C++程序设计精要教程

华中科技大学

第10章　异常与断言

◆10.1 异常处理

- 异常：一种意外破坏程序正常处理流程的事件、由硬件或者软件触发的事件。
- 异常处理可以将错误处理流程同正常业务处理流程分离，从而使程序的正常业务处理流程更加清晰顺畅。
- 异常发生后自动析构调用链中的所有对象，这也使程序降低了内存泄漏的风险。
- 由软件用 throw 语句抛出的异常，会在抛出点建立一个描述异常的对象，由 catch 捕获相应类型的异常。

第10章　异常与断言

◆10.1 异常处理

```
try {
        语句体;
        在某种条件1下抛出异常：  throw  异常对象1;
        在某种条件2下抛出异常：  throw  异常对象2;
}
catch (异常对象1的类型  变量1) { … }   //捕获类型1的异常
catch (异常对象2的类型  变量2) { … }   //捕获类型2的异常
… …
catch (…) { … }   //捕获所有的异常
```

第10章　异常与断言

◆10.1 异常处理

- **throw 异常对象 ⇔ throw(异常对象)。**
- **异常对象，可以是简单的数据类型，也可以是复杂的类对象。例如，throw “ab”，throw (1)，throw new int[2]{1, 2}，throw new A(1)，throw new A(1), throw A(1)。这里 A 是1个类名。**
- **任何throw后面的语句都会被忽略，直接进入异常捕获处理过程catch。**
- **catch (…) 表示捕获任何类型的异常。在 catch( ) { … } 的语句体中，可以用不带参数的 throw 继续抛出已经捕获的异常（相当于这个异常没有被处理）。**
- **一旦异常被某个 catch 处理过程捕获，其它所有的异常捕获过程都会被忽略。**
- **如果抛出的异常没有被任何try语句捕捉，那么这种没有被处理的异常将引起程序终止执行。**
- **C++不支持 finally 处理过程。**

第10章　异常与断言

◆10.2 捕获顺序

- 允许函数模板和模板实例函数定义异常接口。
- 如果父类A的子类为B，B类异常也能被 catch(A)、catch(const A)、catch(volatile A)、 catch(const volatile A) 等捕获。
- 如果父类A的子类为B，指向B类对象的指针异常也能被 catch(A*)、catch(const A*)、 catch(volatile A*)、 catch(const volatile A*) 等捕获。
- 捕获子类对象的catch应放在捕获父类对象的catch前面。
- catch(const volatile void *) 能捕获任意指针类型的异常，const A * 能捕获 A * 的异常，catch(…) 能捕获任意类型的异常。

第10章　异常与断言

【例10.3】定义VECTOR的INDEX和SHORTAGE异常处理过程

```
#include <stdio.h>
class VECTOR {
    int *data;
    int  size;
public:
    VECTOR(int n);
    int &getData(int i);      //取下标所在位置的向量元素
    ~VECTOR( ) { delete [ ] data; };
};
class INDEX {
    int index;
public:
    INDEX(int i) { index = i; }
    int getIndex( ) const { return index; }
};
```

第10章　异常与断言

```
struct SHORTAGE : INDEX {
    SHORTAGE(int i) : INDEX(i) { }
    using INDEX::getIndex;
};

VECTOR::VECTOR(int n)
{
    if ( !(data = new int[size = n]) ) throw SHORTAGE(0);
};

int &VECTOR::getData(int i)
{
    if ( i < 0 || i >= size ) throw INDEX(i);
    return data[i];
};
```

第10章　异常与断言

```
void main(void)
{
    VECTOR v(100);
    try { v.getData(100) = 30; }
    catch( SHORTAGE ) {
        printf("SHORTAGE: Shortage of memory!\n");
    }
    catch (const INDEX r) {
        printf("INDEX: Bad index is %d\n", r.getIndex( ));
    }
    catch (...) {
        printf("ANY: any error caught!\n");
    }
}
```

如果 catch(SHORTAGE) 和 catch(const INDEX r) 的顺序交换一下，结果会怎样?

