第19章 异常处理

本章讨论异常处理子系统。异常处理使你可以用一种有序的方式管理运行错误。利用异常处理，当出现错误时，程序可以自动调用错误处理例程。异常处理的主要优点是能够自动化许多错误处理代码，而这些代码以前要在大型程序中通过“手工”编写。

19.1 异常处理基础

C++异常处理依赖三个关键字：try，catch和throw。笼统地说，监测异常情况的程序语句包含在一个try块中，如果try块中出现异常（即错误），该异常就会被抛出（利用throw）。利用catch可以捕获并处理异常。下面详细讨论这些情况。

想要监测异常情况的代码必须从try结构块内执行（从try结构块中调用的函数也可以抛出一个异常），监控代码抛出的异常可被catch语句捕获，该语句紧跟在抛出异常的try语句之后。try和catch语句的一般形式如下所示：

```
try {
  // try block
}
catch (type1 arg) {
  // catch block
}
catch (type2 arg) {
  // catch block
}
catch (type3 arg) {
  // catch block
}
.
.
.
catch (typeN arg) {
  // catch block
}
```

try语句块可以只包含某个函数中的几行语句，也可以像在一个try块内封装代码一样把main( )函数完全封装起来（此举可以有效地监控整个程序）。

当异常被抛出后，相应的catch语句将捕获并处理这个异常。与try语句块相关的catch语句可以有多个，究竟使用哪一个catch语句取决于异常的类型。也就是说，如果某个catch语句指定的数据类型与抛出的异常类型相匹配，那么该catch语句将被执行（其他所有catch语句将被略过）。当一个异常被捕获时，参数arg将接收它的值。任何数据类型都可以被捕获，包括你所创建的类。如果没有抛出异常（即，try语句块中没有出现错误），则不会执行任何catch语句。

throw语句的一般形式如下所示：

```
throw exception;
```

throw语句产生由exception指定的异常。如果要捕获该异常，throw语句必须要么从try语句块自身中执行，要么从该try语句块内调用的函数中执行（直接和间接）。

如果抛出了一个没有合适的catch语句的异常，程序将会异常终止。如果抛出一个未被处理的异常，则会调用标准库函数terminate()。默认情况下，terminate()通过调用abort()终止程序，但是也可以指定自己的终止处理程序，相应内容在本章后面讨论。

下面这个简单的例子说明了C++中异常处理的操作方式。

```
// A simple exception handling example.
#include <iostream>
using namespace std;

int main()
{
  cout << "Start\n";

  try { // start a try block
    cout << "Inside try block\n";
    throw 100; // throw an error
    cout << "This will not execute";
  }
  catch (int i) { // catch an error
    cout << "Caught an exception -- value is: ";
    cout << i << "\n";
  }

  cout << "End";

  return 0;
}
```

这个程序的输出如下所示：

```
Start
Inside try block
Caught an exception -- value is: 100
End
```

让我们仔细看一看这个程序。可以看到，该程序有一个包含三条语句的try语句块和一个处理整型异常的catch(int i)语句，在try语句块的三条语句中，只有二条语句会执行，即第一条cout语句和throw语句。一旦抛出一个异常，控制流程就转移到catch表达式上，try语句块被终止。也就是说，catch不是通过调用执行的，而是程序执行转移至此（程序的堆栈将根据需要被重置以实现这种转移），因此，跟在throw后面的cout语句永远不会执行。

一般来说，catch语句中的代码总是要努力通过适当的操作来纠正某种错误。如果错误可以纠正，程序将继续执行catch语句后面的语句。然而，如果错误不能纠正，catch语句块将通过调用exit()或abort()终止程序的执行。

我们在前面曾经提到，异常的类型必须与某个catch语句指定的类型相匹配。例如，在前面的例子中，如果将catch语句中的类型改为double，异常将不能被捕获，而且程序将异常终

止。改变后的程序如下所示：

```
// This example will not work.
#include <iostream>
using namespace std;

int main()
{
  cout << "Start\n";

  try { // start a try block
    cout << "Inside try block\n";
    throw 100; // throw an error
    cout << "This will not execute";
  }
  catch (double i) { // won't work for an int exception
    cout << "Caught an exception -- value is: ";
    cout << i << "\n";
  }

  cout << "End";

  return 0;
}
```

