**C++ - 异常类(exception classes) 的 详解 及 代码**

C++ - 异常类(exception classes) 的 详解 及 代码

本文地址: http://blog.csdn.net/caroline\_wendy/article/details/17498665

**异常类(exception classes)**包含4种基类,**bad\_cast**, **bad\_alloc**,**runtime\_error**,**logic\_error**;

**runtime\_error**: 错误只有当程序运行时, 才能检测出来;

**logic\_error**: 应用程序检测出的逻辑错误.

其中bad\_cast, bad\_alloc使用**默认构造器**, runtime\_error, logic\_error, 必须**使用string(C-style或stl)**进行初始化;

用户定义的类也可以**继承(inherit)**自异常类, 并初始化相应的参数;

**代码如下**:

**/\***

**\* cppprimer.cpp**

**\***

**\* Created on: 2013.11.28**

**\* Author: Caroline**

**\*/**

**/\*eclipse cdt, gcc 4.8.1\*/**

**/\* setlocale example \*/**

**#include <iostream>**

**#include <string>**

**#include <stdexcept>**

**using namespace std;**

**class out\_of\_stock : public std::runtime\_error**

**{**

**public:**

**//初始化基类的runtime\_error**

**explicit out\_of\_stock(const std::string &s) : std::runtime\_error(s) {}**

**};**

**class id\_mismatch : public std::logic\_error**

**{**

**public:**

**explicit id\_mismatch(const std::string &s) : std::logic\_error(s) {}**

**id\_mismatch(const std::string &s, const std::string &lhs, const std::string &rhs) :**

**std::logic\_error(s), left(lhs), right(rhs) {}**

**const std::string left, right;**

**};**

**struct My\_food**

**{**

**My\_food(const std::string fi, const int fn) : food\_id(fi), food\_num(fn) {}**

**My\_food& operator+=(const My\_food& rhs);**

**std::string food\_id;**

**int food\_num;**

**};**

**My\_food& My\_food::operator+=(const My\_food& rhs)**

**{**

**if(this->food\_id != rhs.food\_id)**

**throw id\_mismatch("wrong id", this->food\_id, rhs.food\_id);**

**this->food\_num += rhs.food\_num;**

**return \*this;**

**}**

**int main (void)**

**{**

**My\_food M1("Apple", 3), M2("Pear", 2), M3("Apple", 2);**

**try{**

**M1 += M3;**

**std::cout << M1.food\_id << ":" << M1.food\_num << std::endl;**

**M1 += M2;**

**}catch(const id\_mismatch &e){**

**std::cerr << e.what() << ": left id("**

**<< e.left <<") right id(" << e.right << ")" << std::endl;**

**}**

**}**

**输出:**

Apple:5

wrong id: left id(Apple) right id(Pear)

# C++异常类型以及多级catch

首先来回顾一下上节讲到的 try-catch 的用法：

try{

// 可能抛出异常的语句

}catch(exceptionType variable){

// 处理异常的语句

}

我们还遗留下一个问题，就是 catch 关键字后边的exceptionType variable，这节就来详细分析一下。

exceptionType是异常类型，它指明了当前的 catch 可以处理什么类型的异常；variable是一个变量，用来接收异常信息。当程序抛出异常时，会创建一份数据，这份数据包含了错误信息，程序员可以根据这些信息来判断到底出了什么问题，接下来怎么处理。

异常既然是一份数据，那么就应该有数据类型。C++ 规定，异常类型可以是 int、char、float、bool 等基本类型，也可以是指针、数组、字符串、结构体、类等聚合类型。C++ 语言本身以及标准库中的函数抛出的异常，都是 exception 类或其子类的异常。也就是说，抛出异常时，会创建一个 exception 类或其子类的对象。

exceptionType variable和函数的形参非常类似，当异常发生后，会将异常数据传递给 variable 这个变量，这和函数传参的过程类似。当然，只有跟 exceptionType 类型匹配的异常数据才会被传递给 variable，否则 catch 不会接收这份异常数据，也不会执行 catch 块中的语句。换句话说，catch 不会处理当前的异常。

