第 20 章　处理错误和异常

本书之前的程序都是为了演示而创建的，它们没有处理错误以免影响读者将重点放在重要主题上。实际的程序必须考虑错误状态。

本章介绍以下内容：

- 什么是异常？
- 如何使用异常？它们会带来什么问题？
- 如何建立异常层次结构？
- 如何使异常符合全局错误处理的方法？
- 什么是调试器？

20.1　程序中的各种错误

实际的程序很少没有 bug。程序越大，有 bug 的可能越大。实际上，很多 bug“走出家门”进入了最终发布的软件中。编写健壮、没有 bug 的程序是任何对编程持严肃态度者的首要任务。

软件业中的最大问题就是有错误、不稳定的代码。在很多重要的编程工作中，花费最大的是测试和修改。以低成本按时生产出优秀、稳定、可靠的程序的人将给软件业带来革命性影响。

各种 bug 将给程序带来麻烦。首先是逻辑性差：程序能够完成要求的工作，但你并没有正确地考虑算法。其次是语法问题：使用了错误的惯用语、函数或结构。这两种是最常见的，它们是大多数程序员都要警惕的错误。

研究和实际经验表明，在开发过程中发现逻辑问题的时间越晚，修复它所需付出的代价越高。费用最低的修复问题的方法是尽量避免产生错误，其次是编译器可发现的错误。C++标准要求编译器尽可能让更多 bug 在编译时浮出水面。

与只是偶尔导致程序崩溃的错误相比，发现和修复让程序能够通过编译但首次测试时就会出现的错误（导致程序每次运行时都崩溃的错误）的代价更低些。

比逻辑和语法错误更常见的运行阶段问题是脆弱性：程序要求输入一个数字时，如果用户输入一个数字，程序将正常运行；但如果用户输入一个字母，程序将崩溃。另一些程序在内存不足、软盘不在软驱中或 Internet 连接断开时崩溃。

为避免这种脆弱性，程序员致力于提高程序的健壮性。健壮（bulletproof）的程序能够应付运行阶段发生的任何问题，从怪异用户输入到内存不足。

区分 bug、逻辑错误和异常至关重要。bug 是由于程序员犯错引起的，逻辑错误是由于程序员对解决问题的方式不了解引起的；异常是由于不常见、但可预见的问题（如内存或硬盘空间等资源耗尽）引起的。

异常情况

你无法消除异常情况，只能为异常情况做好准备。如果程序要动态地为对象分配内存，但系统没有内存可用，将发生什么？程序将如何应对？如果你导致常见的数学错误之一（被零除），程序将如何做？程序的选

择包括：

- 崩溃；
- 通知用户并妥善地退出；
- 通知用户并允许用户尽量恢复并继续执行；
- 采取正确的措施，在不影响用户的情况下继续运行。

请看程序清单 20.1，它非常简单且运行时将崩溃，但演示了很多程序都可能出现的极其严重的问题。

程序清单 20.1　导致异常情形

```
 0: // This program will crash
 1: #include <iostream>
 2: using namespace std;
 3:
 4: const int DefaultSize = 10;
 5:
 6: int main()
 7: {
 8:    int top = 90;
 9:    int bottom = 0;
10:
11:    cout << "top / 2 = " << (top/ 2) << endl;
12:
13:    cout << "top divided by bottom = ";
14:    cout << (top / bottom) << endl;
15:
16:    cout << "top / 3 = " << (top/ 3) << endl;
17:
18:   cout << "Done." << endl;
19:   return 0;
20: }
```

输出：

```
top / 2 = 45
top divided by bottom =
```

分析：

程序清单 20.1 有意被设计成导致崩溃；然而，如果程序要求用户输入两个数，并将它们相除，也可能出现这样的问题。

第 8 和 9 行声明了两个 int 变量并给它们赋值。也可以提示用户输入两个数字或从文件中读取。在第 11、14 和 16 行，这些数字被用于数学运算。具体地说，它们被用于除法运算。在第 11 和 16 行，没有任何问题，但第 14 行存在严重的问题。除以零将导致异常问题发生：程序崩溃。程序将终止，操作系统很可能会显示一条异常消息。

虽然并非总是必须（乃至希望）悄悄地自动地从所有异常情形恢复，但显然必须做得比这个程序更好些：不能让程序崩溃。

C++的异常处理提供了一种类型安全的集成方法，来应对程序运行时出现的可预见到但不常发生的情形。

20.2　异常的基本思想

异常的基本思想非常简单：

- 计算机试图执行一段代码。这段代码可能要分配资源（如内存）、锁定文件或执行其他各种任务。
- 包含应对代码由于异常原因而执行失败的逻辑（代码）。例如，可能包含捕获各种问题（如无法分配内存、无法锁定文件或各种其他的问题）的代码。
- 在你的代码被其他代码使用时（如一个函数调用另一个），也需要一种机制来将有关问题（异常）的信息传递到下一级。应该有一条从问题发生的代码到处理错误状态的代码的路径。如果函数之间存在中间层，应该给它们提供解决问题的机会，但不应要求它们包含只是为了传递错误状态的代码。

异常处理将这 3 点结合在一起，且工作原理相对比较简单。

20.2.1 异常处理的组成部分

要处理异常，必须首先确定需要监视哪段代码可能发生的异常。这可以使用 try 块来完成。

应创建 try 块来包围可能导致问题的代码块。try 块的基本格式如下：

```
try
{
    SomeDangerousFunction();
}
catch (...)
{
}
```

在这个例子中，当 SomeDangerousFunction( )执行时，如果发生任何异常，都将被发现并捕获。要监视异常，只需加上关键字 try 和大括号。当然，如果异常发生，需要采取措施来处理它。

当 try 块中的代码执行时，如果发生异常，则被称为引发异常。然后可以捕获引发的异常，如前一个范例所示，使用 catch 块来捕获异常。引发异常后，将转移到当前 try 块后面合适的 catch 块执行。在前一个范例中，省略号（...）表示任何异常。也可以捕获特定类型的异常，为此，可以在 try 块后面使用一个或多个 catch 块。例如：

```
try
{
    SomeDangerousFunction();
}
catch(OutOfMemory)
{
    // take some actions
}
catch(FileNotFound)
{
    // take other action
}
catch (...)
{
}
```