由于catch(double i)语句不能捕获整型异常，所以该程序将产生下面的输出结果（当然，描述异常终止的精确信息因编译器的不同而不同）：

```
Start
Inside try block
Abnormal program termination
```

当在try语句块内部调用的函数抛出异常时，异常也可以在try语句块的外部被抛出。例如，下面的程序是正确的：

```
/* Throwing an exception from a function outside the
   try block.
*/
#include <iostream>
using namespace std;

void Xtest(int test)
{
  cout << "Inside Xtest, test is: " << test << "\n";
  if(test) throw test;
}

int main()
{
  cout << "Start\n";

  try { // start a try block
    cout << "Inside try block\n";
    Xtest(0);
    Xtest(1);
```

```
    Xtest(2);
  }
  catch (int i) { // catch an error
    cout << "Caught an exception -- value is: ";
    cout << i << "\n";
  }

  cout << "End";

  return 0;
}
```

这个程序的输出如下所示：

```
Start
Inside try block
Inside Xtest, test is: 0
Inside Xtest, test is: 1
Caught an exception -- value is: 1
End
```

try语句块可以被局部化为一个函数，如果是这样，当每次进入这个函数时，与该函数相应的异常处理将被重置。例如，让我们看一看下面的程序：

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

// Localize a try/catch to a function.
void Xhandler(int test)
{
  try{
    if(test) throw test;
  }
  catch(int i) {
    cout << "Caught Exception #: " << i << '\n';
  }
}

int main()
{
  cout << "Start\n";

  Xhandler(1);
  Xhandler(2);
  Xhandler(0);
  Xhandler(3);

  cout << "End";

  return 0;
}
```

这个程序的输出如下所示：

```
Start
Caught Exception #: 1
```

```
Caught Exception #: 2
Caught Exception #: 3
End
```

可以看到，抛出了三个异常，函数在抛出每一个异常之后返回。当再一次调用该函数时，异常处理被重置。

只有当catch语句捕获了一个异常时，与该语句相关的代码才会被执行，理解这一点非常重要。否则，程序的执行将跳过catch语句（即，程序不会按照顺序执行catch语句）。例如，下面的程序没有抛出异常，因此也就不会执行catch语句：

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

int main()
{
 cout << "Start\n";

  try { // start a try block
    cout << "Inside try block\n";
    cout << "Still inside try block\n";
  }
  catch (int i) { // catch an error
    cout << "Caught an exception -- value is: ";
    cout << i << "\n";
  }
  cout << "End";

  return 0;
}
```

上面这个程序的输出如下所示：

```
Start
Inside try block
Still inside try block
End
```

由此可见，该程序执行时跳过了catch语句。

19.1.1 捕获类

异常可以是任何类型的，包括你所创建的类。事实上，在实际使用的程序中，大多数异常类型都是类而不是内置类型。给异常定义类的最常见的原因是要创建一个描述所出现错误的对象，异常处理程序可以利用这个信息处理发生的错误。下面的例子对此进行了说明：

```
// Catching class type exceptions.
#include <iostream>
#include <cstring>
using namespace std;

class MyException {
public:
  char str_what[80];
  int what;
```

```
  MyException() { *str_what = 0; what = 0; }

  MyException(char *s, int e) {
    strcpy(str_what, s);
    what = e;
  }
};

int main()
{
  int i;

  try {
    cout << "Enter a positive number: ";
    cin >> i;
    if(i<0)
      throw MyException("Not Positive", i);
  }
  catch (MyException e) { // catch an error
    cout << e.str_what << ": ";
    cout << e.what << "\n";
  }

  return 0;
}
```

下面是这个例子的运行结果：

```
Enter a positive number: -4
Not Positive: -4
```

该程序提示用户输入一个正数。如果输入了一个负数，程序将创建一个MyException类的对象以描述这种错误，因此，MyException将错误信息封装起来，这些信息将被异常处理程序使用。一般来说，你所创建的异常类应该能够封装某种错误信息，从而使异常处理程序能够有效地对错误做出反应。

19.1.2 使用多条catch语句

前面曾经提到，一个try语句可以有多条对应的catch语句。事实上，这种情况很常见。然而，每一个catch语句必须捕获一种不同类型的异常。例如，下面的程序既可以捕获整型异常，也可以捕获字符串型的异常。

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

// Different types of exceptions can be caught.
void Xhandler(int test)
{
  try{
    if(test) throw test;
    else throw "Value is zero";
  }
  catch(int i) {
```