第10章　异常与断言

◆10.3 函数的异常接口

- 通过异常接口声明的异常都是由该函数引发的，但该函数自身又不想捕获或处理的异常。

- 异常接口定义的异常出现在函数的参数表后面，用 throw 列出要引发的异常类型：

```
void  f(void) throw(A, B);
//f( )内部可能抛出类型A异常和类型B异常
void  f(void) const throw(A, B);  (成员函数)
void  f(void);                    //可引发任何异常
void  f(void) throw( );           //不引发任何异常
void  f(void) throw(void);        //不引发任何异常
void  f(void) noexcept;           //不引发任何异常
```

throw()、throw(void)、noexcept 是等价的，告诉编译器函数不会抛出任何异常。noexcept 是C++11的新特性，功能更强，有取代throw()的趋势。

第10章　异常与断言

◆10.3 函数的异常接口

- 异常接口不是函数原型的一部分，不能通过异常接口来定义和区分重载函数，不影响函数内联、重载、缺省和省略参数。
- 不引发任何异常的函数引发的异常、引发了未说明的异常称为不可意料的异常。
- 通过 set_unexpected 过程，可以将不可意料的异常处理过程设置为程序自定义的不可意料的异常处理过程，其设置方法和通过 set_terminate 过程设置终止处理函数类似，设置后也返回一个指原先的不可意料的异常处理过程的指针。不同的编译提供的设置函数可能不同。
- 函数模板、类模板的函数成员都可以定义异常接口 (构造函数和析构函数都可以定义异常接口)。

第10章　异常与断言

【例10.4】对数组a的若干相邻元素进行累加

```
#include <iostream>
using namespace std;
//sum()计算从a[s]开始的c个数组元数之和，抛出 const char * 类型的
异常，但没有捕获和处理。
int sum(int a[ ], int t, int s, int c) throw (const char *)   //a[0...t-1]
{  //以下语句若发出 const char * 类型的异常，此后的语句不执行
   if (s < 0 || s >= t || s + c < 0 || s + c > t)
      throw “subscription overflow”;  //抛出 const char * 异常
   int r = 0, x = 0;
   for( x = 0; x < c; x++ )
      r += a[s + x];
   return r;
}
```

第10章　异常与断言

```
void main( )
{
    int m[6] = {1, 2, 3, 4, 5, 6};
    int r = 0;
    try {
        r = sum(m, 6, 3, 4); //发出异常后try中所有语句都不执行，直接到其catch
        r = sum(m, 6, 1, 3); //不发出异常
    }
    //char *p不能捕获const char *类型的异常
    catch(char *p) { cout << p; } //不能捕获throw "subscription overflow";
    catch(const char *e) { //还能捕获char *类型的异常, 可写指针实参可以传递给只读指针形参e
        cout << e;
    } //由于throw时未分配内存，故在catch中无须使用delete  e
}
```

第10章　异常与断言

◆10.3 函数的异常接口

- **noexcept 可以表示 throw() 或 throw(void)。**
- **noexcept 一般用在移动构造函数，析构函数、移动赋值运算符函数等肯定不会出现异常的函数后面。**
- **如果移动构造函数和移动赋值运算符还要申请内存之外的资源，则难免发生异常，此时不应将 noexcept 放在这些函数的参数标后面。**
- **保留字 noexcept 和 throw( ) 可以出现在任何函数的后面，包括constexpr函数和 Lambda表达式的参数表后面。但 throw (除void外的类型参数)不能出现在 constexpr 函数的参数表后面，并且 constexpr 函数也不能抛出异常，否则不能优化生成常量表达式。**

第10章　异常与断言

```
class STACK  {
    int *const  elems;                  //申请内存用于存放栈的元素
    const int   max;                    //栈能存放的最大元素个数
    int   pos;                          //栈实际已有元素个数，栈空时pos=0;
public:
    STACK(int m);                       //初始化栈：最多存放m个元素
    STACK(const STACK &s);              //用栈s深拷贝构造新栈
    STACK(STACK &&s) noexcept;  //用栈s浅拷贝构造新栈, 移动构造
    virtual ~STACK( ) noexcept;         //销毁栈
};
STACK::STACK(STACK &&s) noexcept : elems(s.elems), max(s.max), pos(s.pos)
{   //移动构造
    *(int **)&s.elems = nullptr;        //等价于(int *&)s.elems = nullptr;
    *(int *)&s.max = s.pos = 0;         //等价于(int &) s.max = s.pos = 0;
}
```

