op;
public:
  useResources(int count = 1) {
    out << "useResources()" << endl;
    bp = new bonk[count];
    op = new og;
  }
  ~useResources() {
    out << "~useResources()" << endl;
    delete []bp; // Array delete
    delete op;
  }
};

main() {
  try {
    useResources ur(3);
  } catch(int) {
    out << "inside handler" << endl;
  }
};
```

输出是：

```
useResources()
bonk()
bonk()
bonk()
allocating an og
inside handler
```

当进入类useResources的构造函数后，并且类bonk的构造函数已成功地完成了对象数组的创建，而这时，在og::operator new中抛出一个异常（例如存储耗尽所产生的异常）。这样我们就意外地在异常处理器中结束程序，而 useResources所对应的析构函数未得到调用。这是正常的，因为类useResources的构造函数的全部构造工作没能全部完成，这就意味着基于堆存储的类bonk的对象也不能被析构。

对象化

为了防止上文提到的情况，应避免对象通过本身的构造函数和析构函数将“不完备的”资源分配到对象中。利用这种方法，每个分配就变成了原子的，像一个对象，并且如果失败，那

么已分配资源的对象也被正确地清除。采用模板是修改上例的一个好方法：

```
//: WRAPPED.CPP -- Safe, atomic pointers
#include <fstream.h>
#include <stdlib.h>
ofstream out("wrapped.out");

// Simplified. Yours may have other arguments.
template<class T, int sz = 1> class pwrap {
  T* ptr;
public:
  class rangeError {}; // Exception class
  pwrap() {
    ptr = new T[sz];
    out << "pwrap constructor" << endl;
  }
  ~pwrap() {
    delete []ptr;
    out << "pwrap destructor" << endl;
  }
  T& operator[](int i) throw(rangeError) {
    if(i >= 0 && i < sz) return ptr[i];
    throw rangeError();
  }
};

class bonk {
public:
  bonk() { out << "bonk()" << endl; }
  ~bonk() { out << "~bonk()" << endl; }
  void g() {}
};

class og {
public:
  void* operator new[](size_t sz) {
    out << "allocating an og" << endl;
    throw int(47);
    return 0;
  }
  void operator delete[](void* p) {
    out << "deallocating an og" << endl;
    ::delete p;
  }
};

class useResources {
```

```
  pwrap<bonk, 3> Bonk;
  pwrap<og> Og;
public:
  useResources() : Bonk(), Og() {
    out << "useResources()" << endl;
  }
  ~useResources() {
    out << "~useResources()" << endl;
  }
  void f() { Bonk[1].g(); }
};

main() {
  try {
    useResources ur;
  } catch(int) {
    out << "inside handler" << endl;
  } catch(...) {
    out << "inside catch(...)" << endl;
  }
}
```

不同点是使用模板封装指针并将它送入对象。这些对象的构造函数的调用先于useResources构造函数的调用，如果这些构造函数的创建操作完成之后发生了异常抛出，与它们相对应的析构函数被调用。

模板pwrap演示了比前面所见更为经典的异常使用：如果 operator[]的参数出界，那么就创建一个嵌入类rangeError用于operator[]中。因为operator[]返回一个引用而不是返回0。（没有0引用。）这是一个真实的异常情况：不知道在当前上下文中该做什么，也不能返回一个不可能的值。在此例中，rangeError很简单而且设想所有必须的信息都在类名中，但我们也可加入含有索引值的成员，如果这样做有用的话。

现在输出是：

```
bonk()
bonk()
bonk()
pwrap constructor
allocating an og
~bonk()
~bonk()
~bonk()
pwrap destructor
inside handler
```

对og的空间存储分配又抛出一个异常，但这次bonk对象数组正确地被清除，所以没有存储损耗。

17.6 异常匹配

当一个异常抛出时，异常处理系统会根据所写的异常处理器顺序找到“最近”的异常处理器，而不会搜寻更多的异常处理器。

异常匹配并不要求在异常和处理器之间匹配得十分完美。一个对象或一个派生类对象的引用将与基类处理器匹配（然而假若处理器针对的是对象而非引用，异常对象在传递给处理器时会被“切片”，这样不会受到破坏但会丢失所有的派生类型信息）。假若抛出一个指针，标准指针转化处理会被用于匹配异常，但不会有自动的类型转化将某个异常类型在匹配过程中转化为另一个。下面是一个例子：