我们可以将 catch 看做一个没有返回值的函数，当异常发生后 catch 会被调用，并且会接收实参（异常数据）。

·但是 catch 和真正的函数调用又有区别：

·真正的函数调用，形参和实参的类型必须要匹配，或者可以自动转换，否则在编译阶段就报错了。

而对于 catch，异常是在运行阶段产生的，它可以是任何类型，没法提前预测，所以不能在编译阶段判断类型是否正确，只能等到程序运行后，真的抛出异常了，再将异常类型和 catch 能处理的类型进行匹配，匹配成功的话就“调用”当前的 catch，否则就忽略当前的 catch。

总起来说，catch 和真正的函数调用相比，多了一个「在运行阶段将实参和形参匹配」的过程。

另外需要注意的是，如果不希望 catch 处理异常数据，也可以将 variable 省略掉，也即写作：

try{

// 可能抛出异常的语句

}catch(exceptionType){

// 处理异常的语句

}

这样只会将异常类型和 catch 所能处理的类型进行匹配，不会传递异常数据了。

多级 catch

前面的例子中，一个 try 对应一个 catch，这只是最简单的形式。其实，一个 try 后面可以跟多个 catch：

try{

//可能抛出异常的语句

}catch(exception\_type\_1 e){

//处理异常的语句

}catch(exception\_type\_2 e){

//处理异常的语句

}

//其他的catch

catch(exception\_type\_n e){

//处理异常的语句

}

当异常发生时，程序会按照从上到下的顺序，将异常类型和 catch 所能接收的类型逐个匹配。一旦找到类型匹配的 catch 就停止检索，并将异常交给当前的 catch 处理（其他的 catch 不会被执行）。如果最终也没有找到匹配的 catch，就只能交给系统处理，终止程序的运行。

下面的例子演示了多级 catch 的使用：

**#include<iostream>**

**#include<string>**

**usingnamespacestd;**

**class Base{};**

**class Derived:publicBase{};**

**int main(){**

**try{**

**throw Derived();//抛出自己的异常类型，实际上是创建一个Derived类型的匿名对象**

**cout<<"This statement will not be executed."<<endl;**

**}catch(int){**

**cout<<"Exception type: int"<<endl;**

**}catch(char\*){**

**cout<<"Exception type: cahr \*"<<endl;**

**}catch(Base){//匹配成功（向上转型）**

**cout<<"Exception type: Base"<<endl;**

**}catch(Derived){**

**cout<<"Exception type: Derived"<<endl;**

**}**

**return 0;**

**}**

运行结果：

Exception type: Base

* 1

在 catch 中，我们只给出了异常类型，没有给出接收异常信息的变量。

本例中，我们定义了一个基类 Base，又从 Base 派生类出了 Derived。抛出异常时，我们创建了一个 Derived 类的匿名对象，也就是说，异常的类型是 Derived。

我们期望的是，异常被catch(Derived)捕获，但是从输出结果可以看出，异常提前被catch(Base)捕获了，这说明 catch 在匹配异常类型时发生了向上转型（Upcasting）。

catch 在匹配过程中的类型转换

C/C++ 中存在多种多样的类型转换，以普通函数（非模板函数）为例，发生函数调用时，如果实参和形参的类型不是严格匹配，那么会将实参的类型进行适当的转换，以适应形参的类型，这些转换包括：

算数转换：例如 int 转换为 float，char 转换为 int，double 转换为 int 等。

向上转型：也就是派生类向基类的转换，请猛击《C++向上转型（将派生类赋值给基类）》了解详情。

const 转换：也即将非 const 类型转换为 const 类型，例如将 char \* 转换为 const char \*。

数组或函数指针转换：如果函数形参不是引用类型，那么数组名会转换为数组指针，函数名也会转换为函数指针。

用户自定的类型转换。

catch 在匹配异常类型的过程中，也会进行类型转换，但是这种转换受到了更多的限制，仅能进行「向上转型」、「const 转换」和「数组或函数指针转换」，其他的都不能应用于 catch。

