异常处理

Andrew Huang<bluedrum@163.com>

课程内容

- C 的错误处理的机制
- C++ 的异常处理机制
- 其它异常处理

软件的错误处理

- 一个可用的软件必然是经过测试的. 其中大量的测试用于错误情况的处理.
- 一个软件的可能要花50% 的代码去处理运行异常情况, 甚至更高比例.
- 因此作为开发者, 在哪怕是设计一个基本的函数, 也要考虑的到各种运行错误可能性. 并在代码相应的地方作处理.
  - 例, 普通文件打开异常的可能性
  - 例,IP 地址转换输入函数
- 因此, 熟悉各种错误处理机制是一个开发者必修的功课

在VC++ 的 异常处理

- Visual C++ 提供了对C 语言、C++ 语言及MFC 的支持，因而其涉及到的异常（exception ）处理也包含了这三种类型，即C 语言、C++ 语言和MFC 的异常处理 .
- 除此之外，微软对C 和C++ 的异常处理进行了扩展，提出了结构化异常处理（SEH ）的概念，它支持C 和C++ ,
  - SHE 只能用于Windows 下的编程
  - MFC 异常处理仅支持C++
- 本课程主要总结C 的错误处理, 重点C++ 的异常处理机制, 也会简单介绍一下 MFC/SEH 错误处理

异常处理步骤

- 无论采用哪种处理机制, 异常处理总是包含如下几个步骤
- 程序执行时发生错误;
- 以一个异常对象（最简单的是一个整数）记录错误的原因及相关信息;
- 程序检测到这个错误（读取异常对象）;
- 程序决定如何处理错误;
- 进行错误处理，并在此后恢复/ 终止程序的执行。
- C 、C++ 、MFC 及SEH 在这几个步骤中表现出了不同的特点

C 的错误处理机制

C 语言错误处理机制

- C 语言没有语法级的标准错误处理机制, 只是通过C 标准库提供了几个方法来处理异常. 包括如下处理
  - 异常中止
  - 断言(assert)
  - 全局的错误变量errno

- 非局部跳转(setjmp/longjmp)
- 函数**返回值和回传参数**
- 信号(signal，只适于POSIX 风格操作系统)

- 上述机制并不是健壮的，总有一些情况处理不了，或有这样那样缺点

异常终止

- 标准C 库提供了abort() 和exit() 两个函数，它们可以强行终止程序的运行，其声明处于<stdlib.h> 头文件中。
- 这两个函数本身不能检测异常，但在C 程序发生异常后经常使用这两个函数进行程序终止。
- C 库头文件<stdlib.h> 提供了两个终止程序的函数：abort() 和exit() 。这两个函数运行于异常生命期的4 和5 。它们都不会返回到其调用者中，并都导致程序结束。这样，它们就是结束异常处理的最后一步。

异常终止(2)

- 虽然两个函数在概念上是相联系的，但它们的效果不同：
  - abort() ：程序异常结束。默认情况下，调用abort() 导致运行期诊断和程序自毁。它可能会也可能不会刷新缓冲区、关闭被打开的文件及删除临时文件，这依赖于你的编译器的具体实现。
  - exit() ：文明地结束程序。除了关闭文件和给运行环境返回一个状态码外，exit() 还调用了你挂接的atexit() 处理程序。
- 对于exit 函数，我们可以利用atexit 函数为exit 事件" 挂接" 另外的函数，这种" 挂接" 有点类似Windows 编程中的" 钩子" （Hook ）。譬如：

aexit 例子

- 注意，即便是没有exit(),aexit 仍然会执行.

```
#include <stdio.h>
#include <stdlib.h>

static void atExitFunc(void)
{
        printf("atexit 挂接的函数\n");
}
//EXIT_SUCCESS、EXIT_FAILURE 分别定义为 0 和 1。
int main(void)
{
        atexit(atExitFunc);
        exit(EXIT_SUCCESS);
        printf("程序不会执行到这里\n");
        return 0;
}
```