在这个例子中，当 SomeDangerousFunction( )被执行时，有处理异常的代码。如果引发了异常，它将被传递给 try 块后面的第一个 catch 块。如果该 catch 块有类型参数（就像前一个范例中那样），将检查异常是否与指定的类型匹配。如果不匹配，将检查下一条 catch 语句，这一过程不断进行下去，直到找到匹配的 catch 块或到达非 catch 块代码。找到第一个匹配的 catch 块后将执行它。除非有意想让某些类型的异常逃脱，否则总是应该让最后一个 catch 块使用省略号作为参数。

注意：catch 块也被称为处理程序，因为它能够处理异常。

注意：可以将 catch 块视为类似于被重载的函数，找到特征标匹配的函数后，将执行它。

处理异常的基本步骤如下：

（1）确定程序中执行某种操作且可能引发异常的代码，并将它们放到 try 块中。

（2）创建 catch 块，在异常被引发时捕获它们。可以创建捕获特定类型的异常的 catch 块（通过指定 catch 块的类型参数），也可以创建捕获所有异常的 catch 块（使用省略号作为参数）。

程序清单 20.2 在程序清单 20.1 中添加了基本的异常处理功能，这里使用了 try 块和 catch 块。

注意：有些非常老的编译器不支持异常。然而，异常是 ANSI C++标准的一部分，每个编译器厂商的最新编译器版本都全面支持异常。如果读者使用的是较老的编译器，将无法编译和运行本章的范例。然而，通读本章并在对编译器进行升级复习这些内容，仍不失为一个不错的主意。

程序清单 20.2　捕获异常

```
 0: // trying and catching
 1: #include <iostream>
 2: using namespace std;
 3:
 4: const int DefaultSize = 10;
 5:
 6: int main()
 7: {
 8:    int top = 90;
 9:    int bottom = 0;
10:
11:    try
12:    {
13:       cout << "top / 2 = " << (top/ 2) << endl;
14:
15:       cout << "top divided by bottom = ";
16:       cout << (top / bottom) << endl;
17:
18:       cout << "top / 3 = " << (top/ 3) << endl;
19:    }
20:    catch(...)
21:    {
22:       cout << "something has gone wrong!" << endl;
23:    }
24:
25:    cout << "Done." << endl;
26:    return 0;
27: }
```

输出：

```
top / 2 = 45
top divided by bottom = something has gone wrong!
Done.
```

分析：

不同于前一个程序清单，执行程序清单 20.2 时不会导致崩溃，它能够报告问题并妥善地退出。

这一次将可能发生问题的代码（除法运算）放在 try 块（第 11～19 行）中。为处理异常，try 块后面包含一个 catch 块（第 20～23 行）。

第 20 行的 catch 语句中包含一个省略号。正如前面指出的，这是一种特殊的 catch 语句。除非异常被之

前的 catch 块处理了，否则前面的 try 块中代码导致的所有异常都将由该 catch 块处理。在该程序清单中，只可能发生被零除的错误。正如后面将介绍的，最好考虑更具体的异常类型，这样可以自定义每种异常的处理方式。

读者应该注意到了，该程序清单执行时没有崩溃。另外，从输出可知，程序继续执行了 catch 语句后面的第 25 行，单词“Done”被打印到控制台证明了这一点。

try 块

try 块是以关键字 try 和{打头且以}结尾的一系列语句。

范例：

```
try
{
    Function();
};
```

catch 块

catch 块是以关键字 catch、用括号括起的异常类型和{打头且以}结束的一系列语句。

范例：

```
try
{
    Function();
};
catch (OutOfMemory)
{
    // take action
}
```

20.2.2 手工引发异常

程序清单 20.2 演示了异常处理的两个方面：标记要监控的代码和指定要如何处理异常，然而只处理预定义的异常。异常处理的第二部分是，让你能够创建自己的、将被处理的异常类型。通常创建自己的异常，可以自定义异常处理程序（catch 块），使其对你的应用程序是有意义的。

要创建导致 try 语句对其做出反应的异常，可使用关键字 throw。实际上，你引发异常，而处理程序（catch 块）可能捕获它。throw 语句的基本格式如下：

```
throw exception;
```

该语句引发异常 exception。这将导致程序跳到一个处理程序处执行，如果没有找到匹配的处理程序，程序将终止。

引发异常时，几乎可以使用任何类型的值。正如前面指出的，可以为程序可能引发的每种异常编写相应的处理程序。程序清单 20.3 对程序清单 20.2 进行了修改，演示了如何引发一种基本异常。

程序清单 20.3 引发异常

```
0:  //Throwing
1:  #include <iostream>
2:
3:  using namespace std;
4:
5:  const int DefaultSize = 10;
6:
```

```
 7: int main()
 8: {
 9:     int top = 90;
10:    int bottom = 0;
11:
12:    try
13:    {
14:      cout << "top / 2 = " << (top/ 2) << endl;
15:
16:      cout << "top divided by bottom = ";
17:      if ( bottom == 0 )
18:         throw "Division by zero!";
19:
20:      cout << (top / bottom) << endl;
21:
22:      cout << "top / 3 = " << (top/ 3) << endl;
23:    }
24:    catch( const char * ex )
25:    {
26:      cout << "\n*** " << ex << " ***" << endl;
27:    }
28:
29:    cout << "Done." << endl;
30:    return 0;
31: }
```

输出：

```
top / 2 = 45
top divided by bottom = *** Division by zero! ***
Done.
```

分析：

不同于前一个程序清单，该程序清单更严格地控制了其异常。虽然这不是使用异常的最佳方式，但清楚地演示了如何使用 throw 语句。

第 17 行检查 bottom 的值是否为零。如果是则引发异常。在这个例子中，异常是一个字符串值。

处理程序从第 24 行的 catch 语句开始。该程序程序捕获常量字符指针。在异常方面，字符串与常量字符指针匹配，因此从 24 行开始的处理程序将捕获第 18 行引发的异常。第 26 行显示传递的字符串，并在前面和后面加上星号。第 27 行的右大括号表明处理程序到此结束，因此跳到 catch 语句后面的第一行，并继续执行到程序末尾。

如果异常是一个更严重的问题，可以在第 26 行打印消息后退出应用程序。如果在函数中引发异常，而该函数被其他函数调用，可以将异常向上传递。要将异常向上传递，只需调用 throw 命令且不提供任何参数，这将在当前位置重新引发现有的异常。

