

异常的抛出和处理

C++支持异常处理。所谓异常 exception 就是程序运行中遇到的错误。程序中很多的异常是可以预期的，如编写容器类的人可以预期使用者可能出现数组越界，编写文件输入输出流的人可以预期打开文件的时候可能失败（由于文件不存在或者文件已被别的程序打开）。对于可以预期的异常，所谓的异常处理 exception handling 包括：对错误发生的检测，和当错误发生的时候执行的善后工作。

在 C++以及大多数当代的程序设计语言中，异常处理包括以下两个过程：

1. 检测异常的发生，当发生的时候终止正在执行的出现了异常的程序，并报告这个异常，即异常的抛出。
2. 当异常的发生了并且出现异常的程序被终止后，程序的执行被转移到处理对应的异常的程序。

上面提到的“处理对应的异常的程序”是由程序员编写的，程序员可以选择任何方式处理一个异常。例如，当打开文件失败的时候，处理的方式可以是等 10 毫秒再尝试打开，可以是弹出一个对话框要求用户选择一个新的文件名，也可以是简单地弹出一个对话框告诉用户文件不存在、程序运行失败。

C++中提供一种当前最流行的异常处理的语法结构，它能够使得报告和处理异常的程序更加简单。本章的内容如下：首先，我们通过一个简单的例子以说明 C++中提供的抛出异常和处理异常的基本语法；然后我们分别说明抛出异常和处理异常的语法细节和重点的使用方法；接着我们讨论 C++的异常处理方式的优点；最后我们分析异常抛出和处理的过程中对象的释构。

1. 报告和处理异常的例子

我们将会通过比较使用了 C++中异常处理机制和不使用的两个程序，学习异常处理机制的简单用法，和初步了解为什么需要异常处理机制。

在 C++出现之前的 C 语言是没有异常处理机制的。如果我们在只支持 C 语言的单片机（非常简单的计算机，它们通常用来控制空调、电饭锅等设备）上编程，那么我们可能要写以下这个“老式”的程序。

这下面的例子包括两个部分的程序：第一个函数 `read` 打开一个文件，从文件中读入一个浮点数；第二个函数 `main` 根据程序的运行是否有错进行控制台输出。由于函数 `read` 读取文件，它存在预期的运行错误。所以，它除了需要告诉调用者所读的浮点数，还需要告诉调用者运行的时候是否出错了。

函数 read 中存在两种错误：打开文件错误，和数值错误（假设规定程序读入的应该是正数）。

```
1 void read(double & value, int & errorCode) {
2     errorCode = 0; // means no error
3     ifstream input("data.txt");
4     if (input.fail()) {
5         errorCode = 1; // error openning file
6     }
7     else {
8         input >> value;
9         if (value < 0) {
10             errorCode = 2; // data error
11         }
12     }
13     input.close();
14 }
15
16 int main() {
17     double value;
18     int errorCode;
19     read(value, errorCode);
20     if (errorCode == 0) {
21         cout << "value = " << value;
22     }
23     else if (errorCode == 1) {
24         cout << "Cannot open data file" << endl;
25     }
26     else if (errorCode == 2) {
27         cout << "Data is corrupted" << endl;
28     }
29 }
```

现在，我们把上面的程序改为使用 C++异常处理的版本。

```
1 /* exception types */
2 class FailToOpenAFileException {};
3 class DataCorruptedException {};
4
5 void read(double & value) {
6     ifstream input("data.txt");
7     if (input.fail()) {
```

```

8          /* report a type of error by throwing
9             an object of a corresponding
10            exception type. */
11         throw FailToOpenAFileException();
12     }
13     input >> value;
14     input.close();
15     if (value < 0) {
16         throw DataCorruptedException();
17     }
18 }
19
20 int main() {
21     try {
22         double value;
23         read(value);
24         cout << "value = " << value;
25     }
26     catch (FailToOpenAFileException & ex) {
27         cout << "Cannot open data file" << endl;
28     }
29     catch (DataCorruptedException & ex) {
30         cout << "Data is corrupted" << endl;
31     }
32 }

