華中科技大學

第10章 异常与断言

许向阳

xuxy@hust.edu.cn

大纲

華中科技大學

10.1 异常处理

10.2 捕获顺序

10.3 函数的异常接口

10.4 异常类型

10.5 异常对象的析构

10.6 断言

异常（Exception）

- 程序运行时，检测出将要发生不正常的情况
- 最常见的异常

  除数为0　　数组越界访问

  打开文件时，文件不存在

  无效数据，如输入的年龄为负数

  分配空间时，空间不够导致无法分配

  网络通讯时，网络不通等
- 异常代表着某些不该发生的事情发生了

10.1 异常处理

- **程序设计错误**是指程序员在设计程序时由于人为因素所产生的错误。
- **异常错误**是在程序运行时由于系统异常原因而产生的，不是程序员的失误所致。
- 如果不排除程序设计错误，程序就不能通过编译检查或运行结果不对。
- 如果不处理**异常错误**，程序执行时遇到异常情况就会突然终止或失控，使用户感到莫名其妙。

10.1 异常处理

- 异常处理即异常错误处理
  - 给出错误提示
  - 让程序沿一条不会出错的路径继续执行
  - 通知用户遇到了何种异常，不得不停止执行
  - 结束前做一些必要的工作，如将内存中的数据写入文件、关闭打开的文件、释放动态分配的内存空间等。

10.1 异常处理

Improved error recovery is one of the most powerful ways you can increase the robustness of your code.

10.1 异常处理


```
#include <fstream>
#include <iostream>
using namespace std;
int main(int argc, char ** argv)
{  ifstream source(argv[1]);      //打开文件
  char line[128];
  while(!source.eof()){
      source.getline(line, sizeof(line));
      cout <<line <<endl;
  }
  source.close( );
  return 0;
}
```

10.1 异常处理

异常处理的例子

```
int main(int argc, char ** argv)
{
  ifstream source(argv[1]);      //打开文件
  char line[128];
  if(source.fail( )){
    cout <<"error opening the file "<<argv[1] <<endl;
    exit(1);
  }
  while(!source.eof())
     { source.getline(line, sizeof(line));    cout <<line <<endl;  }
  source.close();
  return 0;
}
```

10.1 异常处理

- 谁负责检查是否出现了异常？
- 出现了什么异常？

  异常的信息：编号？ 异常提示串？

  什么位置？……
- 谁负责接收报告？

  自己处理？向上级报告？

  向上级的上级报告？
- 接到报告后如何处理？
- 在执行一段程序的过程中，可能出现多个异常，

  如何处理？

10.1 异常处理

C语言处理异常的方法

- 在检测到异常时，立即重做

  如输入的数据不正确时，重新输入，纠正错误

- 检查函数的返回值来发现异常错误

  发现异常的函数，无法纠错，向调用函数报告

- 使用signal( )和 raise( )函数

- 使用非局部的跳转goto函数

C语言处理异常的方法

- 用函数返回值表明出错信息
- 设置一个全局的出错标志

  标准C提供errno和perror( )来支持这种方法

- **方法的缺陷：繁琐**
  - 处理异常的代码和正常算法的代码交织在一起，增加了代码的复杂性，降低了可读性。
  - 程序员很容易忽视函数的返回值。printf( )
  - 设置全局的出错标志降低了各个模块的独立性

C语言处理异常的方法

- 使用C 语言标准库中的信号处理系统中的 **signal( )** 函数和 **raise( )**函数。

- 这种方法的缺陷：复杂
  - 程序员需要理解信号产生的机制并安装合适的信号处理机制。
  - 对于大型项目，不同库之间的信号可能会产生冲突。

C语言处理异常的方法

使用非局部的跳转goto函数

- 使用C标准库中非局部的跳转函数：
  setjmp( ) 和 longjmp( )。
- 缺陷：增加了模块之间的耦合性

- 缺陷：不能调用析构函数进行善后处理，不能释放对象占用的资源。实际上不可能有效正确地从异常情况中恢复出来。

10.1 异常处理

C++语言的异常处理方法

- C++的异常处理机制的基本思想

  将异常的检测与处理分离。

- C++中异常错误处理用**try**、**throw**和**catch**三个关键字实现

10.1 异常处理

```
#include <iostream>
int main( )
{  int m,n;
   cout<<"Please input two integers:"; cin>>m>>n;
   try
   {    if (n==0)   throw 0;
        cout<< (m/n)<<endl;
   }
   catch(int)
   {    cout<<"Divided by 0!"<<endl;
        return -1;
   }
  return 0;
}
```