20.2.3　创建异常类

可以创建一个更复杂的类，供引发异常时使用。程序清单 20.4 是一个精简的 Array 类，它基于第 19 章开发的模板。

程序清单 20.4　引发异常

```
 0: #include <iostream>
 1: using namespace std;
```

```
 2:
 3: const int DefaultSize = 10;
 4:
 5: class Array
 6: {
 7:   public:
 8:     // constructors
 9:     Array(int itsSize = DefaultSize);
10:     Array(const Array &rhs);
11:     ~Array() { delete [] pType;}
12:
13:     // operators
14:     Array& operator=(const Array&);
15:     int& operator[](int offSet);
16:     const int& operator[](int offSet) const;
17:
18:     // accessors
19:     int GetitsSize() const { return itsSize; }
20:
21:     // friend function
22:     friend ostream& operator<< (ostream&, const Array&);
23:
24:     class xBoundary {}; // define the exception class
25:
26:   private:
27:     int *pType;
28:     int itsSize;
29: };
30:
31: Array::Array(int size):
32:   itsSize(size)
33: {
34:   pType = new int[size];
35:   for (int i = 0; i < size; i++)
36:     pType[i] = 0;
37: }
38:
39: Array& Array::operator=(const Array &rhs)
40: {
41:   if (this == &rhs)
42:     return *this;
43:   delete [] pType;
44:   itsSize = rhs.GetitsSize();
45:   pType = new int[itsSize];
46:   for (int i = 0; i < itsSize; i++)
47:   {
48:     pType[i] = rhs[i];
49:   }
50:   return *this;
51: }
52:
53: Array::Array(const Array &rhs)
54: {
55:   itsSize = rhs.GetitsSize();
```

```
56:    pType = new int[itsSize];
57:    for (int i = 0; i < itsSize; i++)
58:    {
59:      pType[i] = rhs[i];
60:    }
61:  }
62:
63:  int& Array::operator[](int offSet)
64:  {
65:    int size = GetitsSize();
66:    if (offSet >= 0 && offSet < GetitsSize())
67:      return pType[offSet];
68:    throw xBoundary();
69:    return pType[0]; // appease MSC
70:  }
71:
72:  const int& Array::operator[](int offSet) const
73:  {
74:    int mySize = GetitsSize();
75:    if (offSet >= 0 && offSet < GetitsSize())
76:      return pType[offSet];
77:    throw xBoundary();
78:    return pType[0]; // appease MSC
79:  }
80:
81:  ostream& operator<< (ostream& output, const Array& theArray)
82:  {
83:    for (int i = 0; i<theArray.GetitsSize(); i++)
84:      output << "[" << i << "] " << theArray[i] << endl;
85:    return output;
86:  }
87:
88:  int main()
89:  {
90:    Array intArray(20);
91:    try
92:    {
93:      for (int j = 0; j< 100; j++)
94:      {
95:        intArray[j] = j;
96:        cout << "intArray[" << j << "] okay..." << endl;
97:      }
98:    }
99:    catch (Array::xBoundary)
100:   {
101:     cout << "Unable to process your input!" << endl;
102:   }
103:   cout << "Done." << endl;
104:   return 0;
105: }
```

输出：

```
intArray[0] okay...
intArray[1] okay...
```

```
intArray[2] okay...
intArray[3] okay...
intArray[4] okay...
intArray[5] okay...
intArray[6] okay...
intArray[7] okay...
intArray[8] okay...
intArray[9] okay...
intArray[10] okay...
intArray[11] okay...
intArray[12] okay...
intArray[13] okay...
intArray[14] okay...
intArray[15] okay...
intArray[16] okay...
intArray[17] okay...
intArray[18] okay...
intArray[19] okay...
Unable to process your input!
Done.
```

分析：

程序清单 20.4 使用了一个精简的 Array 类，但添加了异常处理功能，以防超出边界。

第 24 行为类 Array 的声明中声明了一个新类 xBoundary。

无法看出这个新类是一个异常类，它和其他类没有任何区别。这个类极为简单，它没有数据和方法，但它确实是一个有效的类。

实际上，说它没有方法是不对的，因为编译器将自动为它创建一个默认构造函数、析构函数、复制构造函数和赋值运算赋。因此它实际上有 4 个成员函数，但没有数据。

注意，在 Array 中声明它只是为了将它们连接起来。正如第 16 章讨论的，Array 在访问 xBoundary 方面没有特权，xBoundary 在访问 Array 的成员方面也没有特权。

在第 63～70 行以及第 72～79 行，对下标运算符进行了修改，使之对下标进行检查。如果下标不在有效范围内，则将 xBoundary 类作为异常进行引发。括号是必不可少的，这样才是调用 xBoundary 的构造函数，而不是使用一个枚举常量。

在 main( )函数中，第 90 声明了一个能够存储 20 个元素的 Array 对象。第 91 行的关键字 try 表明接下来是一个 try 块，该 try 块到第 98 行结束。在 try 块中，将 101 个整数添加到第 90 行声明的数组中。

在第 99 行，处理程序被声明为捕获 xBoundary 异常。

在第 88～105 行的 main( )函数中，创建了一个 try 块，并在其中对数组的每个成员进行初始化。当 j（第 93 行）递增到 20 时，将访问下标为 20 的成员。这导致第 66 行的测试失败，operatorp[ ]中的第 67 行代码引发 xBoundary 异常。

程序跳到第 99 行的 catch 块，该行的 catch 捕获（处理）异常：打印一条错误消息。程序执行到 catch 块的末尾（第 102 行）。

20.3 使用 try 块和 catch 块

确定在什么地方放置 try 块可能比较困难：可能引发异常的操作并非总是那么明显。下一个问题是在什么地方捕获异常。你也许想在分配内存的地方引发所有内存异常，但想在程序上层处理用户界面的地方捕获异常。

在确定 try 块的位置时，考虑在何处分配内存或资源。要监视的其他异常包括越过边界、无效输入等。

至少应该在 main( )函数中所有代码的周围放置 try/catch 块。try/catch 块通常放在高级函数中，尤其是那些知道程序用户界面的函数中。例如，实用（utility）类通常不应捕获那些需要报告给用户的异常，因为这种类可能被用于窗口程序、控制台程序、甚至通过 Web 或消息收发功能与用户交流的程序中。

20.4　捕获异常的工作原理

捕获异常的工作原理如下：异常被引发后，将检查调用栈。调用栈是在程序的一部分调用另一个函数时创建的函数调用列表。

调用栈记录执行路径。如果 main( )调用了函数 Animal::GetFavoriteFood( )，GetFavoriteFood( )调用了函数 Animal::LookupPreferences( )，而后者又调用了 fstream::operator>>( )，这些调用都将记录在调用栈中。递归函数可能在调用栈中出现多次。