```

在第 2-3 行，我们定义了两个类型，它们分别代表两种错误。在第 11 行，我们看到，当打开文件错误发生的时候，使用了一个我们没有见过的语句：throw 语句。这里 **throw** 是一个保留字，它的后面通常跟着一个对象，这个对象的类型通常代表某种异常。在第 11 行，**throw** 后面跟着的是一个 **FailToOpenAFileException** 类型的匿名对象。**throw** 语句是用来报告一个异常的，也叫做“抛出一个异常”。也就是说，抛出一个代表某种异常的类的对象，就是向系统报告被这个类所代表的异常。

如果一个函数产生一个错误，而且在这个函数里面不能处理这个错误，那么这个函数应该抛出一个异常。当异常被抛出后，没法处理这个异常的程序会马上结束，程序的控制权会被转移到能处理这个异常的地方（后面我们将马上讨论的 `catch-block`）。如果第 11 行的 `throw` 语句被执行，那么函数 `read` 会马上结束。在第 16 行中，我们看到，当读入的数据是被破坏的数据的时候，函数 `read` 会抛出另外一种异常。

在从第 20 行开始的函数 `main` 中，23 行调用了函数 `read`，26-31 行分别捕捉和处理了两种在函数 `read` 中可能抛出的异常。第 21 到 25 行的程序块叫做一个 `try-block`，如果我们要捕捉一些异常，那么我们需要把可能抛出异常的程序片段放在一个 `try-block` 里面。一个 `try-block` 后面可以跟一个或以上的 `catch-block`，每个 `catch-block` 都有一个参数，可以捕捉类型能匹配这个参数的异常。例如，当函数 `read` 中抛出类型为 `FailToOpenAFileException` 的异常时，第一个 `catch-block` 将捕捉这个异常，然后第 26-28 行会被运行。

在 C++ 中，被抛出的异常有以下的特点：

1. 如果一个异常在一个函数里面没有被捕捉，那么这个函数将会结束，而且这个异常将会继续在调用该函数的函数中被抛出。
2. 当一个异常被抛出，在这个异常被某个 `catch-block` 捕

捉并处理之前，任何中间的语句都不会被执行。

例如，如果第 11 行中抛出异常，那么：由于函数 `read` 中没有处理这个异常，函数 `read` 将结束，并且这个异常将会在 23 行继续被抛出；由于异常在 23 行再被抛出，而它将会再 26 行被捕捉。异常被捕捉前的 24 行不会被执行。26 行能捕捉所抛出的异常，是由于第 26 行的 `catch-block` 的参数与所抛出的异常的类型完全相同。在 26-28 行的 `catch-block` 执行完后，整个 `try-catch` 语句（第 21-31 行）就结束了，接着执行的将是 31 行以后的语句。

从这个例子我们可以看到异常处理的其中一个好处：它可以使我们把正常执行的程序模块与处理各种错误的程序模块分开。例如，写 22-24 行的正常执行代码的时候，我们无需考虑处理如何各种可能的错误。

下面我们分别详细介绍异常的抛出和捕捉。

2. 异常的抛出

异常的抛出提供除了 `return` 以外的另一种程序返回的方式：一种出错时的返回方式。不同的是 `return` 终止当前的函数后，会继续调用者函数的执行；而在当前函数里没有处理的异常，会在终止当前的函数后，继续在调用者函数中抛出。如果所有函数都不能捕捉一个异常，那么这个异常会一直往

外抛，直到被 `main` 函数抛出。这个时候，系统会调用名为 `terminate` 的过程来终止程序。也就是说，如果抛出的是一个整个程序都没有捕捉的异常，那么抛异常的效果与调用 `exit(0)` 的效果同样：它们都会终止程序。请看下面的程序和它的输出。

```
1 void errorInside() {  
2     cout << "errorInside() line A" << endl;  
3     int errorCode = 10;  
4     throw errorCode;  
5     cout << "errorInside() line B" << endl;  
6 }  
7  
8 int main() {  
9     cout << "main() line A" << endl;  
10    errorInside();  
11    cout << "main() line B" << endl;  
12 }  
输出  
main() line A  
errorInside() line A  
libc++abi.dylib: terminate called throwing an exception
```