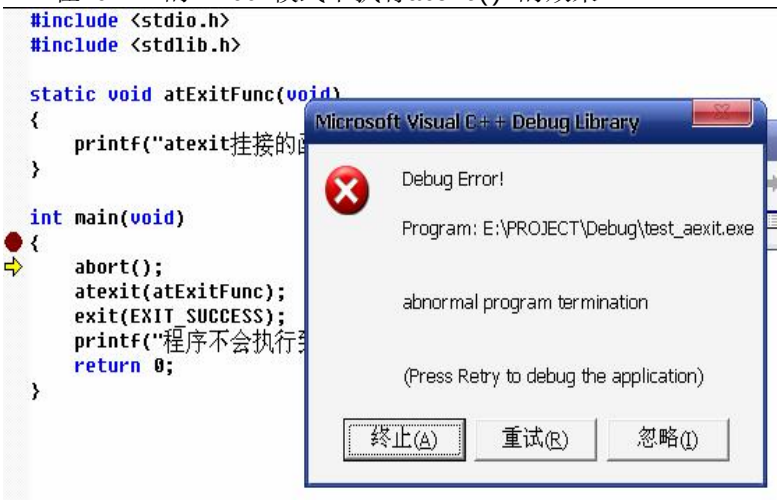
aexit 例子(2)

- atexit可以被多次执行，并挂接多个函数，这些函数的执行顺序为后挂接的先执行

```
#include <stdio.h>
#include <stdlib.h>

static void atExitFunc1(void)
{    printf("atexit 挂接的函数 1\n");}

static void atExitFunc2(void)
{    printf("atexit 挂接的函数 2\n");}

static void atExitFunc3(void)
{    printf("atexit 挂接的函数 3\n");}

int main(void)
{
   atexit(atExitFunc1);
   atexit(atExitFunc2);
   atexit(atExitFunc3);
   return 0;
}
```

abort()

- 在VC++ 的DEBUG 模式下执行abort() 的效果



断言(assert)

- assert 宏在C 语言程序的调试中发挥着重要的作用，它用于检测不会发生的情况，表明一旦发生了这样的情况，程序就实际上执行错误了，例如strcpy 函数：

```
char *strcpy(char *strDest, const char *strSrc)
{
   char *address = strDest;
   assert((strDest != NULL) && (strSrc != NULL));
   while ((*strDest++ = *strSrc++) != ’\0’)
      ;
   return address;
}
```

assert 宏的定义

- 如果程序不在debug模式下，assert宏实际上什么都不做；而在debug模式下，实际上是对_assert()函数的调用，此函数将输出发生错误的文件名、代码行、条件表达式
- 当然这样assert对release模式下无能为力,但是在运行时,仍然有可然发生空指针之类的异常情况,无法保险

```
#ifdef NDEBUG
#define assert(exp) ((void)0)
#else
#ifdef __cplusplus
extern "C"
{
   #endif

   _CRTIMP void __cdecl _assert(void *, void *, unsigned);
   #ifdef __cplusplus
}
#endif
#define assert(exp) (void)( (exp) || (_assert(#exp, __FILE__, __LINE__),
0) )
#endif /* NDEBUG */
```

assert 宏的实例

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
char * myStrcpy( char *strDest, const char *strSrc )
{
   char *address = strDest;
   assert( (strDest != NULL) && (strSrc != NULL) );
   while( (*strDest++ = *strSrc++) != ’\0’ );
      return address;
}
int main(void)
{
   myStrcpy(NULL,NULL);
   return 0;
}
```

全局错误变量errno

- errno在C程序中是一个全局变量，这个变量由C运行时库函数设置，用户程序需要在程序发生异常时检测之。
- C运行库中主要在math.h和stdio.h头文件声明的函数中使用了errno，前者用于检测数学运算的合法性，后者用于检测I/O操作中（主要是文件）的错误

```
#include <errno.h>
#include <math.h>
#include <stdio.h>
int main(void)
{
   errno = 0;
   if (NULL == fopen("d:\\1.txt", "rb"))
   {       printf("%d", errno);      }
   else
   {         printf("%d", errno);      }
   return 0;
}
```

全局错误变量errno

- 在此程序中，如果文件打开失败（fopen 返回NULL ），证明发生了异常。我们读取error可以获知错误的原因，如果D 盘根目录下不存在¡°1.txt¡± 文件，将输出2 ，表示文件不存在；在文件存在并正确打开的情况下，将执行到else 语句，输出0 ，证明errno没有被设置。
- Visual C++ 提供了两种版本的C 运行时库。- 个版本供单线程应用程序调用，另一个版本供多线程应用程序调用。多线程运行时库与单线程运行时库的一个重大差别就是对于类似errno 的全局变量，每个线程单独设置了一个。因此，对于多线程的程序，我们应该使用多线程C 运行时库，才能获得正确的errno 值。
- 注意errno是全局共享的,因此在发生错误要立即取值,否则很可能被下一函数产生错误重置.
- 如果是Posix 的风格的C库(Linux,Unix),还提供strerror (errno)，把错误号转换成一个错误提示字符串

非局部的跳转

- goto 是本地的，它只能在一个函数内部的标号上跳转，而不能将控制权转移到所在程序的任意地点 .
- 为了解决这个限制，C 函数库提供了setjmp() 和longjmp() 函数，它们分别承担非局部标号和goto 作用。头文件<setjmp.h> 申明了这些函数及同时所需的jmp_buf 数据类型。
- longjmp(j,r) 产生异常对象r （它为一个整数） ，
- Longjmp 执行时，会跳转到setjmp(j) 处。而且setjmp() 函数返回值就是异常r

setjmp/longjmp 实例