向上转型在上面的例子中已经发生了，下面的例子演示了 const 转换以及数组和指针的转换：

**#include<iostream>**

**using namespace std;**

**int main(){**

**int nums[]={1,2,3};**

**try{**

**throw nums;**

**cout<<"This statement will not be executed."<<endl;**

**}catch(constint\*){**

**cout<<"Exception type: const int \*"<<endl;**

**}**

**return 0;**

**}**

运行结果：

Exception type: const int \*

* 1

nums 本来的类型是int [3]，但是 catch 中没有严格匹配的类型，所以先转换为int \*，再转换为const int \*。

**C++中的异常类型**

C++语言本身或标准程序库所抛出的所有异常，都派生自[基类](https://so.csdn.net/so/search?q=%E5%9F%BA%E7%B1%BB&spm=1001.2101.3001.7020)exception。这是其他数个标准异常类别的基类，它们共同构成一个类体系：

**这些标准异常类别分为三组：**

**（1）语言本身所支持的异常**

此类异常用以支撑某些语言特性。主要包括：  
bad\_alloc：new操作失败会抛出。  
bad\_cast：执行期间加在一个引用上面的动态性型别转换操作失败时抛出。  
bad\_typeid：执行RTTI时，交给typeid的参数为零或空指针时抛出  
bad\_exception：非预期的异常

**（2）C++标准程序库发出的异常**

总是派生自**logic\_error**。**逻辑错误是由于程序内部逻辑而导致的错误。逻辑错误是可以避免的，且在程序开始执行之前，能够被检测到**。  
C++标准库中定义的逻辑错误如下：

1. class logic\_error : public exception {
2. public:
3. explicit logic\_error (const string& what\_arg);
4. };
6. class invalid\_argument : public logic\_error {
7. public:
8. explicit invalid\_argument (const string& what\_arg);
9. };
11. class out\_of\_range : public logic\_error {
12. public:
13. explicit out\_of\_range (const string& what\_arg);
14. };
16. class length\_error : public logic\_error {
17. public:
18. explicit length\_error (const string& what\_arg);
19. };
21. class domain\_error : public logic\_error {
22. public:
23. explicit domain\_error (const string& what\_arg);
24. };

错误分类解释及举例：

**domain\_error：专业领域内的范畴 invalid\_argument：无效参数，比如讲bitset以char而非0或1进行初始化 length\_error：可能超越了最大极限，比如对着某个字符串附加太多字符。 out\_of\_range：参数不再预期范围内。例如在诸如array的容器或字符串string中采用一个错误索引。**

**（3）程序作用域之外发出的异常**

总是派生自**runtime\_error**，用来指出“**不在程序范围内，且不容易回避”的事件。此类错误只在程序执行时才是可检测的**。C++标准库中的定义如下：

1. class runtime\_error : public exception {
2. public:
3. explicit runtime\_error (const string& what\_arg);
4. };
6. class range\_error : public runtime\_error {
7. public:
8. explicit range\_error (const string& what\_arg);
9. };
11. class overflow\_error : public runtime\_error {
12. public:
13. explicit overflow\_error (const string& what\_arg);
14. };
16. class underflow\_error : public runtime\_error {
17. public:
18. explicit underflow\_error (const string& what\_arg);
19. };

range\_error：内部计算时发生区间错误

overflow\_error：算数运算时发生上溢

underflow\_error：算数运算时发生下溢

**Update:**  
实例代码：

#include <iostream>

#include <string>

#include <bitset>

#include <typeinfo>

#include <vector>

#include <stdexcept>

using namespace std;

//自定义配置器，vector分配空间使用

template<class \_Ty>

class stingyallocator : public allocator<\_Ty>

{

public:

template <class U>

struct rebind {

typedef stingyallocator<U> other;

};

size\_t max\_size( ) const

{

return 10;

};

};

int main()

{

//逻辑错误：out\_of\_range

try {

string str( "Micro" );

string rstr( "soft" );

str.append( rstr, 5, 3 );

cout << str << endl;

}

catch ( exception &e ) {

cerr << "Caught: " << e.what( ) << endl;

cerr << "Type: " << typeid( e ).name( ) << endl << endl;

};

//逻辑错误：length\_error

try

{

vector<int, stingyallocator< int > > myv;

for ( int i = 0; i < 11; i++ )

myv.push\_back( i );

}

catch ( exception &e )