在第 4 行，我们可以看到，在 C++ 中抛出的除了可以是异常对象外，也可以是整数或其它基本数据类型的变量，虽然使用基本数据类型的异常比较罕见。当异常在第 4 行被抛出后，函数 `errorInside` 被终止。在此之后，这个异常在调用者 `main` 函数中的第 10 行继续被抛出，这使得 `main` 函数也被终止。当异常被抛出 `main` 函数而被系统捕捉的时候，系统会终止该程序。从程序的输出可以看到，异常被抛出后的语句（第 5 和 11 行）都没有被执行。

C++中定义了一些异常的类型。我们定义异常类型的时候可以继承这些 C++定义的异常类型，它们有时可以获得更好的系统支持。

```
1 #include <iostream>
2 #include <stdexcept> // standard exception
3 using namespace std;
4
5 void errorInside() {
6     cout << "errorInside() line A" << endl;
7     runtime_error error(":(");
8     cout << error.what() << endl;
9     throw error;
10    cout << "errorInside() line B" << endl;
11 }
12
13 int main() {
14     cout << "main() line A" << endl;
15     errorInside();
16     cout << "main() line B" << endl;
17 }
```

输出

```
main() line A
errorInside() line A
:(
libc++abi.dylib: terminate called throwing an exception
```

从上面的程序我们可以看到，在第 2 行中包含了 C++的类库 `stdexcept` 后，我们可以使用 C++定义的异常 `runtime_error` 类。这个类有一个构造函数，它的参数为一个 `string` 的对象。第 7 行中可以使用字符串常量作为参数是因为 `string` 类中具有构造函数 `string(const char *)`，它使得字符串常量可以隐式地转换为 `string` 的对象，从而使得 `runtime_error` 中的构造函数可以在第 7 行匹配。

如第 8 行所示，类 `runtime_error` 具有对象函数 `what`，它返回对象被构造时传入构造函数的 `string` 的对象。函数 `what` 的作用是：当 `runtime_error` 的异常对象被捕捉的时候，可以通过函数 `what` 得到抛出异常的函数所提供的、关于这个异常的信息。

3. 异常的捕捉和处理

我们要在程序中的某个恰当的地方捕捉异常，使它不被传递到 `main` 函数以外，造成程序的终止。C++中捕捉异常的语法结构叫做 `try-catch` 语句，它由一个 `try-block` 后面紧接者一个或多个 `catch-block` 组成。如果要捕捉一个异常，那么抛出这个异常的语句或函数调用必须出现在一个 `try-catch` 语句的 `try-block` 里面，并且其中一个 `catch-block` 的参数必须能匹配这个抛出的异常。看下面的程序。

```
1 void errorInside() {
2     cout << "errorInside() line A" << endl;
3     throw runtime_error("ERROR");
4     cout << "errorInside() line B" << endl;
5 }
6
7 int main() {
8
9     try {
10         cout << "main() line A" << endl;
11         errorInside();
12         cout << "main() line B" << endl;
13     }
14     catch (runtime_error & ex) {
15         cout << "caught : " << ex.what();
16     }
```

```
17     }  
18  
19 }  
输出  
main() line A  
errorInside() line A  
caught : ERROR
```

在上面的程序中，我们把可能抛出异常的函数 **errorInside** 放到了一个 try-block 中，并在同一个 try-catch 语句的一个 catch-block 中捕捉所抛出的异常 **runtime_error**。在这个程序中，如果函数 **errorInside** 不是在 try-block 中，而是在第 8 行、第 16 行或第 18 行，那么 **errorInside** 抛出的异常将不能被捕捉。

当一个异常被捕捉后，在捕捉这个异常的 catch-block 中可以访问所抛出的异常对象，并使用异常对象提供的错误信息。如在 35 行的 catch-block 中的参数 **ex** 就是在第 24 行所抛出的对象。