```
//: AUTOEXCP.CPP -- No matching conversions
#include <iostream.h>
class except1 {};
class except2 {
public:
  except2(except1&) {}
};

void f() { throw except1(); }

main() {
  try { f();
  } catch (except2) {
    cout << "inside catch(except2)" << endl;
  } catch (except1) {
    cout << "inside catch(except1)" << endl;
  }
}
```

尽管我们可能认为第一个处理器会使用构造函数转化，将一个 except1对象转化成except2对象，但是系统在异常处理期间将不会执行这样的转换，我们将在 except1处终止。

下面的例子展示基类处理器怎样捕获派生类的异常：

```
//: BASEXCPT.CPP -- Exception hierarchies
#include <iostream.h>

class X {
public:
  class trouble {};
  class small : public trouble {};
  class big : public trouble {};
  void f() { throw big(); }
};

main() {
  X x;
  try {
```

```
    x.f();
  } catch(X::trouble) {
    cout << "caught trouble" << endl;
  // Hidden by previous handler:
  } catch(X::small) {
    cout << "caught small trouble" << endl;
  } catch(X::big) {
    cout << "caught big trouble" << endl;
  }
}
```

这里的异常处理机制，对于第一个处理器总是匹配一个 trouble对象或从trouble派生的什么事物，由于第一个处理器捕获涉及第二和第三处理器的所有异常，所以第二和第三处理器永远不被调用。光捕获派生类异常把基类的一般异常放在末端捕获更有意义（或者在随后的下一个开发周期中引入的派生类）。

另外，假若small和big的对象比trouble的大（这常常是真实的，因为通常为派生类添加成员），那么这些对象会被“切片”以适应处理器。当然，在本例中由于派生类没有附加成员，而且在处理器中也没有参数标识，所以这一点并不重要。通常在处理器中，应该使用引用参数而非对象以避免裁剪掉信息。

17.7 标准异常

用于C++类标准库的一批异常可以用于我们自己的程序中。从标准异常类开始会比我们尽量自己定义来得快和容易。假若标准异常类不能满足需要，我们可以继承它并添加自己的特定内容。下面的表描述了标准异常：

exception	是所有标准C++库异常的基类。我们可以调用 what()以获得其特性的显示说明
logic_erro	是由exception派生的。它报告程序的逻辑错误，这些错误在程序执行前可以被检测到
runtime_error	是由exception派生的。它报告程序运行时错误，这些错误仅在程序运行时可以被检测到