{

cerr << "Caught " << e.what( ) << endl;

cerr << "Type " << typeid( e ).name( ) << endl << endl;

};

//逻辑错误：invalid\_argument

try

{

bitset< 32 > bitset( string( "11001010101100001b100101010110000") );

}

catch ( exception &e )

{

cerr << "Caught " << e.what( ) << endl;

cerr << "Type " << typeid( e ).name( ) << endl << endl;

};

//逻辑错误：domain\_error

try

{

throw domain\_error( "Your domain is in error!" );

}

catch (exception &e)

{

cerr << "Caught: " << e.what( ) << endl;

cerr << "Type: " << typeid(e).name( ) << endl << endl;

};

//运行时错误：range\_error

try

{

throw range\_error( "The range is in error!" );

}

catch (exception &e)

{

cerr << "Caught: " << e.what( ) << endl;

cerr << "Type: " << typeid( e ).name( ) << endl << endl << endl;

};

//运行时错误：underflow\_error

try

{

throw underflow\_error( "The number's a bit small, captain!" );

}

catch ( exception &e ) {

cerr << "Caught: " << e.what( ) << endl;

cerr << "Type: " << typeid( e ).name( ) << endl << endl;

};

//运行时错误：overflow\_error

try

{

bitset< 33 > bitset;

bitset[32] = 1;

bitset[0] = 1;

unsigned long x = bitset.to\_ulong( );

}

catch(exception &e)

{

cerr << "Caught " << e.what() << endl;

cerr << "Type: " << typeid(e).name() << endl << endl;

}

return 0;

}

运行结果：

推荐参考资料：

【1】《C++Primer（第三版）》第855页 19.2.8C++标准的异常类层次结构  
【2】《C++标准程序库》第25页 3.3 错误处理和异常处理  
【3】[Confused about std::runtime\_error vs. std::logic\_error](http://stackoverflow.com/questions/2924058/confused-about-stdruntime-error-vs-stdlogic-error)  
【4】[C++中异常类的使用方法](http://blog.csdn.net/goodlixueyong/article/details/6533733)

 下面自定义异常类，继承自logic\_error,MyException.h如下：

**#if !defined(AFX\_MYEXCEPTION\_H\_\_EB39C92B\_B867\_41AA\_83A6\_E614FD4AEC63\_\_INCLUDED\_)**

**#define AFX\_MYEXCEPTION\_H\_\_EB39C92B\_B867\_41AA\_83A6\_E614FD4AEC63\_\_INCLUDED\_**

**#if \_MSC\_VER > 1000**

**#pragma once**

**#endif // \_MSC\_VER > 1000**

**#include<string>**

**#include<stdexcept>**

**class MyException:public std::logic\_error**

**{**

**public:**

**MyException(std::string s,std::string myinfo);**

**const char\* what() const;**

**virtual ~MyException();**

**private:**

**std::string myinfo;**

**};**

**#endif // !defined(AFX\_MYEXCEPTION\_H\_\_EB39C92B\_B867\_41AA\_83A6\_E614FD4AEC63\_\_INCLUDED\_)**

MyException.cpp内容如下：

**#include "MyException.h"**

**MyException::MyException(std::string s,std::string myinfo):logic\_error(s)**

**{**

**this->myinfo=myinfo;**

**}**

**MyException::~MyException()**

**{**

**}**

**const char\* MyException::what() const**

**{**

**return myinfo.c\_str();               //此处**

**16.  }**

# C++中的异常处理机制详解

**异常处理**

增强错误恢复能力是提高代码健壮性的最有力的途径之一，C语言中采用的错误处理方法被认为是紧耦合的，函数的使用者必须在非常靠近函数调用的地方编写错误处理代码，这样会使得其变得笨拙和难以使用。C++中引入了异常处理机制，这是C++的主要特征之一，是考虑问题和处理错误的一种更好的方式。使用错误处理可以带来一些优点，如下:

错误处理代码的编写不再冗长乏味，并且不再和正常的代码混合在一起，程序员只需要编写希望产生的代码，然后在后面某个单独的区段里编写处理错误的嗲吗。多次调用同一个函数，则只需要某个地方编写一次错误处理代码。