在try代码块中包含需要监控的程序部分

抛出一个整型异常

catch语句捕获一个整型异常并处理

10.1 异常处理


```
try{
        ..... throw   1 ;       // 抛出int 型的异常
        ..... throw   "error"; // 抛出字符串型的异常
              throw  MyException("Cannot do it!");
}                              // MyException 是一个 类
catch (int  arg1){
       //exception handling for  int  type  }
catch (const char * arg2){
      //exception handling for  const char * type  }
catch (MyException arg3){
     //exception handling for type MyException  }
catch (...){  ..... }
```

10.1 异常处理


```
int main( )
{  int m,n;
   cout<<"Please input two integers:"; cin>>m>>n;
   try
   {    if (n==0)   throw "Divided by 0!";
        cout<< (m/n)<<endl;
   }
   catch(const char * arg)
   {    cout<<arg<<endl;
        return -1;
   }
  return 0;
}
```

变量arg用来接收throw抛出的异常值

10.1 异常处理

- 要监控的程序部分包含在**try**代码块中；
- 在**try**块中调用的函数也将被监控；
- 如果**try**块中的程序代码发生了异常错误，就使用**throw**抛出异常；
- **try**块中抛出的异常将被紧跟在**try**语句之后的**catch**语句捕获。
- **try** 与 **catch** 是紧密关联的，不能单独出现**try**，也不能单独出现 **catch**

10.2 捕获顺序

- 在try语句后面可以有**一个或多个**catch语句；
- **如果在catch语句中指定的数据类型与异常的类型匹配，那么这个catch语句将被执行。**

  所有其他的catch语句都将被忽略。
- 当异常信息被捕获时，变量arg将用来接收异常信息的值；
- 如果抛出的异常没有与之类型相匹配的catch语句，异常向上一级传递。
- 如果没有出现异常，则不会执行catch中的语句

10.2 捕获顺序


```
int main( )
{  int m,n;
    cout<<"Please input two integers:"; cin>>m>>n;
    try
    {  cout<<division(m,n)<<endl;   }
    catch(int)
    {      cout<<"Divided by 0!"<<endl;
           return -1;
    }
   return 0;
}
int division(int x,int y)
{   if (y==0)      throw 0;
    return x/y;
}
```

函数中未处理异常；

向上一级传递异常

10.2 捕获顺序

```
int main( )
{  int m,n;
   cout<<"Please input two integers:"; cin>>m>>n;
   try
   {    if (n==0)   throw 0;
        cout<< (m/n)<<endl;
   }
   catch(const char * arg)
   {    cout<<arg<<endl;
        return -1;
   }
  return 0;
}
```

抛出的异常的值与变量arg类型不配;
异常应向上一级传递，但main已是最后一级，程序终止

10.2 捕获顺序


```
void Xhandler(int test)
{
  try{    if(test) throw test;
          else throw "Value is zero";
  }
  catch(int i) {
    cout << "Caught One!  Ex. #: " << i << '\n';
  }
  catch(const char *str) {
    cout << "Caught a string: ";
    cout << str << '\n';
  }
}
```

每个catch语句所能捕获的异常必须是不同类型

抛出常量串异常，应用 const char * 来捕获

10.2 捕获顺序


```
int main( )
{  int m,n;
    cout<<"Please input two integers:";
    cin>>m>>n;
    try
    {     if (n==0)   throw 0;
          cout<< (m/n)<<endl;
    }
    catch(const  char * arg)   // 抛出的异常信息的值
                               // 与形参变量arg类型不配
    {     cout<<arg<<endl;
          return -1;
    }
     cout<<"here"<<endl;     //  无int 类型的异常处理，
     return 0;               //  引起非正常的程序终止，
                             //  不会显示 here
}
```