第10章　异常与断言

◆10.4 异常类

- C++提供了一个标准的异常类 exception，以作为标准类库引发的异常类型的基类，exception 等异常由标准名字空间std提供。
- exception 的函数成员不再引发任何异常。
- 函数成员 what( ) 返回一个只读字符串，该字符串的值没有被构造函数初始化，必须在派生类中重新定义函数成员 what( )。
- 异常类 exception 提供了处理异常的标准框架，应用程序自定义的异常对象应当自 exception 继承。
- 在 catch 有父子关系的多个异常对象时，应注意catch顺序。

第10章　异常与断言

◆10.4 异常对象的析构

- 如果是通过new产生的指针类型的异常，在 catch 处理过程捕获后，通常应使用合适的delete释放内存，否则可能造成内存泄漏。
- 如果继续传播指针类型的异常，则可以不使用 delete。
- 从最内层被调函数抛出异常到外层调用函数的catch处理过程捕获异常，由此形成的函数调用链所有局部对象都会被自动析构，因此使用异常处理机制能在一定程度上防止内存泄漏。但是，调用链中的指针通过 new 分配的内存不会自动释放。
- 特殊情况在产生异常对象的过程中也会出现内存泄漏情况：未完成构造的对象。

第10章　异常与断言

【例10.7】局部对象的析构过程

```
#include <exception>
#include <iostream>
using namespace std;
class EPISTLE : exception { //定义异常对象的类型
public:
    EPISTLE(const char *s) : exception(s) { cout<<"Construct: " << s; }
    ~EPISTLE( ) noexcept { cout << "Destruct: " << exception::what( ); };
    const char *what( ) const throw( ) { return exception::what( ); };
};
void h( ) {
    EPISTLE  h("I am in h( )\n");
    throw new EPISTLE("I have throw an exception\n");
}
```

第10章　异常与断言

```
void g( ) {  EPISTLE  g("I am in g( )\n");  h( );  }
void f( ) {  EPISTLE   f("I am in f( )\n");   g( );  }
void main(void) {
   try {   f( );   }
   catch (const EPISTLE *m) {
      cout  <<  m->what( );
   }
}
```

main() -> f(), f -> g(), g -> h(), h -> 抛出异常(指针) ->
局部对象h、g、f依次析构 -> main捕获异常并 delete

第10章　异常与断言

◆10.6 断言

- 断言函数 assert (bool expr) 在 assert.h 中定义。
- assert ( )是一个带有整型参数的用于调试程序的函数，如果表达式的值为真则程序继续执行。
- 否则，将输出断言表达式、断言所在代码文件名称以及断言所在程序的行号，然后调用 abort() 终止程序的执行。
- 断言输出的代码文件名称包含路径（编译时值），运行时程序拷到其它目录也还是按原有路径输出代码文件名称。assert() 在运行时检查断言。
- 保留字 static_assert (bool expr) 在编译时检查，为真时不终止编译运行。

第10章　异常与断言

【例10.9】断言的用法

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
class  SET {
    int *elem, used, card;
public:
    SET(int card);
    virtual int has(int)const;
    virtual SET &push (int);
    virtual ~SET( ) noexcept { if (elem) { delete elem;  elem = 0; } };
};
SET::SET(int c) {
    card = c;
    elem = new int[c];
    assert(elem);   //当elem非空时继续执行
    used = 0;
}
```

第10章　异常与断言

```
int SET::has(int v) const {
    for (int k = 0; k < used; k++)  if (elem[k] == v) return 1;
    return 0;
}
SET &SET::push(int v) {
    assert( !has(v) );                  //当集合中无元素v时继续执行
    assert( used < card );              //当集合还能增加元素时继续执行
    elem[used++] = v;
    return *this;
}
void  main(void)
{
    static_assert(sizeof(int) == 4);   //VS2019采用x86编译模式时为真，不终止编译运行
    SET  s(2);                          //定义集合只能存放两个元素
    s.push(1).push(2);;                 //存放第1，2个元素
    s.push(3);                          //因不能存放元素3，断言为假，程序被终止
}
```