异常沿调用向上传递给每个封闭块（enclosing block），这被称为堆栈解退（unwinding the stack）。当堆栈被解退时，将对堆栈中的局部对象调用析构函数，将对象销毁。

每个 try 块后面都有一个或多个 catch 语句。如果异常与某个 catch 语句匹配，将执行该 catch 语句并认为其已得到处理。如果没有匹配的 catch 语句，将继续解退堆栈。

如果异常直到程序开始位置（main( )）还没有被捕获，将调用内置的处理程序来终止程序。

需要注意的是，异常沿堆栈向上传递是条单行线。在异常向上传递的过程中，堆栈被解退，堆栈中的对象被销毁。没有回头路可走：异常被捕获后，程序将继续执行捕获异常的 catch 语句后面的语句。

因此，在程序清单 20.4 中，程序继续执行第 101 行：捕获 xBoundary 异常的 catch 语句后面的第一行。引发异常后，程序跳到后面的 catch 块处继续执行，而不是继续执行异常引发点后面的代码。

20.4.1　使用多条 catch 语句

可能引发多种异常。在这种情况下，可以依次使用多条 catch 语句，就像 switch 语句中的条件一样。与 default 语句对应的是“捕获所有异常”的 catch 语句：catch(...)。程序清单 20.5 演示了多种异常。

程序清单 20.5　多种异常

```
0:  #include <iostream>
1:  using namespace std;
2:
3:  const int DefaultSize = 10;
4:
5:  class Array
6:  {
7:    public:
8:      // constructors
9:      Array(int itsSize = DefaultSize);
10:     Array(const Array &rhs);
11:     ~Array() { delete [] pType; }
12:
13:     // operators
14:     Array& operator=(const Array&);
15:     int& operator[](int offSet);
16:     const int& operator[](int offSet) const;
17:
18:     // accessors
19:     int GetitsSize() const { return itsSize; }
20:
```

```
21:     // friend function
22:     friend ostream& operator<< (ostream&, const Array&);
23:
24:     // define the exception classes
25:     class xBoundary {};
26:     class xTooBig {};
27:     class xTooSmall{};
28:     class xZero {};
29:     class xNegative {};
30:   private:
31:     int *pType;
32:     int  itsSize;
33: };
34:
35: int& Array::operator[](int offSet)
36: {
37:   int size = GetItsSize();
38:   if (offSet >= 0 && offSet < GetItsSize())
39:     return pType[offSet];
40:   throw xBoundary();
41:   return pType[0];  // appease MFC
42: }
43:
44:
45: const int& Array::operator[](int offSet) const
46: {
47:   int mysize = GetItsSize();
48:   if (offSet >= 0 && offSet < GetItsSize())
49:     return pType[offSet];
50:   throw xBoundary();
51:
52:   return pType[0];  // appease MFC
53: }
54:
55:
56: Array::Array(int size):
57:   itsSize(size)
58: {
59:   if (size == 0)
60:     throw xZero();
61:   if (size < 10)
62:     throw xTooSmall();
63:   if (size > 30000)
64:     throw xTooBig();
65:   if (size < 1)
66:     throw xNegative();
67:
68:   pType = new int[size];
69:   for (int i = 0; i < size; i++)
70:     pType[i] = 0;
71: }
72:
```

```
73: int main()
74: {
75:    try
76:    {
77:       Array intArray(0);
78:       for (int j = 0; j < 100; j++)
79:       {
80:          intArray[j] = j;
81:          cout << "intArray[" << j << "] okay..." << endl;
82:       }
83:    }
84:    catch (Array::xBoundary)
85:    {
86:       cout << "Unable to process your input!" << endl;
87:    }
88:    catch (Array::xTooBig)
89:    {
90:       cout << "This array is too big..." << endl;
91:    }
92:    catch (Array::xTooSmall)
93:    {
94:       cout << "This array is too small..." << endl;
95:    }
96:    catch (Array::xZero)
97:    {
98:       cout << "You asked for an array";
99:       cout << " of zero objects!" << endl;
100:   }
101:   catch (...)
102:   {
103:      cout << "Something went wrong!" << endl;
104:   }
105:   cout << "Done." << endl;
106:   return 0;
107: }
```

输出：

```
You asked for an array of zero objects!
Done.
```

分析：

第 25～29 行声明了 4 个新类：xTooBig、xTooSmall、xZero 和 xNegative。第 56～71 行的构造函数检查被传递参数 size 的值。如果太大、太小、为零或负数，将引发异常。

在 try 块后面为每种异常提供了一条 catch 语句，但不包括异常 size 为负，这种异常将由第 101 行的“捕获所有异常”语句 catch(...)捕获。

请多次尝试将数组大小设置为不同的值。然后尝试输入–5，在这种情况下，读者可能认为将引发 xNegative 异常，然而构造函数中的测试顺序防止了这种情况发生：在判断 size<1 之前判断 size<10。要修复这种问题，可将第 61 和 62 行与第 65 行和 66 行相换，然后重新编译。

提示：构造函数被调用后，便为对象分配了内存。因此，在构造函数中引发异常可能为对象分配了内存但该对象不可用。通常应该将构造函数放在 try/catch 块中，并在发生异常时将对象标记为不可用，每个成员

函数都应检查该“有效”标记，以避免使用初始化被中断的对象进而导致其他错误。

20.4.2 异常层次结构

异常是类，因此也可以从它们派生出其他类。也许创建一个 xSize 类，然后从它派生出 xZero、xTooSmall、xTooBig 和 xNegative 更合适。这样，有些函数可以只捕获 xSize 异常，而另一些函数可以捕获具体类型的 xSize 异常。程序清单 20.6 演示了这种想法。