4. 异常处理机制的分析

为了更好地理解和使用 C++ 的异常处理机制，我们来讨论 C++ 的异常处理机制为什么设计成现在这种方式。首先，C++ 把异常处理机制分为抛出异常和处理捕获处理异常两个部分，当异常被抛出的时候程序的控制权可以立即跳到处理异常的模块。这样做的好处包括以下 3 个：

1. 可以把处理正常运行情况下的程序逻辑与处理各种错误的逻辑分开。只需要在正常运行的程序中加入检测错误和抛出异常对象的语句即可，能避免影响可读性。
2. 允许错误发生的程序模块与处理错误的程序模块的分离。让这两部分的程序可以写在不同的函数里面，甚至不同的类里面，增加程序的灵活性。
3. 让错误信息的传递更加简单。试想一个函数如果要为它间接调用的 100 个函数传递 `errorCode`，那么写程序将变得无比的繁琐。

下面我们讨论把异常的抛出（即报告一个异常）和异常的处理分开的必要性。好的程序设计风格要求设计程序时把程序的逻辑细分：每个函数只实现简单的逻辑。因此，错误产生的地方和真正引起错误的地方一般情况下是不同的：真正引起错误的函数一般情况下是错误产生的函数的调用者，而且通常不是直接调用者。例如，一个文件打开错误可能是由于用户在对话框里输入错误的文件名造成的。但是，通常地出现错误的地方没有足够的信息处理错误，更应该在引起错误的地方去处理错误，因为在这里有更多关于错误原因的信息。例如用户界面程序知道打开程序失败后，可以再次弹出对话框让用户输入正确的文件名。而打开文件出错的程序往往不可能作出通用的且正确的处理，因为它能被不同的程序调用。如被一个网络的程序调用的时候，引起文件名错误的

原因可能是网络传输的错误，这个时候就应该通知网络重传。这是完全不同的处理方式。

异常处理正是能提供这种可使错误的报告（抛出异常）和错误的处理（捕捉和处理异常）分离的编程结构。

最后，异常捕捉是根据异常对象的类型来被捕捉的，因此不同的异常应该有不同的类型。当然有些逻辑错误性质的错误我们可能不打算去捕捉或者无需区别，这个时候我们可以统一使用一种类型，如 `runtime_error`。异常类型可以不只是一个空壳类型，它可以包含一些用来记录错误信息的成员变量，以使得错误处理的程序模块能更好地分析和处理错误。

5. 异常处理实例

我们将通过一个实例讲解异常处理的一般用法，这个例子中也包括 C++ 中异常处理的一些新的语法。这个例子围绕一个容器类的实现。初始化容器类对象的时候可能因为用户给定一个不正常的容器大小而抛出异常，当访问容器中的元素是也可能因为用户给出的索引错误而抛出多种异常。然后我们在 `main` 函数中示范如何捕捉和处理不同的异常。

在这个例子中，我们还将介绍一个很好用的工具：**`stringstream`**。它和控制台输入输出、文件输入输出流(`fstream`)都是 `istream` 和 `ostream` 的子类。也就是说，它可以使用输入

