第九章 异常处理

9.1 异常处理概述

9.1.1 什么是异常

程序的错误通常包括：

- 语法错误：指程序的书写不符合语言的语法规则。这类错误可由编译程序发现。
- 逻辑错误（或语义错误）：指程序设计不当造成程序没有完成预期的功能。这类错误可通过对程序进行静态分析和动态测试发现。
- 运行异常：指程序设计对程序运行环境考虑不周而造成的程序运行错误。

在程序运行环境正常的情况下，运行异常的错误是不会出现的。
导致程序运行异常的情况是可以预料的，但它是无法避免的。
为了保证程序的鲁棒性（Robustness），必须在程序中对可能出现的异常错误进行预见性处理。

9.1.2 异常处理的基本手段

异常处理包括：发现异常、处理异常

发现异常后如何处理异常？

- **就地处理**：在发现异常的地方处理异常。
- **异地处理**：在其它地方（非异常发现地）处理异常。

1. **异常的就地处理**

   就地处理异常的常见做法是调用C++标准库中的函数exit或abort终止程序执行（在cstdlib或stdlib.h中声明）

   - abort立即终止程序的执行，不作任何的善后处理工作。
   - exit在终止程序的运行前，会做关闭被程序打开的文件、调用全局对象和static存储类的局部对象的析构函数（注意：不要在这些对象类的析构函数中调用exit）等工作

2. **异常的异地处理**

发现异常时，往往在异常的发现地（如在被调用的函数中）不知道如何处理这个异常，或者不能很好地处理这个异常。这时，可由程序的其它地方（如函数的调用者）来处理。

一种解决途径：通过函数的返回值，或指针/引用类型的参数，或全局变量把异常情况通知函数的调用者，由调用者处理异常。

该途径的不足：

- 通过函数的返回值返回异常情况会导致正常返回值和异常返回值交织在一起，有时无法区分。
- 通过指针/引用类型的参数返回异常情况，需要引入额外的参数，给函数的使用带来负担。
- 通过全局变量返回异常情况会导致使用者会忽视这个全局变量的问题。（不知道它的存在）
- 程序的可读性差！程序的正常处理代码与异常处理代码混杂在一起。

另一种解决异常的异地处理途径：通过语言提供的结构化异常处理机制进行处理。

9.2 C++的异常处理机制

把有可能遭遇异常的一系列操作（语句或函数调用）构成一个try语句块。
如果try语句块中的某个操作在执行中发现了异常，则通过执行一个throw语句抛掷（产生）一个异常对象，之后的操作不再进行。
抛掷的异常对象将由程序中能够处理这个异常的地方通过catch语句块来捕获并处理之。

1. **try语句**

try语句块的作用是启动异常处理机制。格式为：try { <语句序列> }

2. **throw语句**

throw语句用于在发现异常情况时抛掷（产生）异常对象。格式为：throw <表达式>;
<表达式>为任意类型的C++表达式（void除外）。

执行throw语句后，接在其后的语句将不再继续执行，而是转向异常处理（由某个catch语句给出）。

3. **catch语句**

catch语句块用于捕获throw抛掷的异常对象并处理相应的异常。格式为：catch (<类型> [<变量>]) { <语句序列> }

- <类型>用于指出捕获何种异常对象，它与throw所产生的异常对象的类型匹配规则与函数重载的绑定规则类似。
- <变量>用于存储捕获到的异常对象。它可以缺省，表明catch语句块只关心异常对象的类型，而不考虑具体的异常对象。

catch语句块要紧接在某个try语句的后面。

一个try语句块的后面可以跟多个catch语句块，用于捕获不同类型的异常对象并进行处理

注意：

- 如果在try语句块的<语句序列>执行中没有抛掷（throw）异常对象，则其后的catch语句不执行，而是继续执行try语句块之后的非catch语句。
- 如果在try语句块的<语句序列>执行中抛掷了（throw）异常对象，
    - 如果该try语句块之后有能够捕获该异常对象的catch语句，则执行这个catch语句中的<语句序列>，然后继续执行这个catch语句之后的非catch语句。
    - 如果该try语句块之后没有能够捕获该异常对象的catch语句，则按嵌套的异常处理规则进行处理。

9.2.2 异常的嵌套处理

当在内层的try语句的执行中产生了异常，则首先在内层try语句块之后的catch语句序列中查找与之匹配的处理，如果内层不存在能捕获相应异常的catch，则逐步向外层进行查找。
如果抛掷的异常对象在程序的函数调用链上没有给出捕获，则调用系统的terminate函数进行标准的异常处理。默认情况下，terminate函数将会去调用abort函数。

9.3 基于断言的程序调试

实际上，在调试程序时输出程序在一些地方的某些变量或表达式的值，其目的是为了确认程序运行到这些地方时状态是否正确。

上述目的可以在程序的一些关键或容易出错的点上插入一些断言来表达。

断言（assertion）是一个逻辑表达式，它描述了程序执行到断言处应满足的条件。如果条件满足则程序继续执行下去，否则程序异常终止。

断言的作用：

- 帮助对程序进行理解和形式化验证。
- 在程序开发阶段，帮助开发者发现程序的错误和进行错误定位。

宏assert

C++标准库中提供了一个宏assert（在头文件cassert或assert.h中定义），可以用来实现断言。其格式为：
assert(<表达式>);
<表达式>一般为一个关系/逻辑表达式

assert执行时，

- 如果<表达式>的值为true，程序继续正常执行。
- 如果<表达式>的值为false，则它会:
  首先，显示出相应的表达式、该assert所在的源文件名以及所在的行号等诊断信息；
  然后，调用库函数abort终止程序的运行。

宏assert的实现

宏assert是通过条件编译预处理命令来实现的，其实现细节大致如下：

```
//cassert 或 assert.h
……
#ifdef  NDEBUG
#define assert(exp)  ((void)0)
#else
#define assert(exp)  ((exp)?(void)0:<输出诊断信息并调用库函数abort>)
#endif
……
```

宏assert只有在宏名NDEBUG没定义时才有效，这时，程序一般处于开发、测试阶段。程序开发结束提交时，应该让宏名NDEBUG有定义，然后重新编译程序，这样，assert就不再有效了。

宏名NDEBUG在哪儿定义？

- 在编译命令中指出。例如：
  cl <源文件1> <源文件2> … -D NDEBUG …
- 在开发环境中的“项目|…属性|C/C++|预处理器|预处理器定义”中指出。（VC++2015）