```
#include <setjmp.h>
#include <stdio.h>

jmp_buf j; //定义一个跳转上下文
void raise_exception(void)
    {
    printf("exception raised\n");
    longjmp(j, 1); /*  长跳转,跳到 j 所指向的上下文,即错误处理函数  */
  /* longjmp  的值设为 1*/
    printf("this line should never appear\n");
    }

int main(void)
    {
    if (setjmp(j) == 0) //设置上下文
        {
        printf("""setjmp"" is initializing ""j""\n");
        raise_exception();
        printf("this line should never appear\n");
        }
    else
        { /*  因为 longjmp  设为 1,所以将执行这里代码*/
        printf("""setjmp"" was just jumped into\n");
        /* this code is the exception handler */
        }
    return 0;
    }
```

返回值和回传参数

- errno 全局变量有很多限制
- 函数返回值是C 标准库所喜欢的通报异常方法
- 回传指针和C++ 的引用型的参数是函数返回值的变形

```
if ((p = malloc(n)) == NULL)
   /*  出错返回 NULL  指针*/

if ((c = getchar()) == EOF)
   /*  出错返回 EOF(-1) */

if ((ticks = clock()) < 0)
   /*  出错返回小于 0 值  */
```

C 各种处理机制的缺点

1.返回值	可以说这是最常用的错误处理方式之一，但其存在着一个致命的问题。就是返回值的检查与否是由调用者主动控制的。如果调用者不检查返回值，那也没有任何 机制能够强迫他这么做。再一个，考虑在 C++中参数表相同而返回值不同的重载情况。在这种情况下，如果调用者不检查返回值的话，编译器根本不清楚应该调用哪个函数。
2.全局状态标示符	这种办法同返回值一样，也是需要调用者主动检查的。并且由于其是全局的，因此在多线程程序中，还必须保证它的线程安全性，必须要让检查者知道这是谁的返回值。
3.setjmp()/longjmp()	你完全可以将 **longjmp()当成远程的 goto 语句进行调用(goto 语句只能左右于本地函数里)。但这个函数却存在着很大甚至是致命的危险。暂且放下该函数会破坏结构化程序设计风格不说。其一，longjmp()只能处理 int 型的异常。其二，也就是最致命的一点就是，longjmp()不会调用析构函数，而 C++的 异常处理机制却会完成这个事情。因此，在 C++中，千万不要使用 setjmp()、longjmp()函数。**
4.断言	对于断言(Assert)，其仅仅是在 **Debug 版本中起作用，在 Release 中其是不存在的。另外断言与我们通常所说的错误处理方式不同，他是用来处理我们可能会发生这个错误，并能够避免的这种情况。**

C++异常处理机制

C++ 的异常处理机制

- 标准C++语言中专门集成了异常处理的相关语法
  - 所有的C 标准库异常体系都需要运行库的支持,不是内置机制,而C++内置机制.
- C++为了支持异常处理,新增了try,catch,throw关键字
- 异常的抛出方式为使用throw(type e)

```
try
{ //可能引发异常的代码 }
catch(type_1 e)
{// type_1 类型异常处理 }
catch(type_2 e)
{ // type_2 类型异常处理  }
catch (...)//会捕获所有未被捕获的异常，必须最后出现
{
}
```

一个异常处理实例

```
#include <stdio.h>
//定义 Point 结构体（类）
typedef struct tagPoint
{
    int x;
    int y;
} Point;
//扔出 int 异常的函数
static void f(int n)
{
    throw 1;//假设这里出错,仍出一个错误,
//并且用整数 1 来标识
      }