错误不能被忽略，如果一个函数必须向调用者发送一次错误信息。它将抛出一个描述这个错误的对象。

**传统的错误处理和异常处理**

在讨论异常处理之前，我们先谈谈C语言中的传统错误处理方法，这里列举了如下三种:

在函数中返回错误，函数会设置一个全局的错误状态标志。

使用信号来做信号处理系统，在函数中raise信号，通过signal来设置信号处理函数，这种方式耦合度非常高，而且不同的库产生的信号值可能会发生冲突

使用标准C库中的非局部跳转函数 setjmp和longjmp ，这里使用setjmp和longjmp来演示下如何进行错误处理:

#include

#include

jmp\_buf static\_buf; //用来存放处理器上下文，用于跳转

void do\_jmp()

{

//do something,simetime occurs a little error

//调用longjmp后，会载入static\_buf的处理器信息，然后第二个参数作为返回点的setjmp这个函数的返回值

longjmp(static\_buf,10);//10是错误码，根据这个错误码来进行相应的处理

}

int main()

{

int ret = 0;

//将处理器信息保存到static\_buf中,并返回0，相当于在这里做了一个标记，后面可以跳转过来

if((ret = setjmp(static\_buf)) == 0) {

//要执行的代码

do\_jmp();

} else { //出现了错误

if (ret == 10)

std::cout << "a little error" << std::endl;

}

}

错误处理方式看起来耦合度不是很高，正常代码和错误处理的代码分离了，处理处理的代码都汇聚在一起了。但是基于这种局部跳转的方式来处理代码，在C++中却存在很严重的问题，那就是对象不能被析构，局部跳转后不会主动去调用已经实例化对象的析构函数。这将导致内存泄露的问题。下面这个例子充分显示了这点

#include

#include

using namespace std;

class base {

public:

base() {

cout << "base construct func call" << endl;

}

~base() {

cout << "~base destruct func call" << endl;

}

};

jmp\_buf static\_buf;

void test\_base() {

base b;

//do something

longjmp(static\_buf,47);//进行了跳转，跳转后会发现b无法析构了

}

int main() {

if(setjmp(static\_buf) == 0) {

cout << "deal with some thing" << endl;

test\_base();

} else {

cout << "catch a error" << endl;

}

}

在上面这段代码中，只有base类的构造函数会被调用，当longjmp发生了跳转后，b这个实例将不会被析构掉，但是执行流已经无法回到这里，b这个实例将不会被析构。这就是局部跳转用在C++中来处理错误的时候带来的一些问题，在C++中异常则不会有这些问题的存在。那么接下来看看如何定义一个异常，以及如何抛出一个异常和捕获异常吧.

**异常的抛出**

class MyError {

const char\* const data;

public:

MyError(const char\* const msg = 0):data(msg)

{

//idle

}

};

void do\_error() {

throw MyError("something bad happend");

}

int main()

{

do\_error();

}

上面的例子中，通过throw抛出了一个异常类的实例，这个异常类，可以是任何一个自定义的类，通过实例化传入的参数可以表明发生的错误信息。其实异常就是一个带有异常信息的类而已。异常被抛出后，需要被捕获，从而可以从错误中进行恢复，那么接下来看看如何去捕获一个异常吧。在上面这个例子中使用抛出异常的方式来进行错误处理相比与之前使用局部跳转的实现来说，最大的不同之处就是异常抛出的代码块中，对象会被析构，称之为堆栈反解．

**异常的捕获**

C++中通过catch关键字来捕获异常，捕获异常后可以对异常进行处理，这个处理的语句块称为异常处理器。下面是一个简单的捕获异常的例子:

try{

//do something

throw string("this is exception");

} catch(const string& e) {

cout << "catch a exception " << e << endl;

}

catch有点像函数，可以有一个参数，throw抛出的异常对象，将会作为参数传递给匹配到到catch，然后进入异常处理器，上面的代码仅仅是展示了抛出一种异常的情况，加入try语句块中有可能会抛出多种异常的，那么该如何处理呢，这里是可以接多个catch语句块的，这将导致引入另外一个问题，那就是如何进行匹配。