10.2 捕获顺序

非正常的程序终止

- 如果抛出的异常没有与之类型相匹配的catch语句，则该异常信息将被传递到调用该程序模块的上一级，它的上级捕获到这个异常信息后进行处理。
- 如果上一级模块仍然未处理，就再传递给其上一级，逐级上传。
- 如果到最高一级还无法处理。那么将发生非正常的程序终止（abnormal program termination）

自定义运行终止函数

- 如果程序中抛出了一个未被处理的异常信息，系统将调用C++标准库中的函数terminate( )，默认情况下，terminate( )用abort( ) 终止程序。

- 程序员可以编写自己的终止函数，通过set_terminate函数传递给异常处理模块，系统在找不到相匹配的异常错误处理模块时调用该函数。

10.2 捕获顺序


```
void myterminate() //自定义的运行终止函数
{
    cout<<"This is my terminater."<<endl;
    //...释放程序中申请的系统资源
    exit(1);
}
int main()
{
    try{
        set_terminate(myterminate);
        //...
        throw "Exception ... ";
    }
    catch(int i){  }
    return 0;
}
```

自定义的运行终止函数

10.2 捕获顺序

捕获所有的异常

```
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(...) { // 捕获所有的异常
    cout << "Caught One!\n";  }
}
```

10.2 捕获顺序

```
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 " << i << '\n';
   }
   catch(...) { // 捕获所有其他的异常
     cout << "Caught One!\n";
   }
}
```

捕获所有其他的异常

10.2 捕获顺序

华中科技大学

没有操作数的 throw 语句

- 只能出现在 catch 语句中；
- 向上一级传递这一异常；

- 在catch 中，可以出现含有操作数的throw语句，向上一级抛出新的异常；
- 在try 语句中，不能出现无操作数的throw。

10.3 函数的异常接口

- 可由该函数引发的、而其自身又不想捕获或处理的异常；
- 在函数的参数表后面定义异常接口，用throw列出要引发的异常类型；
- … func(…)；// 可引发任何异常
- … func(…) noexcep ； //不引发任何异常
  … func(…) throw( ); //不引发任何异常
  … func(…) throw(void); //不引发任何异常
- … func(…) throw(A, B, C);
  引发 A 类型、B类型、C类型的异常
- … func(…) const throw(A, B, C); (成员函数)

10.3 函数的异常接口


```
int sum(int a[ ], int t, int s, int c) throw (const char *)
{
    if (s < 0 || s >= t || s + c<0 || s + c>t)
          throw "subscription overflow";
        // 若发出const char *类型的异常，
        // 下面的语句不执行
    int r = 0,x=0;
    for (x = 0; x < c; x++)
        r += a[s+x];
    return r;
}
```

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；
- 若函数申明不引发任何异常的函数，但函数体又引发的异常；或者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；
- 函数模板和模板函数 可定义异常接口；
- 类模板及模板类的函数成员可定义异常接口
  构造函数和析构函数都可以定义异常接口

10.3 函数的异常接口


```
void f1()   // noexcept
{     throw "occur error"; }
int main() {
   try {      f1();   }
    catch (...) {
        cout << "catch any exception" << endl;
    }
    return 0;
}
```

对于 void f1( ) {…} ， 显示 catch any exception

对于 void f1( ) noexcept {…} ,

编译警告：不引发异常，但又包含了异常。

运行结果：不会显示 catch any exception , 异常终止

10.3 函数的异常接口

- noexcept可以表示throw()或throw(void)；
- noexcept一般用在不会出现异常的函数后面；
- noexcept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；
- throw(除void外的类型参数)不应出现在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。

10.4 异常类型

- 程序员可自己创建异常类型；
- 在实际程序中，大多数异常的类型都是类，而不是内置数据类型（标准类型）；
- 创建一个类来描述发生的错误信息，可以帮助异常处理模块处理错误；
- catch可以捕获任意类型的异常，