I/O流异常类ios::failure也由exception派生，但它没有进一步的子类：

下面两张表中的类都可以按说明使用，也可以作为基类去派生我们自己的更为特殊的异常类型。

由logic_error派生的异常	
domain_error	报告违反了前置条件
invalid_argument	指出函数的一个无效参数
length_error	指出有一个产生超过NPOS长度的对象的企图（NPOS：类型size_t 的最大可表现值）
out_of_range	报告参数越界
bad_cast	在运行时类型识别中有一个无效的 dynamic_cast表达式（见第 18章）
bad_typeid	报告在表达式typeid(*p)中有一个空指针 P（运行时类型识别的特性见第 18章）

由runtime_error派生的异常	
range_error	报告违反了后置条件
overflow_error	报告一个算术溢出
bad_alloc	报告一个存储分配错误

17.8 含有异常的程序设计

对大多数程序员尤其是C程序员，在他们的已有的程序设计语言中不能使用异常，需进一步矫正。下面是一些含有异常的程序设计原则。

17.8.1 何时避免异常

异常并不能回答所发生的所有问题。实际上若对异常进行钻牛角尖式的推敲，将会遇到许多麻烦。下面的段落指出异常不能被保证的情况。

1. 异步事件

标准C的signal()系统以及其他类似的系统操纵着异步事件：该事件发生在程序控制的范围以外，它的发生是程序所不能预计的。由于异常和它的处理器都在相同的调用栈上，所以异常不能用来处理异步事件。也就是异常限制在某范围内，而异步事件必须有完全独立的代码来处理，这些代码不是普通程序流的一部分（典型的如中断服务和事件循环例程）。

这并不是说异步事件不能和异常发生关系。但是，中断服务处理器都尽可能快地工作，然后返回。在一些定义明确的程序点上，一个异常可以以基于中断的方式抛出。

2. 普通错误情况

假若有足够的信息去处理一个错误，这个错误就不是一个异常。我们应该关心当前的上下文环境，而不应该把异常抛向更大的上下文环境中。同样，在 C++中也不应当为机器层的事件抛出异常，如“除零溢出”。可以认为这些“异常”可由其他的机制去处理，如操作系统或硬件。这样，C++异常可以相当有效，并且它们的使用和程序级的异常条件相互隔离。

3. 流控制

一个异常看上去有点象一个交替返回机制，也有点象一个 switch语句段，我们可能被它们吸引，改变了想使用它们的初衷，这是一个很糟糕的想法，一部分原因是因为异常处理系统比普通的程序运行缺乏效率。异常是一个罕有的事件，所以普通程序不应为其支付时间，来自非错误条件的其他什么地方的异常也会给使用我们的类或函数的用户带来相当的混乱。

4. 不强迫使用异常

一些程序相当简单，如一些实用程序，可能仅仅需要获取输入和执行一些加工。如果在这类程序中试图分配存储然而失败了，或打开一个文件然而失败了等等，这样可以在这类程序中使用assert()以示出错信息，使用abort()终止程序，允许系统清除混乱。但是如果我们自己努力去捕获所有异常，修复系统资源，则是不明智的。从根本上说，假若我们不必使用异常，我们就不要用。

5. 新异常，老代码

另一种情形出现在对没有使用异常的已存在的程序进行修改的时候。我们可能引入一个使用异常的库而且想知道是否有必要在程序中修改所有的代码。假定已经安放了一个可接受的出错处理配置，这里所要做的最明智的事情是围绕着使用新类 try块的最大程序块，追加一个catch(…)和基本出错信息。我们可以追加有必要的更多特定的处理器，并使修改更为细致。但是，在这种情况下，被迫增加的代码必须是最小限度的。