输出操作符(<<和>>)。这里，我们只用到 `stringstream` 的输出操作符，用来构造一个表示错误的字符串。`stringstream` 定义在头文件中 `sstream`。

```
1 #include <iostream>
2 #include <sstream> // string stream
3 #include <string>
4 #include <stdexcept> // standard exception
5 using namespace std;
6
7 class NegativeIndexException
8 {
9 };
10
11 class IndexOutOfBoundsException : public runtime_error
12 {
13 private:
14     int size;
15     int index;
16
17 public:
18     IndexOutOfBoundsException(int size, int index)
19     : runtime_error("Index out of bound")
20     {
21         this->size = size;
22         this->index = index;
23     }
24
25     string what() {
26         stringstream ss;
27         ss << runtime_error::what();
28         ss << ", index = " << index;
29         ss << ", size = " << size;
30         return ss.str();
31     }
32 };
33
34 class Array
35 {
36 private:
37     int size;
```

```

38     double * elements;
39
40 public:
41     Array(int size) {
42         if (size > 10000) {
43             throw runtime_error("size too large.");
44         }
45         this->size = size;
46         elements = new double[size];
47         for (int i = 0; i < size; ++ i) {
48             elements[i] = 0;
49         }
50     }
51
52     double & operator [] (int index) {
53         if (index < 0) {
54             throw NegativeIndexException();
55         }
56         if (index >= size) {
57             throw IndexOutOfRangeException(size, index);
58         }
59         return elements[index];
60     }
61
62     ~Array() {
63         delete [] elements;
64     }
65 };
66
67 int main() {
68     try {
69         int size;
70         cin >> size;
71         Array array(size);
72         int index;
73         cin >> index;
74         double value = array[index];
75         cout << "value = " << value << endl;
76     }
77     catch (NegativeIndexException & ex) {
78         cout << "NegativeIndexException" << endl;
79     }
80     catch (IndexOutOfRangeException & ex) {
81         cout << "IndexOutOfRangeException" << endl;

```

```
82     }  
83     catch (runtime_error & ex) {  
84         cout << ex.what() << endl;  
85     }  
86 }
```

从第 7 行和第 9 行开始的两个类是异常类。其中第二个异常类继承了 `runtime_error` 并具有两个成员变量。在 25 行中 `override` 了父类中的 `what` 函数。在 26 行中定义了一个 `stringstream` 的对象 `ss`。在 26 行中调用了父类中的 `what` 函数。调用父类中被 `override` 了的函数的方法是：给出函数的全名。父类中的 `what` 函数返回的字符串的内容就是在 19 行初始化父类对象时所给出的字符数组的内容。在 30 行中 `stringstream` 的对象函数 `str` 返回一个字符串，它的内容是当前已被写入 `stringstream` 的对象的字符。

在容器类 `Array` 中我们分别在 2 个函数中抛出了 3 个不同类型的异常。当把容器的初始化大小定义得过大得时候，第 1 个异常会被抛出；当访问元素是给定错误索引的时候，第 2 和第 3 个异常中的其中一个会被抛出。注意，C++ 中任意一个时刻只能有最多一个异常，不能有两个或以上的异常同时被抛出。

在 77-85 行中，分别有 3 个 `catch-block` 捕捉和处理以上的 3 个异常。如果在 68-76 行中没有异常抛出，那么这 3 个 `catch-block` 都不会运行。如果有一个异常抛出，那么这个异

常将会按从上到下的顺序逐个地被匹配这 3 个 `catch-block`，首先匹配的 `catch-block` 将会执行。

这里要注意的是，在一个 `try-catch` 语句中，只会有最多一个 `catch-block` 会执行。也就是说，如果有多个 `catch-block` 能匹配一个异常对象，只有最上面的一个会被执行，其余的会被忽略。由于这个原因，能匹配一个父类异常对象的 `catch-block` 必须放在匹配一个子类异常对象的 `catch-block` 的后面，否则匹配子类异常对象的 `catch-block` 将不能匹配任何异常。例如，如果我们把上例中后两个 `catch-block` 交换，那么如果一个 `IndexOutOfBoundsException` 的异常被抛出，那么它将首先被参数为 `runtime_error` 的 `catch-block` 捕获，而参数为 `IndexOutOfBoundsException` 的 `catch-block` 将没有机会匹配任何它能匹配的异常。

看完这个例子，我们可以总结以下的编程建议：

1. 应该使用专门定义的类型来作为异常对象的类型；为了避免误解，不应该使用基础数据类型或者普通数据类型，如 `string` 或 `vector`，来作为异常对象的类型。
2. 为了避免子类的异常对象被匹配父类的 `catch-block` 捕获，避免在同一个 `try-catch` 语句中混合使用父类和子类的异常类型。

