# 13异常处理

异常是指被程序抛出的一个对象，他表示在程序中出现了一个错误。异常可以是任意类型的对象，只要它能被拷贝即可。（p299）

异常可以携带一些有关错误的描述信息。异常的类型表示错误种类，异常携带的数据记录了错误的出现时的情形。例如，在标准库异常中含有一个字符串，它可以告诉我们抛出异常的位置。

1. 传统的错误处理（p299）

除了使用异常机制处理错误，其他几种传统的处理方式都有各自的不足：

* 终止程序。这是一种非常极端的处理方式，例如：

if(something\_wrong) exit(1);

对于绝大多数错误来说，我们有能力也必须处理的更好。例如，在大多数情况下，我们在终止程序之前，至少应该给出一条比较准确的错误信息或者把该错误记录下来。尤其是，如果不清楚它所处的程序的目的和作用，就不应该简单地调用exit()或者abort()。如果一个程序不允许轻易崩溃，显然在其中不能使用任何采用无条件终止的库。

* 返回错误值。这种策略也并非百试不爽，因为有时候我们根本就得不到适合的错误值。例如：

int get\_int(); //从输入中获得下一个整数

对上面这个执行输入操作的函数来说，每一个int值都可能是他的结果，因此我们无法指定其中某个值作为输入错误的标示值。一种解决方案是修改get\_int()令其返回一对值。然而即使这么做可以应付一些情况，看起来也不怎么方便。以后每次调用该函数都必须检查一下它返回的错误值，无形中增加了程序的工作量，而且也会使程序的规模翻倍。另外，函数的调用者常常会忽略可能出现的错误或者忘记检验返回值。总之，这种策略不足以系统地检查并处理全部错误。

* 返回合法值，而程序却处于“错误状态”。问题是主调函数可能没有意识到程序已经处于错误状态了。
* 调用错误处理函数。例如：

if(something\_wrong) something\_handler(); //问题并未得到解决，只是暂时转移了

这种策略其实只是前几种处理措施的变形。我们很容易继续发问：“那么错误处理函数应该干什么呢？”除非这里的错误处理函数能够完整地解决所有问题，否则他还是会陷入之前的境地当中：终止程序、返回一个标志发生了错误的值、设置错误状态或者抛出异常。

1. （p303-304）偶尔也需要从一种错误报告模式转换成另一种，例如，在调用C语言库抛出异常；反之，也能再从C++库返回C库程序前捕获异常并设置errno：

void callC() //在C++中调用C函数，吧errno转换成throw

{

errno = 0;

c\_function();

if(errno)

{

//…如果可且必要的话，执行局部清理…

throw C\_blewit(errno);

}

}

extern “C” void call\_plus\_function() noexcept //在C语言中调用一个C++函数，

//把throw转换成errno

{

try

{

c\_plus\_plus\_function();

}

catch(…)

{

//…如果可能且必要的话，执行局部清理…

errno = E\_CPLPLFCTBLEWIT;

}

}

1. 异常与效率（p304-305）

原则上讲，我们可以做到当不抛出异常时就不会产生异常处理的开销。而且，这么做也能尽量让抛出异常的代价不一定像调用函数那么高。总的来说，有可能在不显著增加内存负担的前提下保持与C语言调用序列、调试器规则等的互通性，但是比较难。

bool g(int);

bool h(const char\*);

char\* read\_long\_string();

bool f()

{

char\* s = read\_long\_string();

//…

if(g(1))

{

if(h(s))

{

free(s);

return true;

}

else

{

free(s);

return false;

}

}

else

{

free(s);

return false;

}

}

使用局部缓冲替换s可以省去调用free()的代码,但是我们必须自己执行边界检查，这样做的复杂性更高。

noexcept说明符对于改善生成的代码很有帮助。举个例子：

void g(int) noexcept;

void h(const string&) noexcept;