```
    cout << "Caught Exception #: " << i << '\n';
  }
  catch(const char *str) {
    cout << "Caught a string: ";
    cout << str << '\n';
  }
}

int main()
{
  cout << "Start\n";

  Xhandler(1);
  Xhandler(2);
  Xhandler(0);
  Xhandler(3);

  cout << "End";

  return 0;
}
```

这个程序的输出如下所示：

```
Start
Caught Exception #: 1
Caught Exception #: 2
Caught a string: Value is zero
Caught Exception #: 3
End
```

可以看出，每条catch语句只对其自己的类型做出响应。

一般来说，catch表达式按照它们在程序中出现的顺序被检验，只有与异常相匹配的语句才会被执行，所有其他的catch语句块将被忽略。

19.2 处理派生类异常

因为基类的catch子句可以与基类派生的类相匹配，所以当试图捕获涉及基类及其派生类的异常类型时，必须对catch语句的排列顺序格外谨慎。如果既要捕获基类异常，又要捕获派生类异常，则应该在catch序列中首先放置派生类。否则，基类的catch语句将会捕获所有的派生类异常。例如，看一看下面的程序：

```
// Catching derived classes.
#include <iostream>
using namespace std;

class B {
};

class D: public B {
};

int main()
{
```

```
  D derived;

  try {
    throw derived;
  }
  catch(B b) {
    cout << "Caught a base class.\n";
  }
  catch(D d) {
    cout << "This won't execute.\n";
  }

return 0;
}
```

这里，因为derived是一个将B作为基类的对象，所以它将被第一个catch子句捕获，第二个catch子句将永远不会执行。有些编译器在遇到这种情况时会发出一个警告信息，还有些编译器则会报错。无论采用哪一种编译器，要想纠正这种情况，须将catch子句的顺序倒转过来。

19.3 异常处理选项

C++的异常处理还有几种附加的功能，这些功能可以使异常处理更加简便，下面将对此进行讨论。

19.3.1 捕获所有异常

有时候我们想使异常处理程序捕获所有异常，而不是只捕获某一种类型的异常。这个目的很容易实现，只需采用下面的catch语句：

```
catch(...) {
// process all exceptions
}
```

其中，省略号可以匹配任何数据类型。下面的程序说明了catch(...)语句的使用情况：

```
// This example catches all exceptions.
#include <iostream>
using namespace std;

void Xhandler(int test)
{
  try{
    if(test==0) throw test; // throw int
    if(test==1) throw 'a'; // throw char
    if(test==2) throw 123.23; // throw double
  }
  catch(...) { // catch all exceptions
    cout << "Caught One!\n";
  }
}

int main()
{
```

```
  cout << "Start\n";

  Xhandler(0);
  Xhandler(1);
  Xhandler(2);

  cout << "End";

  return 0;
}
```

这个程序的输出如下所示：

```
Start
Caught One!
Caught One!
Caught One!
End
```

可以看到，利用一条catch语句可以捕获3个throw语句抛出的所有异常。

catch(...)有一种很好的用法，即将其作为一组catch语句的最后一条语句，此时它提供一种实用的默认或“捕获所有异常”语句。例如，下面这个程序与前面的程序略有不同，用于显式地捕获整型异常，但却利用catch(...)捕获所有其他异常。

```
// This example uses catch(...) as a default.
#include <iostream>
using namespace std;

void Xhandler(int test)
{
  try{
    if(test==0) throw test; // throw int
    if(test==1) throw 'a'; // throw char
    if(test==2) throw 123.23; // throw double
  }
  catch(int i) { // catch an int exception
    cout << "Caught an integer\n";
  }
  catch(...) { // catch all other exceptions
    cout << "Caught One!\n";
  }
}

int main()
{
  cout << "Start\n";

  Xhandler(0);
  Xhandler(1);
  Xhandler(2);

  cout << "End";

  return 0;
}
```

这个程序的输出如下所示：

```
Start
Caught an integer
Caught One!
Caught One!
End
```

这个例子说明，如果不想显式地处理异常，将catch(...)作为默认值是一种捕获所有异常的好办法。此外，通过捕获所有异常，可以防止由于某个未处理的异常而引起的程序异常终止。

19.3.2 限制异常

可以限制函数抛出的异常类型。事实上，还可以阻止函数抛出任何异常。为了实现这些限制，必须在函数定义中添加一个throw子句，其一般的语法形式如下所示：