程序清单 20.6 类层次结构和异常

```
0:  #include <iostream>
1:  using namespace std;
2:
3:  const int DefaultSize = 10;
4:
5:  class Array
6:  {
7:    public:
8:      // constructors
9:      Array(int itsSize = DefaultSize);
10:     Array(const Array &rhs);
11:     ~Array() { delete [] pType;}
12:
13:     // operators
14:     Array& operator=(const Array&);
15:     int& operator[](int offSet);
16:     const int& operator[](int offSet) const;
17:
18:     // accessors
19:     int GetItsSize() const { return itsSize; }
20:
21:     // friend function
22:     friend ostream& operator<< (ostream&, const Array&);
23:
24:     // define the exception classes
25:     class xBoundary {};
26:     class xSize {};
27:     class xTooBig : public xSize {};
28:     class xTooSmall : public xSize {};
29:     class xZero : public xTooSmall {};
30:     class xNegative : public xSize {};
31:   private:
32:     int *pType;
33:     int itsSize;
34: };
35:
36:
37: Array::Array(int size):
38:    itsSize(size)
39: {
40:    if (size == 0)
41:       throw xZero();
```

```
42:    if (size > 30000)
43:      throw xTooBig();
44:    if (size <1)
45:      throw xNegative();
46:    if (size < 10)
47:      throw xTooSmall();
48:
49:    pType = new int[size];
50:    for (int i = 0; i < size; i++)
51:      pType[i] = 0;
52:  }
53:
54:  int& Array::operator[](int offSet)
55:  {
56:    int size = GetItsSize();
57:    if (offSet >= 0 && offSet < GetItsSize())
58:      return pType[offSet];
59:    throw xBoundary();
60:    return pType[0];  // appease MFC
61:  }
62:
63:  const int& Array::operator[](int offSet) const
64:  {
65:    int mySize = GetItsSize();
66:
67:    if (offSet >= 0 && offSet < GetItsSize())
68:      return pType[offSet];
69:    throw xBoundary();
70:
71:    return pType[0];  // appease MFC
72:  }
73:
74:  int main()
75:  {
76:    try
77:    {
78:      Array intArray(0);
79:      for (int j = 0; j < 100; j++)
80:      {
81:        intArray[j] = j;
82:        cout << "intArray[" << j << "] okay..." << endl;
83:      }
84:    }
85:    catch (Array::xBoundary)
86:    {
87:      cout << "Unable to process your input!" << endl;
88:    }
89:    catch (Array::xTooBig)
90:    {
91:      cout << "This array is too big..." << endl;
92:    }
93:
```

```
94:    catch (Array::xTooSmall)
95:    {
96:       cout << "This array is too small..." << endl;
97:    }
98:    catch (Array::xZero)
99:    {
100:      cout << "You asked for an array";
101:      cout << " of zero objects!" << endl;
102:   }
103:   catch (...)
104:   {
105:      cout << "Something went wrong!" << end_;
106:   }
107:   cout << "Done." << endl;
108:   return 0;
109: }
```

输出：

```
This array is too small...
Done.
```

分析：

重要修改在第 27～30 行，这里建立了类层次结构。类 xTooBig、xTooSmall 和 xNegative 是从 xSize 派生而来的，而 xZero 是从 xTooSmall 派生而来的。

创建了一个大小为 0 的 Array，但这是什么呢？看似捕获的异常是错误的！然而，仔细检查 catch 块将发现，它在捕获 xZero 异常前捕获 xTooSmall 异常。由于引发的是 xZero 异常，而 xZero 对象也是 xTooSmall 对象，因此它被 xTooSmall 的处理程序捕获。被处理后，异常将不会被传递给其他处理程序，因此 xZero 的处理程序永远不会被调用。

这种问题的解决方案是，仔细排列处理程序的顺序，将具体的处理程序放在前面，将不那么具体的处理程序放在后面。在这个例子中，交换 xZero 和 xTooSmall 的处理程序的位置就可以解决问题。

20.5 异常中的数据及给异常对象命名

对于被引发的异常，通常需要知道除其类型之外的其他信息，以便能够正确地应对错误。和其他类一样，异常类也可以包含数据成员、在构造函数中初始化数据成员、在任何时候读取数据成员。程序清单 20.7 演示了这一点。

程序清单 20.7 读取异常对象中的数据

```
0:  #include <iostream>
1:  using namespace std;
2:
3:  const int DefaultSize = 10;
4:
5:  class Array
6:  {
7:    public:
8:      // constructors
9:      Array(int itsSize = DefaultSize);
10:     Array(const Array &rhs);
```

```
11:     ~Array() { delete [] pType;}
12:
13:     // operators
14:     Array& operator=(const Array&);
15:     int& operator[](int offSet);
16:     const int& operator[](int offSet) const;
17:
18:     // accessors
19:     int GetitsSize() const { return itsSize; }
20:
21:     // friend function
22:     friend ostream& operator<< (ostream&, const Array&);
23:
24:     // define the exception classes
25:     class xBoundary {};
26:     class xSize
27:     {
28:       public:
29:         xSize(int size):itsSize(size) {}
30:         ~xSize(){}
31:         int GetSize() { return itsSize; }
32:       private:
33:         int itsSize;
34:     };
35:
36:     class xTooBig : public xSize
37:     {
38:       public:
39:         xTooBig(int size):xSize(size){}
40:     };
41:
42:     class xTooSmall : public xSize
43:     {
44:       public:
45:         xTooSmall(int size):xSize(size){}
46:     };
47:
48:     class xZero  : public xTooSmall
49:     {
50:       public:
51:         xZero(int size):xTooSmall(size){}
52:     };
53:
54:     class xNegative : public xSize
55:     {
56:       public:
57:         xNegative(int size):xSize(size){}
58:     };
59:
60:   private:
61:     int *pType;
62:     int  itsSize;
```

```
63: };
64:
65:
66: Array::Array(int size):
67: itsSize(size)
68: {
69:    if (size == 0)
70:       throw xZero(size);
71:    if (size > 30000)
72:       throw xTooBig(size);
73:    if (size < 1)
74:       throw xNegative(size);
75:    if (size < 10)
76:       throw xTooSmall(size);
77:
78:    pType = new int[size];
79:    for (int i = 0; i < size; i++)
80:       pType[i] = 0;
81: }
82:
83:
84: int& Array::operator[] (int offSet)
85: {
86:    int size = GetitsSize();
87:    if (offSet >= 0 && offSet < size)
88:       return pType[offSet];
89:    throw xBoundary();
90:    return pType[0];
91: }
92:
93:  const int& Array::operator[] (int offSet) const
94:  {
95:    int size = GetitsSize();
96:    if (offSet >= 0 && offSet < size)
97:       return pType[offSet];
98:    throw xBoundary();
99:    return pType[0];
100: }
101:
102: int main()
103: {
104:    try
105:    {
106:       Array intArray(9);
107:       for (int j = 0; j < 100; j++)
108:       {
109:          intArray[j] = j;
110:          cout << "intArray[" << j << "] okay..." << endl;
111:       }
112:    }
113:    catch (Array::xBoundary)
114:    {
```

```
115:      cout << "Unable to process your input!" << endl;
116:    }
117:    catch (Array::xZero theException)
118:    {
119:      cout << "You asked for an Array of zero objects!" << endl;
120:      cout << "Received " << theException.GetSize() << endl;
121:    }
122:    catch (Array::xTooBig theException)
123:    {
124:      cout << "This Array is too big..." << endl;
125:      cout << "Received " << theException.GetSize() << endl;
126:    }
127:    catch (Array::xTooSmall theException)
128:    {
129:      cout << "This Array is too small..." << endl;
130:      cout << "Received " << theException.GetSize() << endl;
131:    }
132:    catch (...)
133:    {
134:      cout << "Something went wrong, but I've no idea what!\n";
135:    }
136:    cout << "Done." << endl;
137:    return 0;
138:  }
```