这样，f()生成的代码就有可能改进了。

1. 异常保障（p305）

如果在通过抛出异常中止某个操作后，程序仍然处于有效状态，则成这个操作是异常安全的操作。

基本保障和强保障都不允许资源泄露，这对于无法承受资源泄露的系统来说是非常必要的。尤其是，一个抛出异常的操作仅仅确保运行对象处于定义良好的状态还不够，它必须释放掉之前申请的全部资源。例如在异常抛出点，所有分配的内存只能处于两种状态：要么已经释放掉，要么属于某个对象并且保证将来被正确地释放掉。例如：

void f(int i)

{

int \* p = new int[10];

//…

if(i<0)

{

delete []p; //在抛出异常前释放掉，否则将泄露

throw Bad();

}

//…

}

谨记内存不是唯一一种可能泄露的资源。我们通常把从系统某处申请并且将（显示地或者隐式地）归还回去的东西统称为资源。文件、锁、网络连接和线程都是资源。函数在抛出异常前必须释放这些资源或者把它们移交给其他资源句柄。

1. 资源管理（p307）

当函数请求某种资源是，也就是说，当它打开文件、从自由存储分配一些内容或者和请求一个互斥锁时，系统常常要求这些资源能在未来时刻被正确地释放掉。所谓“正确地释放掉”是指函数应该在返回它的调用者之前释放掉它请求的资源。例如：

void use\_file(const char\* fn) //貌似正确的代码

{

FILE\* f = fopen(fn, “r”);

//…使用f…

fclose(f);

}

这段代码看起来没什么问题，实则暗藏风险。假设在调用fopen()之后尚未调用flose()的某个时刻程序出错了，异常将导致use\_file()直接退出，并且再也不会执行fclose()。在不支持异常处理的编程语言中，这种情况是由发生。例如，C语言标准库函数longjmp()就会造成上述问题。一条普通的return语句也可能造成在未关闭f的情况下就退出use\_file()。

（p308）通常情况下，释放资源的顺序应该与请求资源的顺序相反，这非常类似于构造函数创建对象以及析构函数销毁对象的行为，因此，我们可以用含有构造函数和析构函数的类的对象来处理请求和释放资源的问题。例如，我们定义File\_ptr，他的行为与FILE\*类似：

class File\_ptr

{

FILE\* p;

public:

File\_ptr(const char\* n, const char\* a) //打开文件n

:p{fopen(n, a)}

{

if(p == nullptr) throw runtime\_error{“File\_ptr:Can’t open file”};

}

File\_ptr(const string& n, const char\* a) //打开文件n

: File\_ptr{n.c\_str(), a}

{}

explicit File\_ptr(FILE\* pp) //假定pp的所有权

:p{pp}

{

if(p == nullptr) throw runtime\_error{“File\_ptr:nullptr”};

}

//…适当的移动和拷贝操作…

~File\_ptr()

{

fclose(p);

}

operator FILE\*()

{

return p;

}

}

我们可以用FILE\*构造File\_ptr，也可以提供给fopen()的参数构造File\_ptr。不管怎么样，File\_ptr对象都将在它的作用域末尾被销毁，并由它析构函数负责关闭文件。如果File\_ptr无法打开文件，他会抛出一个异常，这样就无需每次使用该文件句柄都检测是否是nullptr了。我们的函数简化为如下形式：

void use\_file(const char\* fn)

{

File\_ptr f(fn, “r”);

//…使用f…

}

不论函数是正常退出还是因为抛出异常退出，系统都会调用析构函数。也就是说，异常处理机制使得我们可以把错误处理代码从核心算法中独立出来。剩下的代码得到了简化，与原来的形式相比出错的概率也降低了。

这种使用局部对象管理资源的技术通常称为“资源获取及初始化”(RAII)。这是一种比较通用的技术，其前提是具备了构造函数和析构函数的属性，并且这两个函数与错误处理机制可以有机地融合在一起。

对于一个对象来说，只有当它的构造函数完成了，我们才认为该对象创建成功了。之后，栈展开会为对象调用析构函数。如果对象由若干个子对象组成，则先构造每个字对象，再构造对象本身；数组的情况与之类似，先构造每个函数，再构造数组（在展开时只销毁完整构造的元素）。