6. 异常抛出过程中的对象释构

我们现在看看在异常被抛出的过程中对象的释构。实际上，当程序因为抛出的异常而结束的时候，对象的释构与程序使用 `return` 结束是一样的：已经构造了的作为局部变量的对象会被释构。而异常对象本身在捕捉到这个异常的 `catch-block` 运行结束的时候才被释放。如果对象是动态分配的，那么它必须通过 `delete` 来释放；如果异常终止了对应的释放语句，那么可能造成内存泄漏。看下面的程序。

```
1 #include <iostream>
2 #include <string>
3 using namespace std;
4
5 class Object
6 {
7 private:
8     int id;
9
10 public:
11     Object(int id) {
12         this->id = id;
13         cout << "Object() : " << id << endl;
14     }
15
16     ~Object() {
17         cout << "~Object() : " << id << endl;
18     }
19
20 };
21
22 class Exception
23 {
24 private:
25     int id;
26
```

```

27 public:
28     Exception() {
29         id = 1;
30         cout << "Exception() : " << id << endl;
31     }
32
33     Exception(const Exception & ex) {
34         id = ex.id + 1;
35         cout << "Exception(Exception &) : " << id << endl;
36     }
37
38     ~Exception() {
39         cout << "~Exception() : " << id << endl;
40     }
41
42 };
43
44 int main() {
45     Object object1(1);
46     throw Exception(); // requires Exception(Exception &)
47     Object object2(2);
48 }

```

```

Object() : 1
Exception() : 1
~Object() : 1
libc++abi.dylib: terminate called throwing an exception

```

从上面的例子我们可以看到被构造了的 `object1` 会在异常抛出后被释放。如果我们在 33 行没有 `const`，那么在 46 行中我们会看到一个编译错误：46 行的匿名异常对象的类型是常量，不能匹配拷贝构造函数的参数。这里意味着程序会用 `throw` 后面的对象来构造另外一个异常对象：实际上被抛出的异常对象。但是我们在运行的时候并看不到拷贝构造函数的执行。这是应为编译器优化的介入，把多余的对象和它的构造去掉了。这类优化属于返回值优化 RVO(return value optimization)。

```

50 int main() {
51     try {
52         Object object1(1);
53         throw Exception(); // requires Exception(Exception &)
54         Object object2(2);
55     }
56     catch (Exception & ex) {
57         Object object3(3);
58     }
59     Object object4(4);
60 }

```

```

Object() : 1
Exception() : 1
~Object() : 1
Object() : 3
~Object() : 3
~Exception() : 1
Object() : 4
~Object() : 4

```

从上面的程序中，我们看到异常对象在被捕捉和处理完后才被释放。可以看到异常对象的释构是在 **object3** 的释构之后和 **object4** 的构造之前的。

另外，我们要避免在一个类的构造函数和析构函数中抛出异常。在构造函数中有异常抛出的对象的析构函数不会被调用。在析构函数中抛出异常容易造成内存泄漏，而且容易产生同时有两个被抛出异常的程序状态错误。

习题

1. 用自己的话解释下面的概念

异常 exception

抛出异常处理 `throwing an exception`

接住和处理异常 `catch and handle an exception`

`throw` 语句 `throw statement`

2. 用自己的话回答下面的问题

在 C++ 以及大多数当代的程序设计语言中，异常处理的形式是怎样的？

打开并读取一个文件的过程可能出现那些异常？

在一个容器类中可能抛出那些异常？

如何捕捉一个异常？

C++ 把异常处理机制分为抛出异常和处理捕获处理异常两个部分，它好处是什么？

异常处理提供一种能使错误的报告（抛出异常）和错误的处理（捕捉和处理异常）分离的编程结构，那么这种分离的必要性是什么呢？

一个具有多个 `catch-block` 的 `try-catch` 语句按什么顺序匹配一个异常？

能匹配一个父类异常对象的 `catch-block` 为什么必须放在匹配一个子类异常对象的 `catch-block` 的后面？

异常抛出过程中的局部对象如何释构？

异常对象何时何地地被释构？