输出：

```
This array is too small...
Received 9
Done.
```

分析：

xSize 的声明被修改为包含一个成员变量 itsSize（第 33 行）和一个成员函数 GetSize()（第 31 行）。另外，添加了一个构造函数，它接受一个 int 参数并初始化成员变量，如第 29 行所示。

派生类声明了一个只初始化基类的构造函数。为节省篇幅，没有声明其他函数。

第 113～135 行的 catch 语句修改为将它们捕获的异常命名为 theException，并使用这个对象来访问存储在 itsSize 中的数据。

注意：在发生异常（出现某种错误）后，如果要创建异常对象，应避免在创建它时引发同样的问题。因此，如果要创建 OutOfMemory 异常，则不应在其构造函数中分配内存。

让每条 catch 语句分别打印相应的消息很繁琐，也容易出错。这种工作应由对象来完成，它知道自己的类型和存储的值。程序清单 20.8 采用了一种面向对象程度更高的方法来解决该问题：使用虚函数让每个异常都能“做正确的事情”。

程序清单 20.8　按引用传递及在异常中使用虚函数

```
0:  #include <iostream>
1:  using namespace std;
2:
3:  const int DefaultSize = 10;
4:
5:  class Array
```

```
{
public:
// constructors
  Array(int itsSize = DefaultSize);
  Array(const Array &rhs);
  ~Array() { delete [] pType;}

  // operators
  Array& operator=(const Array&);
  int& operator[](int offSet);
  const int& operator[](int offSet) const;

  // accessors
  int GetItsSize() const { return itsSize; }

  // friend function
  friend ostream& operator<<
  (ostream&, const Array&);

  // define the exception classes
  class xBoundary {};
  class xSize
  {
   public:
     xSize(int size):itsSize(size) {}
     ~xSize(){}
     virtual int GetSize() { return itsSize; }
     virtual void PrintError()
     {
       cout << "Size error. Received: ";
       cout << itsSize << endl;
     }
   protected:
     int itsSize;
  };

  class xTooBig : public xSize
  {
   public:
     xTooBig(int size):xSize(size){}
     virtual void PrintError()
     {
       cout << "Too big! Received: ";
       cout << xSize::itsSize << endl;
     }
  };

  class xTooSmall : public xSize
  {
   public:
     xTooSmall(int size):xSize(size){}
     virtual void PrintError()
```

```
58:         {
59:           cout << "Too small! Received: ";
60:           cout << xSize::itsSize << endl;
61:         }
62:     };
63:
64:     class xZero  : public xTooSmall
65:     {
66:       public:
67:         xZero(int size):xTooSmall(size){}
68:         virtual void PrintError()
69:         {
70:           cout << "Zero!!. Received: " ;
71:           cout << xSize::itsSize << endl;
72:         }
73:     };
74:
75:     class xNegative : public xSize
76:     {
77:       public:
78:         xNegative(int size):xSize(size){}
79:         virtual void PrintError()
80:         {
81:           cout << "Negative! Received: ";
82:           cout << xSize::itsSize << endl;
83:         }
84:     };
85:
86:   private:
87:     int *pType;
88:     int  itsSize;
89: };
90:
91: Array::Array(int size):
92:   itsSize(size)
93: {
94:   if (size == 0)
95:     throw xZero(size);
96:   if (size > 30000)
97:     throw xTooBig(size);
98:   if (size < 0)
99:     throw xNegative(size);
100:  if (size < 10)
101:    throw xTooSmall(size);
102:
103:  pType = new int[size];
104:  for (int i = 0; i < size; i++)
105:    pType[i] = 0;
106: }
107:
108: int& Array::operator[] (int offSet)
109: {
```

```
110:    int size = GetitsSize();
111:    if (offSet >= 0 && offSet < GetitsSize())
112:      return pType[offSet];
113:    throw xBoundary();
114:    return pType[0];
115: }
116:
117: const int& Array::operator[] (int offSet) const
118: {
119:    int size = GetitsSize();
120:    if (offSet >= 0 && offSet < GetitsSize())
121:      return pType[offSet];
122:    throw xBoundary();
123:    return pType[0];
124: }
125:
126: int main()
127: {
128:    try
129:    {
130:      Array intArray(9);
131:      for (int j = 0; j < 100; j++)
132:      {
133:        intArray[j] = j;
134:        cout << "intArray[" << j << "] okay..." << endl;
135:      }
136:    }
137:    catch (Array::xBoundary)
138:    {
139:      cout << "Unable to process your input!" << endl;
140:    }
141:    catch (Array::xSize& theException)
142:    {
143:      theException.PrintError();
144:    }
145:    catch (...)
146:    {
147:      cout << "Something went wrong!" << endl;
148:    }
149:    cout << "Done." << endl;
150:    return 0;
151: }
```

输出：

```
Too small! Received: 9
Done.
```

分析：

在程序清单 20.8 中，第 33～37 行为 xSize 类中声明了虚方法 PrintError( )，它打印一条错误消息和对象的大小（itsSize）。在每个派生类中都覆盖了该方法。

第 141 行将异常处理程序声明为接受一个异常对象引用。通过对象引用来调用 PrintError( )时，多态使得正确的 PrintError( )版本被调用。代码更清晰、更易于理解和维护。