**异常的匹配**

异常的匹配我认为是符合函数参数匹配的原则的，但是又有些不同，函数匹配的时候存在类型转换，但是异常则不然，在匹配过程中不会做类型的转换，下面的例子说明了这个事实:

#include

using namespace std;

int main()

{

try{

throw 'a';

}catch(int a) {

cout << "int" << endl;

}catch(char c) {

cout << "char" << endl;

}

}

上面的代码的输出结果是char，因为抛出的异常类型就是char，所以就匹配到了第二个异常处理器。可以发现在匹配过程中没有发生类型的转换。将char转换为int。尽管异常处理器不做类型转换，但是基类可以匹配到派生类这个在函数和异常匹配中都是有效的，但是需要注意catch的形参需要是引用类型或者是指针类型，否则会 导致切割派生类这个问题。

//基类

class Base{

public:

Base(string msg):m\_msg(msg)

{

}

virtual void what(){

cout << m\_msg << endl;

}

void test()

{

cout << "I am a CBase" << endl;

}

protected:

string m\_msg;

};

//派生类，重新实现了虚函数

class CBase : public Base

{

public:

CBase(string msg):Base(msg)

{

}

void what()

{

cout << "CBase:" << m\_msg << endl;

}

};

int main()

{

try {

//do some thing

//抛出派生类对象

throw CBase("I am a CBase exception");

}catch(Base& e) { //使用基类可以接收

e.what();

}

}

上面的这段代码可以正常的工作，实际上我们日常编写自己的异常处理函数的时候也是通过继承标准异常来实现字节的自定义异常的，但是如果将Base&换成Base的话，将会导致对象被切割，例如下面这段代码将会编译出错，因为CBase被切割了，导致CBase中的test函数无法被调用。

try {

//do some thing

throw CBase("I am a CBase exception");

}catch(Base e) {

e.test();

}

到此为此，异常的匹配算是说清楚了，总结一下，异常匹配的时候基本上遵循下面几条规则:

异常匹配除了必须要是严格的类型匹配外，还支持下面几个类型转换．

允许非常量到常量的类型转换，也就是说可以抛出一个非常量类型，然后使用catch捕捉对应的常量类型版本

允许从派生类到基类的类型转换

允许数组被转换为数组指针，允许函数被转换为函数指针

假想一种情况，当我要实现一代代码的时候，希望无论抛出什么类型的异常我都可以捕捉到，目前来说我们只能写上一大堆的catch语句捕获所有可能在代码中出现的异常来解决这个问题，很显然这样处理起来太过繁琐，幸好C++提供了一种可以捕捉任何异常的机制，可以使用下列代码中的语法。

catch(...) {  
    //异常处理器，这里可以捕捉任何异常，带来的问题就是无法或者异常信息  
   }  
如果你要实现一个函数库，你捕捉了你的函数库中的一些异常，但是你只是记录日志，并不去处理这些异常，处理异常的事情会交给上层调用的代码来处理．对于这样的一个场景C++也提供了支持．

try{

throw Exception("I am a exception");

}catch(...) {

//log the exception

throw;

}

通过在catch语句块中加入一个throw，就可以把当前捕获到的异常重新抛出．在异常抛出的那一节中，我在代码中抛出了一个异常，但是我没有使用任何catch语句来捕获我抛出的这个异常，执行上面的程序会出现下面的结果．

terminate called after throwing an instance of 'MyError'  
Aborted (core dumped)

为什么会出现这样的结果呢?，当我们抛出一个异常的时候，异常会随着函数调用关系，一级一级向上抛出，直到被捕获才会停止，如果最终没有被捕获将会导致调用terminate函数，上面的输出就是自动调用terminate函数导致的，为了保证更大的灵活性，C++提供了set\_terminate函数可以用来设置自己的terminate函数．设置完成后，抛出的异常如果没有被捕获就会被自定义的terminate函数进行处理．下面是一个使用的例子:

#include

#include

#include

using namespace std;

class MyError {

const char\* const data;

public:

MyError(const char\* const msg = 0):data(msg)

{

//idle

}

};

void do\_error() {

throw MyError("something bad happend");

}

//自定义的terminate函数，函数原型需要一致

void terminator()