//扔出 Point 异常的函数
static void f(Point point)
{
    Point p;
    p.x = 0;
    p.y = 0;
    throw p;//假设这里也出错,仍出一个错误,
//并且用 POINT p 来标识}
```

个异常处理实例(2)

```
int main()
{
    Point point;
    point.x = 0;
    point.y = 0;

    try
    {
        f(point); //抛出 Point 异常
        //f(1); //抛出 int 异常
    }
    catch (int e)//捕获类型为整数的异常
        { //对应了,捕获 throw 1 的异常
        printf("捕获到 int 异常： %d\n", e);
    }
    catch (Point e)
    {
        printf("捕获到Point异常:(%d,%d)\n", e.x, e.y);
    }

    return 0;
}
```

throw 的语法格式

- throw表示主动扔出一个异常,语法格式
  - throw [expression]
- 函数在定义时通过异常规格申明定义其会抛出什么类型的异常，其格式为：
  - throw([type-ID-list])
  - type-ID-list是一个可选项，其中包括了一个或多个类型的名字，它们之间以逗号分隔
  - 例:void func() throw(int, some_class_type)
    - 表示funcm内部出错的话可能会扔出两种类型异常, 分别是int和some_class_type类型异常 .
    - 这样用于提醒调用函数的开发者,要准备对应的catch语句进行捕获.
    - 这一机制主要是为了方便,当调用者看不到函数实现源码的情况(比如是一个库函数),可以通过声明知道内部将会扔出哪几种类型的异常
    - 如果type-ID-list 为空表示,函数不扔出任何异常
      - int func(int i) throw();
  - 具体实例 CString Mid(int *nFirst*) const;
    throw(CMemoryException);

try -- catch 语法格式

- try 块中的异常处理函数对异常进行捕获。其可以包含一个或多个处理函数，其形式如下：
  - catch (exception-declaration) compound-statement
  - 处理函数的异常申明指明了其要捕获什么类型的异常。
  - 对于异常申明其可以是无名的,例如:catch(char *) ，其表明会捕获一个char * 类型异常，但由于是无名的，因此不能对其进行操作
  - 异常申明也可以存在如下形式:catch(...) ，其表明会捕获任何类型的异常。
  - throw 后面没有接任何对象，这表明throw 会再次抛出已存在的异常对象，因此其必须位于catch 块中。
  -

throw 抛出异常的特点

- 可以抛出基本数据类型异常，如int 和char 等；
- 可以抛出复杂数据类型异常，如结构体（在C++ 中结构体也是类）和类；
- C++ 的异常处理必须由调用者主动检查。一旦抛出异常，而程序不捕获的话，那么abort() 函数就会被调用，弹出abortx 终止对话框，程序被终止；
- 可以在函数头后加throw([type-ID-list]) 给出异常规格，声明其能抛出什么类型的异常。type-ID-list 是一个可选项，其中包括了一个或多个类型的名字，它们之间以逗号分隔。如果函数没有异常规格指定，则可以抛出任意类型的异常。

try -- catch 实例

```
void func() throw(int, some_class_type)
{
    int i;
    ........
    throw i;//扔出一个整数异常

    ........
}
int main()
{
    try
    {
       func();
    }
    catch(int e)
    {
       //处理 int 型异常
            throw; //不处理,重新把异常扔到上一层 try
     }
    catch(some_class_type)
    {
       //处理 some_class_type 型异常
    }
    .......
    return 0;
}
```

异常类的基类exception

- 标准异常都派生自一个公共的基类exception。基类包含必要的多态性函数提供异常描述，可以被重载
- 下面是exception类的原型：

```
class exception
{
   public:
      exception() throw();
      exception(const exception& rhs) throw();
      exception& operator=(const exception& rhs) throw();
      virtual ~exception() throw();
      virtual const char *what() const throw();
};
```

标准异常

- C++ 为一些固定的错误提供的一些标准异常

```
namespace std
{
   //exception 派生
   class logic_error; //逻辑错误,在程序运行前可以检测出来

   //logic_error 派生
   class domain_error; //违反了前置条件
   class invalid_argument; //指出函数的一个无效参数
   class length_error; //指出有一个超过类型 size_t 的最大可表现值长度的对象的企图
   class out_of_range; //参数越界
   class bad_cast; //在运行时类型识别中有一个无效的 dynamic_cast 表达式
   class bad_typeid; //报告在表达试 typeid(*p)中有一个空指针 p