```
ret-type func-name(arg-list) throw(type-list)
{
  // ...
}
```

这里，只有包含在用逗号分隔的类型列表（type-list）中的数据类型可以被函数抛出。如果抛出任何其他类型的表达式，将导致程序异常终止。如果不想使函数抛出任何异常，则应使用空的数据类型列表。

试图抛出一个不被函数支持的异常时将会调用标准库函数unexpected()。默认情况下，这将调用abort()，导致程序的异常终止。然而，如果愿意，可以指定自己的异常处理程序，本章后面将会对此进行说明。

下面的程序说明了如何限制函数抛出的异常类型。

```
// Restricting function throw types.
#include <iostream>
using namespace std;

// This function can only throw ints, chars, and doubles.
void Xhandler(int test) throw(int, char, double)
{
  if(test==0) throw test; // throw int
  if(test==1) throw 'a'; // throw char
  if(test==2) throw 123.23; // throw double
}

int main()
{
  cout << "start\n";

  try{
    Xhandler(0); // also, try passing 1 and 2 to Xhandler()
  }
  catch(int i) {
    cout << "Caught an integer\n";
  }
  catch(char c) {
```

```
    cout << "Caught char\n";
  }
  catch(double d) {
    cout << "Caught double\n";
  }
    cout << "end";

  return 0;
}
```

在这个程序中，函数Xhandler( )只能抛出整型、字符型和双精度型的异常。如果试图抛出任何其他类型的异常，程序将会异常终止（也就是说，将会调用unexpected( )）。为了进一步理解此例，可以从类型列表中删掉int，然后再重新运行一次这个程序。

一个函数只能被限制为抛回给调用它的try语句块的异常类型，理解这一点非常重要。也就是说，只要异常可以在函数内部捕获，那么该函数中的try语句块可以抛出任何异常类型。这个限制仅适用于在函数外抛出异常的情况。

下面对Xhandler( )的修改将阻止它抛出任何异常。

```
// This function can throw NO exceptions!
void Xhandler(int test) throw()
{
  /* The following statements no longer work. Instead,
     they will cause an abnormal program termination. */
  if(test==0) throw test;
  if(test==1) throw 'a';
  if(test==2) throw 123.23;
}
```

19.3.3 再次抛出异常

如果希望在异常处理程序中再次抛出一个表达式，则可以通过调用throw实现，这会使当前的异常传递到一个外部的try/catch序列上。之所以这样做，最常见的原因是允许多个处理程序访问该异常。例如，一个异常处理程序或许只处理异常的一个方面，而另一个处理程序处理该异常的另一个方面。异常只能从catch语句块中（或从catch语句块内调用的函数中）被再次抛出。当再次抛出一个异常时，该异常将不再被同一个catch语句所捕获，它将传递给外部的catch语句。下面的程序说明了如何再次抛出一个异常，该异常是一个char *异常。

```
// Example of "rethrowing" an exception.
#include <iostream>
using namespace std;

void Xhandler()
{
  try {
    throw "hello"; // throw a char *
  }
  catch(const char *) { // catch a char *
    cout << "Caught char * inside Xhandler\n";
    throw ; // rethrow char * out of function
  }
```

```
}
int main()
{
  cout << "Start\n";
  try{
    Xhandler();
  }
  catch(const char *) {
    cout << "Caught char * inside main\n";
  }
  cout << "End";
  return 0;
}
```

这个程序的输出如下所示：

```
Start
Caught char * inside Xhandler
Caught char * inside main
End
```

19.4 理解 terminate( )和 unexpected( )

前面曾经提到，在异常处理过程中如果出现错误，就会调用terminate( )和unexpected( )。这两个函数由标准C++库所提供，其原型如下所示：