{

cout << "I'll be back" << endl;

exit(0);

}

int main()

{

//设置自定义的terminate，返回的是原有的terminate函数指针

void (\*old\_terminate)() = set\_terminate(terminator);

do\_error();

}

 上面的代码会输出I'll be back  
到此为此关于异常匹配的我所知道的知识点都已经介绍完毕了，那么接着可以看看下一个话题，异常中的资源清理．

**异常中的资源清理**

在谈到局部跳转的时候，说到局部调转不会调用对象的析构函数，会导致内存泄露的问题，C++中的异常则不会有这个问题，C++中通过堆栈反解将已经定义的对象进行析构，但是有一个例外就是构造函数中如果出现了异常，那么这会导致已经分配的资源无法回收，下面是一个构造函数抛出异常的例子:

#include

#include

using namespace std;

class base

{

public:

base()

{

cout << "I start to construct" << endl;

if (count == 3) //构造第四个的时候抛出异常

throw string("I am a error");

count++;

}

~base()

{

cout << "I will destruct " << endl;

}

private:

static int count;

};

int base::count = 0;

int main()

{

try{

base test[5];

} catch(...){

cout << "catch some error" << endl;

}

}

 上面的代码输出结果是:

I start to construct

I start to construct

I start to construct

I start to construct

I will destruct

I will destruct

I will destruct

catch some error

在上面的代码中构造函数发生了异常，导致对应的析构函数没有执行，因此实际编程过程中应该避免在构造函数中抛出异常，如果没有办法避免，那么一定要在构造函数中对其进行捕获进行处理．最后介绍一个知识点就是函数try语句块，如果main函数可能会抛出异常该怎么捕获?，如果构造函数中的初始化列表可能会抛出异常该怎么捕获?下面的两个例子说明了函数try语句块的用法:

#include

using namespace std;

int main() try {

throw "main";

} catch(const char\* msg) {

cout << msg << endl;

return 1;

}

 main函数语句块，可以捕获main函数中抛出的异常．

class Base

{

public:

Base(int data,string str)try:m\_int(data),m\_string(str)//对初始化列表中可能会出现的异常也会进行捕捉

{

// some initialize opt

}catch(const char\* msg) {

cout << "catch a exception" << msg << endl;

}

private:

int m\_int;

string m\_string;

};

int main()

{

Base base(1,"zhangyifei");

}

上面说了很多都是关于异常的使用，如何定义自己的异常，编写异常是否应该遵循一定的标准，在哪里使用异常，异常是否安全等等一系列的问题，下面会一一讨论的．

**标准异常**

C++标准库给我们提供了一系列的标准异常，这些标准异常都是从exception类派生而来，主要分为两大派生类，一类是logic\_error，另一类则是runtime\_error这两个类在stdexcept头文件中，前者主要是描述程序中出现的逻辑错误，例如传递了无效的参数，后者指的是那些无法预料的事件所造成的错误，例如硬件故障或内存耗尽等，这两者都提供了一个参数类型为std::string的构造函数，这样就可以将异常信息保存起来，然后通过what成员函数得到异常信息．

#include

#include

#include

using namespace std;

class MyError:public runtime\_error {

public:

MyError(const string& msg = "") : runtime\_error(msg) {}

};

//runtime\_error logic\_error 两个都是继承自标准异常，带有string构造函数

//

int main()

{

try {

throw MyError("my message");

} catch(MyError& x) {

cout << x.what() << endl;

}

}

**异常规格说明**

假设一个项目中使用了一些第三方的库，那么第三方库中的一些函数可能会抛出异常，但是我们不清楚，那么C++提供了一个语法，将一个函数可能会抛出的异常列出来，这样我们在编写代码的时候参考函数的异常说明即可，但是C++11中这中异常规格说明的方案已经被取消了，所以我不打算过多介绍，通过一个例子看看其基本用法即可，重点看看C++11中提供的异常说明方案:

#include

#include

#include

#include

using namespace std;

class Up{};

class Fit{};

void g();

//异常规格说明，f函数只能抛出Up 和Fit类型的异常

void f(int i)throw(Up,Fit) {

switch(i) {

case 1: throw Up();

case 2: throw Fit();