   //exception 派生
   class runtime_error; //运行时错误,仅在程序运行中检测到

   //runtime_error 派生
   class range_error; //违反后置条件
   class overflow_error; //报告一个算术溢出
   class bad_alloc; //存储分配错误
}
```

自定义异常类

- Exception其中的一个重要函数为what() ,它返回一个表示异常的字符串指针
- 开发者可以把某一类自定义错误定义为一个Exception的派生类

```
class myexception:public exception
{   public:
   myexception():exception("重载 exception 的例子")
   {}
};

int main()
{
   try
   {
      throw myexception();
   }
   catch (exception &r) //捕获异常,    {
//注意这里也可以捕获 exception 的派生类,r.what()是多态
      cout << "捕获到异常：" << r.what() << endl;
   }
   return 0;}
```

throw 被捕获顺序

- 首先被最内层的catch捕获,如果最内层catch里,即没有与throw 的类型相匹配的,又没有catch(¡-) 去通吃所有错误. 这个throw 将会被提交到上一层.
- 如果上层有对应类型的catch, 则被捕获, 否则会提交到更高一层.

- 如果到最顶层的main() 仍然没有对应的catch，则被系统转到缺省的异常处理函数．在VC++ 中是调用abort()．
- 对于已经在catch 捕获的错误，可以通过不带参数的throw 重新向外抛出错误．

C++ 异常处理的优点

- 把可能出现异常的代码和异常处理代码隔离开，结构更清晰．
- 把内层错误的处理直接转移到适当的外层来处理，化简了处理 流程．传统的手段是通过一层层返回错误码把错误处理转移到 上层，上层再转移到上上层，当层数过多时将需要非常多的判断， 以采取适当的策略
- 可以抛出比较复杂的异常类型，在出现异常时，能够获取异常的信息，指出异常原因．并可以给用户优雅的提示．

C++ 异常处理的优点(2)

- 局部出现异常时，在执行处理代码之前，会执行堆栈回退，即为 所有局部对象调用析构函数，保证局部对象行为良好 ，但不会主动释放new 生成对象或类型．
- 可以在出现异常时保证不产生内存泄漏．通过适当的try,catch 布局，可以保证 delete pobj; 一定被执行
- 可以在处理块中尝试错误恢复．保证程序几乎不会崩溃． 通过适当处理，即使出现除0 异常，内存访问违例，也能 让程序不崩溃，继续运行，这种能力在某些情况下及其重要

C++ 异常处理的注意

- 如果使用普通的处理方式:ASSERT,return 等已经 足够简洁明了，请不要使用异常处理机制
- 可以处理任意类型的异常． 你可以人为地抛出任何类型的对象作为异常．throw 100; throw "hello";
- 需要一定的开销，频繁执行的关键代码段避免使用 C++ 异常处理机制．

其它异常处理机制(选)

MFC 异常处理

- MFC 较好地将异常封装到CException 类及其派生类中，自成体系
- MFC 定义一组宏:TRY ,CATCH, AND_CATCH, 和END_CATCH,THROW 和 THROW_LAST
  - 非常类似try ,catch 和throw
- MFC 现在建议使用C++ 标准异常处理

SE(结构化异常) 处理

- 结构化异常处理（Structured Exception Handling ，简称SEH ）是微软针对Windows 程序异常处理进行的扩展
- 在Visual C++ 中，它同时支持C 和C++ 语言。SEH 不宜与标准C++ 异常处理和MFC 异常处理混用，
- 对于C++ 程序，微软建议使用标准C++ 的异常处理。
- 为了支持SEH ，Visual C++ 中定义了四个关键字（由于这些关键字是非标准关键字，其它编译器不一定支持），用以扩展C 和C++ 语言：
  - （1 ）__except
  - （2 ）__finally

- （3）__leave
- （4）__try

各种异常处理的对比

异常处理	支持语言	是否标准	复杂度	推荐使用
C 异常处理	C 语言	标准 C	简单	推荐
C++异常处理	C++语言	标准 C++	较简单	推荐
MFC 异常处理	C++语言	仅针对 MFC 程序	较简单	不推荐
SEH 异常处理	C 和 C++语言	仅针对 Microsoft 编译环境	较复杂	不推荐

课堂练习

- 把自动开发的string 类加入C++ 标准异常处理机制
  - 要求定义一个自定义异常派生类，例strException
  - 在所有出错地方，采用throw 扔出strException.
  - 在主程序的测试函数，用try catch 进行捕获并进行错误处理