```
void terminate( );
void unexpected( );
```

这些函数需要<exception>头文件。

只要异常处理子系统不能找到与异常相匹配的catch语句，terminate( )函数就会被调用。如果最初没有异常抛出，而程序却试图再次抛出一个异常，此时也会调用terminate( )函数。另外，在许多其他难以说清的情况下也会调用terminate( )。例如，在由于发生异常而退栈的过程中，当对象被销毁的构造函数抛出一个异常时，就会出现这种情况。一般来说，当没有其他异常处理程序可用时，terminate( )是最后一种手段。默认情况下，terminate( )调用abort( )。

当函数试图抛出一个不在throw列表中的异常时，会调用unexpected( )函数。默认情况下，unexpected( )调用terminate( )。

19.4.1 设置 terminate 和 unexpected 处理程序

terminate( )和unexpected( )函数通过调用其他函数来处理错误。正如我们刚刚讲到的那样，terminate( )调用abort( )，unexpected( )调用terminate( )，因此，默认时，这两个函数在发生异常处理错误时都可以终止程序的执行。然而，可以改变被terminate( )和unexpected( )调用的函数，这样可以使程序完全控制异常处理子系统。

为了改变终止处理程序，可以使用set_terminate( )，如下所示：

terminate_handler set_terminate(terminate_handler *newhandler*) throw( );

这里，newhandler是一个指向新的终止处理程序的指针。该函数返回一个指向老的终止处理程序的指针。新的终止处理程序的类型必须是terminate_handler，其定义如下所示：

```
typedef void (*terminate_handler) ( );
```

终止处理程序只需做一件事，即终止程序的执行，它绝对不能以任何形式返回到程序或继续执行程序。

为了改变意外（unexpected）处理程序，可以使用set_unexpected()，如下所示：

```
unexpected_handler set_unexpected(unexpected_handler newhandler) throw( );
```

这里，newhandler是一个指向新的意外处理程序的指针。该函数返回一个指向老的意外处理程序的指针。新的意外处理程序的类型必须是unexpected_handler，其定义如下所示：

```
typedef void (*unexpected_handler) ( );
```

这个处理程序可以抛出一个异常、终止程序的执行或调用terminate()。然而，它决不能返回到程序。

set_terminate()和set_unexpected()都需要使用<exception>头文件。

下面给出了一个定义terminate()处理程序的例子：

```
// Set a new terminate handler.
#include <iostream>
#include <cstdlib>
#include <exception>
using namespace std;

void my_Thandler() {
  cout << "Inside new terminate handler\n";
  abort();
}

int main()
{
  // set a new terminate handler
  set_terminate(my_Thandler);

  try {
    cout << "Inside try block\n";
    throw 100; // throw an error
  }
  catch (double i) { // won't catch an int exception
    // ...
  }

  return 0;
}
```

这个程序的输出如下所示：

```
Inside try block
Inside new terminate handler
abnormal program termination
```

19.5 uncaught_exception( )函数

C++异常处理子系统还提供另外一个比较实用的函数：uncaught_exception( )，这个函数的原型如下所示：

```
bool uncaught_exception( );
```

如果一个异常被抛出但尚未被捕获，该函数返回true。一旦异常被捕获，该函数返回false。

19.6 exception和bad_exception类

当一个由C++标准库提供的函数抛出一个异常时，它将是一个从基类exception派生的对象。bad_exception类的对象可以被意外处理程序抛出。这些类都需要<exception>头文件。

19.7 异常处理的应用

异常处理是要提供一种结构化方法，通过这种方法可以使程序处理异常事件。这意味着当出现错误时，错误处理程序必须做一些合理的工作。例如，考虑下面这个简单的程序，该程序要求输入两个数，然后用第二数除第一个数。它使用异常处理来管理被0除产生的错误。

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

void divide(double a, double b);

int main()
{
  double i, j;

  do {
    cout << "Enter numerator (0 to stop): ";
    cin >> i;
    cout << "Enter denominator: ";
    cin >> j;
    divide(i, j);
  } while(i != 0);

  return 0;
}

void divide(double a, double b)
{
  try {
    if(!b) throw b; // check for divide-by-zero
    cout << "Result: " << a/b << endl;
  }
  catch (double b) {
    cout << "Can't divide by zero.\n";
  }
}
```

上面的程序虽然简单，但却说明了异常处理的本质。因为被0除是非法的，所以如果输入

的第二个数是0，程序将不能继续执行。遇到这种情况时，处理异常的方法是不执行这个除法操作（这将导致程序异常终止）并通知用户发生了错误，然后程序再次提示用户输入另外两个数，这样就可以用一种有序的方式对错误进行处理，使用户可以继续执行程序。这些基本概念同样适用于更复杂的异常处理应用。

当出现灾难性错误时，异常处理对从深层嵌套的例程退出尤其有用。在这方面，C++异常处理取代了十分不便的基于C的setjmp( )和longjmp( )函数。

记住，使用异常处理的关键在于提供一种有序的处理错误的方法，这意味着在可能的情况下纠正错误。