我们也可以把我们的异常生成代码隔离在try块中，并且编写一个把当前异常转换成已存在的出错处理方案的处理器。

创建一个为其他人使用的库，而且无从知晓用户在遭遇决定性错误的情况下如何反应，这时考虑异常才真正重要。

17.8.2 异常的典型使用

使用异常便于：

1) 使问题固定下来和重新调用这个（导致异常的）函数。

2) 把事情修补好而继续运行，不去重试函数。

3) 计算一些选择结果用于代替函数假定产生的结果。

4) 在当前上下文环境尽其所能并且再把同样的异常弹向更高的上下文中。

5) 在当前上下文环境尽其所能并且把一个不同的异常弹向更高的上下文中。

6) 终止程序。

7) 包装使用普通错误方案的函数（尤其是C的库函数），以便产生异常替代。

8) 简化，假若我们的异常方案建造得过于复杂，使用时会令人懊恼。

9) 使我们的库和程序更安全。这是短期投资（为了调试）和长期投资（为了应用的健壮性）问题。

1. 随时使用异常规格说明

异常的规格说明像一个函数原型：它告诉用户书写异常处理代码以及处理什么异常。它告诉编译器异常可能出现在这个函数中。

当然，我们不能总是通过检查代码而预见什么异常会发生在特定的函数中。有时这个特定函数所调用的函数产生了一个出乎意料的异常，有时一个不会抛出异常的老函数被一个会抛出异常的新函数替换了，这样我们将产生对 unexpect()的调用。无论何时，只要使用异常规格说明或者调用含有异常的函数，都应该创建自己的 unexpected()函数，该函数记录信息而且重新抛出同样的异常。

2. 起始于标准异常

在创建我们自己的异常前应检查标准 C++异常库。假若标准异常正合所需，则这样会使我们的用户更易于理解和处理。

假若所需要的异常类型不是标准库的一部分，则尽量从某个已存在标准 expection中派生形成。假若在expection的类接口中总是存在what()函数的期望定义，这会使用户受益匪浅。

3. 套装我们自己的异常

如果为我们的特定类创建异常，在我们的类中套装异常类是一个很好的主意，这为读者提供了一个清晰的消息 —— 这些异常仅为我们的类所使用。另外，它可防止命名域混乱。

4. 使用异常层次

异常层次为不同类型的重要错误的分类提供了一个有价值的方法，这些错误可能会与我们的类或库冲突。该层次可为用户提供有帮助的信息，帮助他们组织自己的代码，让他们可以选择是忽略所有异常的特定类型还是正确地捕获基类类型。而且在以后，任何异常可通过对相同基类的继承而追加，而不会被迫改写所有的已生成代码 —— 基类处理器将捕获新的异常。

当然，标准C++异常是一个异常层次的优秀例子，通过使用可进一步增强和丰富它。

5. 多重继承

我们会记得，在第15章中，多重继承最必要做的地方就是需要把一个指向对象的指针向上映射到两个不同的基类，也就是需要两个基类的多态行为的地方。这样，异常层次对于多重继承是有用的，因为多重继承异常类的任一根的基类处理器都可处理异常。

6. 用“引用”而非“值”去捕获

如果抛出一个派生类对象而且该对象被基类的对象处理器通过值捕获到，对象会被“切片”，

这就是说，随着向基类对象的传递，派生类元素会依次被割下，直到传递完成。这样的偶然性并不是所要的，因为对象的行为像基类而不象它本来就是的派生类对象（实际就是“切片”以前）。下面是一个例子：