20.6　异常和模板

要在模板中使用异常，可以为模板的每个实例创建一个异常类，也可以使用在模板外声明的异常类。程序清单 20.9 演示了这两种方式。

程序清单 20.9　在模板中使用异常

```
0:  #include <iostream>
1:  using namespace std;
2:
3:  const int DefaultSize = 10;
4:  class xBoundary {};
5:
6:  template <class T>
7:  class Array
8:  {
9:    public:
10:     // constructors
11:     Array(int itsSize = DefaultSize);
12:     Array(const Array &rhs);
13:     ~Array() { delete [] pType;}
14:
15:     // operators
16:     Array& operator=(const Array<T>&);
17:     T& operator[](int offSet);
18:     const T& operator[](int offSet) const;
19:
20:     // accessors
21:     int GetItsSize() const { return itsSize; }
22:
23:     // friend function
24:     friend ostream& operator<< (ostream&, const Array<T>&);
25:
26:     // define the exception classes
27:
28:     class xSize {};
29:
30:   private:
31:     int *pType;
32:     int itsSize;
33: };
34:
35: template <class T>
36: Array<T>::Array(int size):
37:   itsSize(size)
38: {
39:   if (size <10 || size > 30000)
40:     throw xSize();
41:   pType = new T[size];
42:   for (int i = 0; i<size; i++)
```

```
43:       pType[i] = 0;
44: }
45:
46: template <class T>
47: Array<T>& Array<T>::operator=(const Array<T> &rhs)
48: {
49:    if (this == &rhs)
50:       return *this;
51:    delete [] pType;
52:    itsSize = rhs.GetitsSize();
53:    pType = new T[itsSize];
54:    for (int i = 0; i < itsSize; i++)
55:       pType[i] = rhs[i];
56: }
57: template <class T>
58: Array<T>::Array(const Array<T> &rhs)
59: {
60:    itsSize = rhs.GetitsSize();
61:    pType = new T[itsSize];
62:    for (int i = 0; i < itsSize; i++)
63:       pType[i] = rhs[i];
64: }
65:
66: template <class T>
67: T& Array<T>::operator[](int offSet)
68: {
69:    int size = GetitsSize();
70:    if (offSet >= 0 && offSet < GetitsSize())
71:       return pType[offSet];
72:    throw xBoundary();
73:    return pType[0];
74: }
75:
76: template <class T>
77: const T& Array<T>::operator[](int offSet) const
78: {
79:    int mysize = GetitsSize();
80:    if (offSet >= 0 && offSet < GetitsSize())
81:       return pType[offSet];
82:    throw xBoundary();
83: }
84:
85: template <class T>
86: ostream& operator<< (ostream& output, const Array<T>& theArray)
87: {
88:    for (int i = 0; i < theArray.GetitsSize(); i++)
89:       output << "[" << i << "] " << theArray[i] << endl;
90:    return output;
91: }
92:
93:
94: int main()
```

```
 95:  {
 96:    try
 97:    {
 98:      Array<int> intArray(9);
 99:      for (int j = 0; j < 100; j++)
100:      {
101:        intArray[j] = j;
102:        cout << "intArray[" << j << "] okay..." << endl;
103:      }
104:    }
105:    catch (xBoundary)
106:    {
107:      cout << "Unable to process your input!" << endl;
108:    }
109:    catch (Array<int>::xSize)
110:    {
111:      cout << "Bad Size!" << endl;
112:    }
113:
114:    cout << "Done." << endl;
115:    return 0;
116:  }
```

输出：

```
Bad Size!
Done.
```

分析：

第一个异常类（xBoundary）是在模板定义外（第 4 行）声明的。第二个异常类 xSize 是在模板定义内（第 28 行）声明的。

异常类 xBoundary 没有和模板类捆绑在一起，但可以像使用其他类那样使用它。xSize 和模板捆绑在一起，必须通过 Array 实例来调用。正如读者看到的，两条 catch 语句的语法不同：第 105 行为 catch (xBoundary)，第 109 行为 catch (Array<int>::xSize)。后者与 int 型 Array 实例捆绑在一起。

20.7　没有错误的异常

C++程序员工作之余聚集在网吧聊天时，常常会聊到这样一个话题：异常是否应用于常见状态。一些人坚持认为，就其本质而言，异常只应用于那些可预见的、程序员必须预先考虑的异常情形，而不是代码中正常处理的部分。

另一些人指出，异常提供了强大而清晰的返回方式，可跨越多层函数调用而没有内存泄漏的危险。一个常见的例子是：用户在图形用户界面（GUI）环境中请求执行操作。捕获这种请求的代码必须调用对话管理器的一个成员函数，该成员函数调用处理请求的代码，处理请求的代码调用决定使用哪个对话框的代码，这些代码又调用创建对话框的代码，而后者调用处理用户输入的代码。如果用户单击“取消”按钮，程序必须返回处理最初请求的第一个调用方法。

解决这种问题的一种方法是，将初始调用放在一个 try 块中，并将 CancelDialog 作为异常，由“取消”按钮的处理程序引发。这是安全有效的，但单击“取消”按钮是正常情形，而不是异常情形。

这常常会变得有点像宗教争论，但解决争端的合理方式是提出如下问题：以这种方式使用异常会使代码更容易还是更难理解？出现错误和内存泄露的风险更小还是更大？维护这种代码更容易还是更难？像其他决

第一样，这些决策也要求折衷，不没有惟一的显而易见的正确答案。

20.8 关于代码蜕变

代码蜕变（Code Rot）是一种众所周知的现象，指的是软件由于缺乏维护为恶化。编写得很好、经过充分调试的程序在交付几个星期后，在用户的系统上变坏了。几个月后，用户将发现程序逻辑被破坏，很多对象开始剥落。

除将源代码放在密闭的容器中之外外，惟一可采取的防护措施是，编写程序时确保以后修复时能够快速、轻松地找到问题所在。

注意：代码蜕变是程序员的一个玩笑，用于说明一个重要的经验教训。程序是极其复杂的，各种错误可能潜伏很长一段时间才暴露出来。保护措施是编写易于维护的程序。

这意味着编写的代码必须易于理解，对微妙的地方进行注释。交付 6 个月后再阅读自己编写的代码时，你将感到完全陌生，奇怪竟然有人编写了如此晦涩难懂的逻辑。

20.9 bug 和调试

几乎所有的现代开发环境都包括一个或多个功能强大的调试器。使用调试器的基本思想是：首先运行调试器，它加载源代码，然后在调试器中运行程序。这样能够看到程序中每条指令的执行情况以及检查变量在程序执行期间的变化情况。