}

g();

}

void g() {throw 47;}

void my\_ternminate() {

cout << "I am a ternminate" << endl;

exit(0);

}

void my\_unexpected() {

cout << "unexpected exception thrown" << endl;

// throw Up();

throw 8;

//如果在unexpected中继续抛出异常，抛出的是规格说明中的 则会被捕捉程序继续执行

//如果抛出的异常不在异常规格说明中分两种情况

//1.异常规格说明中有bad\_exception ，那么会导致抛出一个bad\_exception

//2.异常规格说明中没有bad\_exception 那么会导致程序调用ternminate函数

// exit(0);

}

int main() {

set\_terminate(my\_ternminate);

set\_unexpected(my\_unexpected);

for(int i = 1;i <=3;i++)

{

//当抛出的异常,并不是异常规格说明中的异常时

//会导致最终调用系统的unexpected函数，通过set\_unexpected可以

//用来设置自己的unexpected汗函数

try {

f(i);

}catch(Up) {

cout << "Up caught" << endl;

}catch(Fit) {

cout << "Fit caught" << endl;

}catch(bad\_exception) {

cout << "bad exception" << endl;

}

}

}

}

上面的代码说明了异常规格说明的基本语法，以及unexpected函数的作用，以及如何自定义自己的unexpected函数，还讨论了在unexpected函数中继续抛出异常的情况下，该如何处理抛出的异常.C++11中取消了这种异常规格说明．引入了一个noexcept函数，用于表明这个函数是否会抛出异常

void recoup(int) noexecpt(true); //recoup不会抛出异常

void recoup(int) noexecpt(false); //recoup可能会抛出异常

此外还提供了noexecpt用来检测一个函数是否不抛出异常．

**异常安全**

异常安全我觉得是一个挺复杂的点，不光光需要实现函数的功能，还要保存函数不会在抛出异常的情况下，出现不一致的状态．这里举一个例子，大家在实现堆栈的时候经常看到书中的例子都是定义了一个top函数用来获得栈顶元素，还有一个返回值是void的pop函数仅仅只是把栈顶元素弹出，那么为什么没有一个pop函数可以 即弹出栈顶元素，并且还可以获得栈顶元素呢?

template<typename T> T stack<T>::pop()

{

if(count == 0)

throw logic\_error("stack underflow");

else

return data[--count];

}

如果函数在最后一行抛出了一个异常，那么这导致了函数没有将退栈的元素返回，但是Count已经减１了，所以函数希望得到的栈顶元素丢失了．本质原因是因为这个函数试图一次做两件事，1.返回值，2.改变堆栈的状态．最好将这两个独立的动作放到两个独立的函数中,遵守内聚设计的原则，每一个函数只做一件事．我们 再来讨论另外一个异常安全的问题，就是很常见的赋值操作符的写法，如何保证赋值操作是异常安全的．

class Bitmap {...};

class Widget {

...

private:

Bitmap \*pb;

};

Widget& Widget::operator=(const Widget& rhs)

{

delete pb;

pb = new Bitmap(\*rhs.pb);

return \*this;

}

上面的代码不具备自我赋值安全性，倘若rhs就是对象本身，那么将会导致\*rhs.pb指向一个被删除了的对象．那么就绪改进下．加入证同性测试．

Widget& Widget::operator=(const Widget& rhs)

{

If(this == rhs) return \*this; //证同性测试

delete pb;

pb = new Bitmap(\*rhs.pb);

return \*this;

}

但是现在上面的代码依旧不符合异常安全性，因为如果delete pb执行完成后在执行new Bitmap的时候出现了异常，则会导致最终指向一块被删除的内存．现在只要稍微改变一下，就可以让上面的代码具备异常安全性．

Widget& Widget::operator=(const Widget& rhs)

{

If(this == rhs) return \*this; //证同性测试

Bitmap \*pOrig = pb;

pb = new Bitmap(\*rhs.pb); //现在这里即使发生了异常，也不会影响this指向的对象

delete pOrig;

return \*this;

}

这个例子看起来还是比较简单的，但是用处还是很大的，对于赋值操作符来说，很多情况都是需要重载的．