使用异常类

```
#include <iostream>
using namespace std;
class MyException {
public:
    char str_what[80];
    MyException() { *str_what = 0; }
    MyException(const char *s) {
        strcpy(str_what, s);
    }
};
```

10.4 异常类型

使用异常类

```
int main()
{    int a, b;
     try {
     cout << "Enter numerator and denominator: ";
     cin >> a >> b;
     if(!b)    throw MyException("Cannot divide by 0!");
     else   cout << "Quotient is " << a/b << "\n";
     }
  catch (MyException e) { // catch an error
    cout << e.str_what << "\n";
  }
  return 0;
}
```

10.4 异常类型

使用异常类

```
try {
        throw  MyException("Cannot divide by 0!");
   //    MyException    temp("Cannot divide by 0!");
   //     throw  temp;
}
catch (MyException &e) {  }
//  catch (MyException e) {  }
```

不同的写法，执行过程有差异；
细节等同于函数的参数的传递

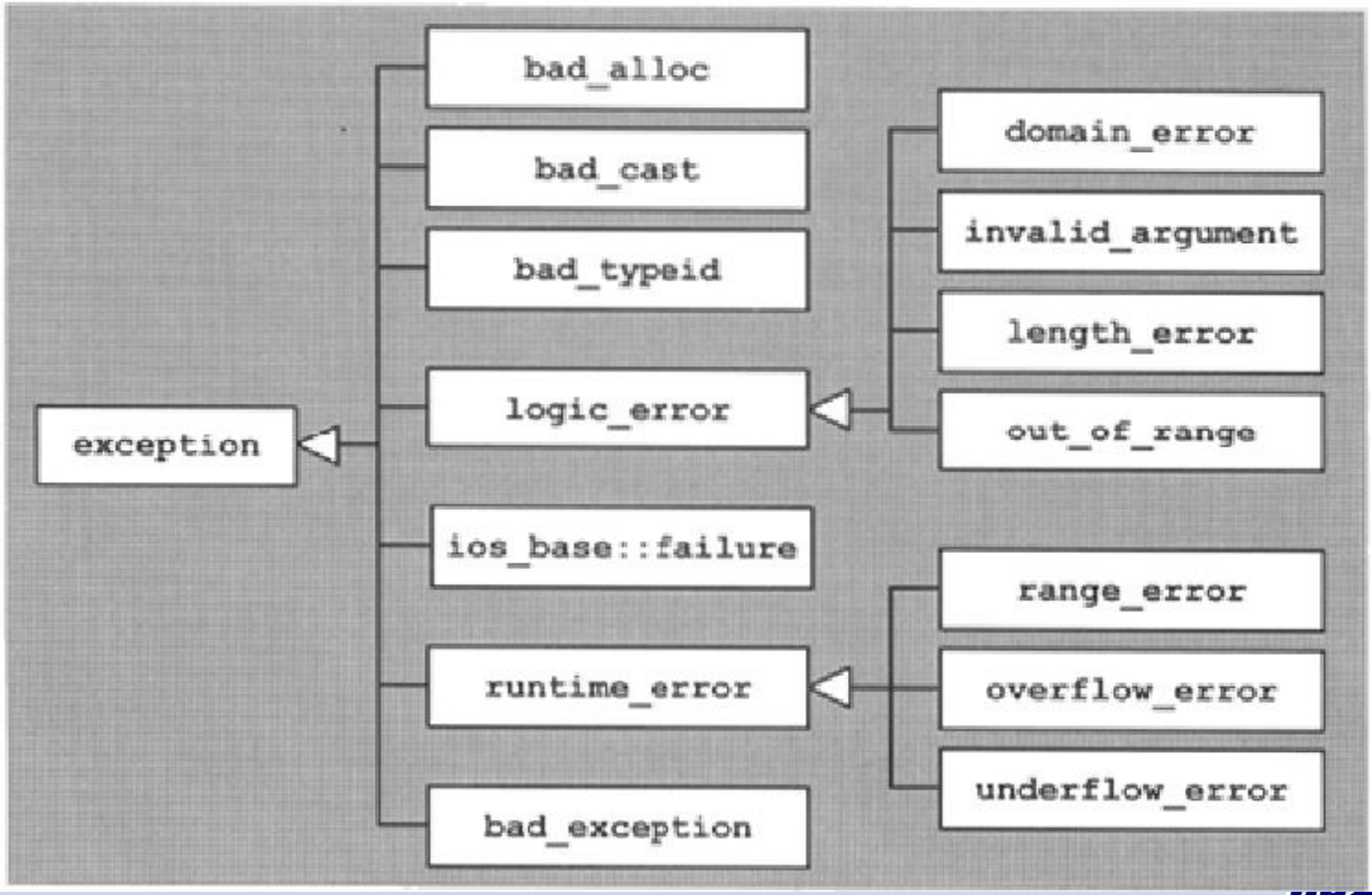
10.4 异常类型

- 如果父类A的子类为B，B类异常能被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，指向可写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。
- 注意catch(const volatile void *)能捕获任意指针类型的异常，catch(...)能捕获任意类型的异常。

10.4 异常类型

- 如果通过new产生的指针类型的异常，在catch处理后，通常应使用delete释放内存；
- 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；
- 调用链中通过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。

10.4 异常类型


exception
bad_alloc
bad_cast
bad_typeid
logic_error
ios_base::failure
runtime_error
bad_exception
domain_error
invalid_argument
length_error
out_of_range
range_error
overflow_error
underflow_error

10.4 异常类型

标准异常的名字	抛出异常的主体	对应的头文件
bad_alloc	new	< new >
bad_cast	dynamic_cast	<typeinfo>
bad_typeid	typeid	<typeinfo>
bad_exception	exception specification	<exception>

10.4 异常类型


```
#include <iostream>