```
//: CATCHREF.CPP -- Why catch by reference?
#include <iostream.h>

class base {
public:
  virtual void what() {
    cout << "base" << endl;
  }
};

class derived : public base {
public:
  void what() {
    cout << "derived" << endl;
  }
};

void f() { throw derived(); }

main() {
  try {
    f();
  } catch(base b) {
    b.what();
  }
  try {
    f();
  } catch(base& b) {
    b.what();
  }
}
```

输出为

```
base
derived
```

当对象通过值被捕获时,因为它被转化成一个base对象(由构造函数完成)，而且在所有的情况下表现出base对象的行为；然而当对象通过引用被捕获时，仅仅地址被传递而对象不会被切片，所以它的行为反映了它处于派生中的真实情况。

虽然也可以抛出和捕获指针，但这样做会引入更多的耦合——抛出器和捕获器必须为怎样分配和清理异常对象而达成一致。这是一个问题，因为异常本身可能会由于堆的耗尽而产生。如果抛出异常对象，异常处理系统会关注所有的存储。

7. 在构造函数中抛出异常

由于构造函数没有返回值，因此在先前我们可以有两个选择以报告在构造期间的错误：

1) 设置一个非局部标志并且希望用户检查它。

2) 返回一个不完全被创建的对象并且希望用户检查它。

这是一个严重的问题，因为 C程序员必须依赖一个隐含的保证：对象总是成功地被创建，这在类型如此粗糙的C中是不合理的。但是在C++程序中，构造失败后继续执行是注定的灾难，于是构造函数成为抛出异常最重要的地方之一。现在有一个安全有效的方法去处理构造函数错误。然而我们还必须把注意力集中在对象内部的指针上和构造函数异常抛出时的清除方法上。

8. 不要在析构函数中导致异常

由于析构函数会在抛出其他异常时被调用，所以永远不要打算在析构函数中抛出一个异常，或者通过执行在析构函数中的相同动作导致其他异常的抛出。如果这些发生了，这意味着在已存在的异常到达引起捕获之前抛出了一个新的异常，这会导致对 terminate()的调用。

这里的意思是：假若调用一个析构函数中的任何函数都有可能会抛出异常，这些调用应该写在析构函数中的一个try块中，而且析构函数必须自己处理所有自身的异常。这里的异常都不应逃离析构函数。

9. 避免无保护的指针

请看第17.5.1节中的WRAPPED.CPP程序，假若资源分配给无保护的指针，那么意味着在构造函数中存在一个缺点。由于该指针不拥有析构函数，所以当在构造函数中抛出异常时那些资源将不能被释放。

17.9 开销

为了使用新特性必然有所开销。当异常被抛出时有相当的运行时间方面的开销，这就是从来不想把异常用于普通流控制的一部分的原因，而不管它多么令人心动。异常的发生应当是很少的，所以开销聚集在异常上而不是在普通的执行代码上。设计异常处理的重要目标之一是：在异常处理实现中，当异常不发生时应不影响运行速度。这就是说，只要不抛出异常，代码的运行速度如同没有加载异常处理时一样。无论与否，异常处理都依赖于使用的特定编译器。

异常处理也会引出额外信息，这些信息被编译器置于栈上。

除了能作为特定的“ 异常范围 ”（它可能恰恰是全局范围）的对象传进送出外，异常对象可以像其他对象一样被正确地在周围传递。当异常处理器工作完成时，异常对象也被相应地销毁。

17.10 小结

错误恢复是和我们编写的每个程序相关的基本原则，在 C++中尤其重要，创建程序组件为其他人重用是开发的目标之一。为了创建一个稳固系统，必须使每个组件具有健壮性。

C++中异常处理的目标是简化大型可靠程序的创建，使用尽可能少的代码，使应用中没有不受控制的错误而使我们更加自信。这几乎不损害性能，并且对其他代码的影响很小。

基本异常不特别难学，我们应该在程序中尽量地使用它们。异常是能给我们提供即时而显著的好处的特性之一。

17.11 练习

1) 创建一个含有可抛出异常的成员函数的类。在该类中，创建一个被嵌套的类用作一个异

常对象，它带有一个char*参数，该参数表示一个描述型字符串。创建一个可抛出该异常的成员函数。（标明函数的异常规格说明）书写一个try块使它能调用该函数并且捕获异常，以打印描述型字符串的方式处理该异常。

2) 重写第12章中的stash类以便为operator[]抛出out-of-range异常。

3) 写一个一般的main()，它可取走所有的异常并且报告错误。

4) 创建一个拥有自身运算符new的类。该运算符分配10个对象，在对第11个对象分配时假定“存储耗尽”并抛出一个异常。增加一个静态函数用于回收存储。现在，创建一个伴有try块和能够调用存储恢复例程的catch子句的主程序，将这些都放入一个while循环中，演示异常恢复和连续执行的情形。

5) 创建一个可抛出异常的析构函数，编写代码以向自己证明这是一个糟糕的想法。该代码可展示如果处理器对一个已存在异常施加影响之前一个新异常又抛出了，那么terminate()会被调用。

6) 向我们自己证明所有的异常对象（被抛出的）都能被正确地销毁。

7) 向我们自己证明假若我们在堆上创建一个异常对象并且抛出一个指向该对象的指针，则它不会被清理掉。

8) （高级）。使用一个带有构造函数和拷贝构造函数的类来追踪异常的创建和传递，这些构造函数和拷贝构造函数显示它们自身而且尽可能地提供关于对象是怎样创建的信息（就拷贝构造函数而言，说明创建什么对象）。创立一个有趣的状态，抛出我们的新类型的对象并分析结果。