所有的编译器都允许使用或不使用符号（symbol）进行编译。使用符号的编译命令编译器在源代码和生成的程序之间建立必要的映射关系；调试器根据映射关系指出程序的下一个操作的对应的源代码行。

全屏幕符号调试器使这项繁琐的工作变得轻松愉快。加载调试器时，它将读取所有的源代码，并将其显示在窗口中。你可以将函数调用作为一步执行，也可以命令调试器进入函数，逐行地执行其中的代码。

在大多数调试器中，可以在源代码和输出之间切换，查看每条语句的执行结果。更为强大的功能是，可以查看每个变量的当前状态，查看复杂的数据结构，查看类中成员数据的值，查看各种指针指向的内存和其他内存单元中的实际值。可以在调试器中使用多种控制方式，包括设置断点、设置监视点、查看内存和汇编代码。

20.9.1 断点

断点命令调试器在到达某行代码后暂停执行程序。这让程序一直运行到被怀疑的代码行。断点可帮助你分析在关键代码行执行前后，变量的当前状态。

20.9.2 监视点

可让调试器显示某个变量的值或某个变量被读写时暂停程序执行。监视点让你能够设置这些条件，甚至在程序运行时修改变量的值。

20.9.3 查看内存

有时，查看内存中存储的实际值很重要。现代调试器可以用实际变量的格式显示值，即将字符串显示为字符、长整数显示为数字而不是 4 个字节的内容等。高级 C++调试器甚至能够显示整个对象，提供包括 this 指针在内的所有成员变量的当前值。

20.9.4 查看汇编代码

虽然要发现 bug，只需查看源代码即可，但当其他手段都不管用时，可以命令调试器显示根据每行源代

码生成的汇编代码。可以查看内存寄存器和标记，并在必要时深入研究程序的内部工作原理。

读者应学会使用调试器。它是战胜 bug 的最强有力武器。运行阶段错误是最难发现和消除的，强大的调试器让你几乎可能找到所有的错误，虽然查找起来可能并不那么轻松。

20.10　小　　结

本章介绍了有关创建和使用异常的基本知识。异常是可以在程序的某些地方创建并引发的对象，在这些地方，执行的代码不能处理错误或发生了其他异常情况。程序的其他部分（调用堆栈的上方）实现了 catch 块，它能够捕获异常并采取适当的措施。

异常是用户创建的常规对象，可以按值或引用进行传递。它们可以包含数据和方法，catch 块可以根据这些数据决定如何处理异常。

可以创建多个 catch 块，但异常与某个 catch 块匹配后，就被视为已被处理，不会被传递给后面的 catch 块。应按正确的顺序排列 catch 块，将具体的 catch 块放在前面，让通用的 catch 块处理那些其他 catch 块不能处理的异常，这非常重要。

本章还介绍了有关符号调试器的基本知识，包括使用监视点和断点等。这些工具让你能够将注意力集中到程序中导致错误的部分，查看程序执行期间变量的变化情况。

20.11　问　与　答

问：为什么要引发异常？为什么不就地处理错误？

答：通常，在程序的不同地方可能导致相同的错误。异常让你能够集中处理这些错误。另外，导致错误的地方可能不是决定如何处理错误的最佳位置。

问：为什么要生成异常对象？为什么不只传递错误码？

答：异常对象比错误码功能更强大、更灵活。它们能够传递更多的信息，同时可以使用构造函数/析构函数机制来分配和释放正确处理异常所需的资源。

问：为什么不将异常用于非错误状态？能够快速返回以前的代码区域，即使在非异常状态下也如此，不是很方便吗？

答：是的。有些 C++程序员仅为此目的而使用异常。这样做是危险的，当堆栈被解退时无意中将某些对象留在自由存储区中，导致内存泄漏。使用细致的编程技术和优秀的编译器，通常可以避免这种情况。然而，这是个人审美观问题，有些程序员认为，异常从其本质上说不能用于正常状态。

问：必须在引发异常的 try 块后面捕获它的吗？

答：不，可以在调用栈的任何地方捕获异常。当堆栈被解退时，异常传递将沿堆栈向上传递，直到被处理为止。

问：在可以使用 cout 和诸如此类的其他语句的情况下，为什么要使用调试器？

答：调试器提供了一种功能强大得多的机制，让你能够以步进方式执行程序并监视变量的变化情况，而无需使用成千上万条的调试语句，使代码混乱不堪。另外，每次添加和删除代码行时都会带来一定的风险。如果你调试的目的只是为了修复问题，但在删除添加的 cout 语句时不小心删除了真正的代码行，将带来新的问题。

20.12　作　　业

作业包括测验和练习，前者帮助加深读者对所学知识的理解，后者提供了使用新学知识的机会。请尽量

先完成测验和练习题，然后再对照附录 D 中的答案，继续学习下一章之前，请务必弄懂这些答案。

20.12.1 测验

1. 什么是异常？
2. 什么是 try 块？
3. 什么是 catch 语句？
4. 异常对象可包含什么信息？
5. 异常对象在什么时候被创建？
6. 应按值还是引用来传递异常对象？
7. 如果 catch 语句捕获基类异常对象，它能捕获到派生类异常对象吗？
8. 如果使用了两条 catch 语句，一条捕获基类异常对象，另一条捕获派生类异常对象，应将哪条语句放在前面？
9. catch(...)的含义是什么？
10. 什么是断点？

20.12.2 练习

1. 编写一个 try 块、一条 catch 语句和一个简单异常。
2. 修改练习 1 的答案，在异常类中添加数据和一个存取器函数，并在 catch 块中使用它。
3. 将练习 2 中的类修改为异常类层次结构。修改 catch 块，使其捕获派生类对象和基类对象。
4. 修改练习 3 中的程序，使其包含 3 级函数调用。
5. 查错：下面的代码有什么错误？

```
#include "stringc.h"          // our string class

class xOutOfMemory
{
   public:
      xOutOfMemory( const String& where ) : location( where ){}
      ~xOutOfMemory(){}
      virtual String where(){ return location };
   private:
      String location;
}

int main()
{
   try
   {
      char *var = new char;
      if ( var == 0 )
         throw xOutOfMemory();
   }
   catch( xOutOfMemory& theException )
   {
      cout << "Out of memory at " << theException.location() << endl;
   }
   return 0;
}
```

该程序清单演示了如何处理内存耗尽错误。