#include <new>     //需要包含该头文件
using namespace std;
...
   try {
     p = new int[32]; // 为整型数组申请动态存储单元
   }
   catch (bad_alloc xa) {
     cout << "Allocation failure.分配内存失败。\n";
     return 1;
   }
...
```

10.4 异常类型

- 将程序中正常处理的代码（描述问题的算法）与异常处理代码分离开来，提高了程序的可读性。

- 提供了一种更规则的处理异常的风格，便于软件项目组人员之间的合作。
  - 通常情况下，类的创建者监控代码段，从类中抛出异常。
  - 类的使用者捕获到异常并处理。

10.4 异常类型

- 在异常发生时，能够撤销对象，并自动调用析构函数进行善后处理，释放对象所占用的系统资源。

发生异常时资源释放

```
class Y {
        int* p;
        void init() ;
public:
        Y(int s) { p = new int[s] ; init( ) ; }
        ~Y( ) { delete[] p; }
        / / ...
};
```

A safe variant

```
class Z {
        vector<int> p;
        void init() ;
public:
        Z(int s) : p(s) { init( ) ; }
        / / ...
};
```

10.6 断言

- 断言（assert）是一个带有整型参数的用于调试程序的函数，在运行时检查断言；
- 如果实参的值为真则程序继续执行；
- 实参为假，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。
- static_assert定义的断言在编译时检查，为假时终止编译运行。

10.6 断言


```
#include <assert.h>
class  SET {
    int* elem, used, card;
public:      SET(int card);
    virtual int has(int)const;
    virtual SET& push (int);                    //插入一个元素
    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;
}
```

10.6 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);      //当集合能增加元素时继续执行
    elem[used++] = v;
    return *this;
}
```

10.6 断言


```
void  main(void)
{
    static_assert(sizeof(int)==4);   // 静态断言
      //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
}
```

总结

- 什么是异常？怎样抛出异常？怎样捕获异常？
- 什么是断言？编译时断言和运行时断言有何差别？
- try …… catch 关联在一起使用
- throw 可间接在一个 try 块中，即 含有throw语句的函数，或者其更上一级的调用函数在 try 块中。
- 一个异常若没有被任何 catch 所捕获，则引起程序异常终止；
- 多个catch 的运行规则，按顺序检查是否匹配、匹配一个后，后面的catch 不再检查。