假设有一个类X，它的构造函数负责请求两种资源：文件x和互斥量y。这些请求有可能会失败并抛出异常。对于X的构造函数来说，当它结束的时候，既不能只请求文件而未请求互斥量，也不允许之请求互斥量而未请求文件。另外，当在实现上述目标的同时还应该避免增加程序的编程负担。我们用File\_ptr和std::unique\_lock这两个类的对象表示请求到的资源。此时，我们可以吧请求某项资源的的工作转换成初始化表示资源的局部变量对象：

class Locked\_file\_handle

{

File\_ptr p;

unique\_lock<mutex> lck;

public:

X(const char\* file, mutex& m)

: p{file, “r”}

, lck{m}

{}

//…

}

就像处理局部对象时一样，由系统负责记录资源的来龙去脉，用户无需为此烦恼。例如，如果在构造p但是尚未构造lck的时刻发生了异常，则程序将调用p的析构函数，但是不会调用lck的析构函数。

这意味着只要我们使用了类似地资源请求模型，构造函数的作者就不用显示地编写异常处理代码了。

内存是最常用的资源，string、vector以及其他标准容器使用RAII隐式地管理内存的请求和释放。与使用new和delete管理内存方式的方式相比，前者不但可以节约大量编程工作，还能避免很多错误。

当涉及对象的指针而非局部对象时，我们使用unique\_ptr和shared\_ptr避免内存泄露。

1. finally（p310）

在之前的介绍中，我们把资源表示为一个包含析构函数的类的对象，这种做法可能会带来一些困扰。为了编写任意代码以在异常发生后执行清理工作，人们曾经设计了很多“最终的”语言概念。这些技术通常只能用于特定的场景，因此RAII相比并不占优势，但如果确实需要的话，RAII也可以支持此类技术。首先定义一个类，他在析构函数中执行任意操作。

template<typename F>

struct Final\_action

{

Fianl\_action(F f):clean {f} {}

~Final\_action(){ clean(); }

F clean;

};

我们通过构造函数的参数提供“最终操作”。

接下来定义一个函数，它可以方便地推断某个操作的类型：

template<class F>

Final\_action<F> finally(F f)

{

return Final\_action<F>(f);

}

最后检验finally()的效果：

void text()

//处理非常规的自愿请求任务

//该代码验证我们可以在其中嵌入任意操作

{

int\* p = new int{7}; //其实应该使用unique\_ptr

int buf = (int\*)malloc(100\*sizeof(int)); //C风格的资源请求

auto act1 = finally([&]{ delete p; free(buf); cout<<”Goodby, Cruel world!\n”; });

int var = 0;

cout<<”var = ”>>var<<’\n’;

//嵌套的块

{

var = 1;

auto act2 = finally([&]{cout<<”finally!\n”; var = 7;});

cout<<”var = ”>>var<<’\n’;

} //调用act2

}

上述代码输出：

var = 0

var = 1

finally!

var = 7

Goodby, Cruel world!

此外，p和buf请求和指向的内存区域也没正确地删除（delete）和清空（free()）掉了。

基本上，finally()对于块的作用域for语句椎间盘买个递增部分的作用类似：它在块一开始的地方就自定了最终要执行的操作，这么做不仅容易于程序员阅读，而且从说明的角度来看它本来就应该在这合理。它告诉程序当前作用域结束时应该做什么，这样程序员就不用事事留意通知线程可能在如何退出，并且在每处都编写对应的代码了。

1. （p312）在很多技术可用于检查预置的条件和不变式。如果我们希望检查的原因保持中立，则常使用断言（assertion，简写为assert）。断言是一个逻辑表达式，我们假定断言的值为true。然而断言绝不仅仅是一条注释，我们还需要注明一旦它的值为false时应该做什么。显然在大量系统中，对于断言有各种各样的需求：

* 我们需要在编译时断言（由编译器求值）和运算时断言（在运行时求值）中做出选择。
* 对于运行时断言我们需要选择处理的方式：抛出异常、终止程序韩式直接忽略。
* 除非某些逻辑条件为true，否则不应该生成代码。例如除非逻辑条件为ture，否则不应该对某些运行时断言求值。通常情况下，所谓逻辑条件是指某些类似于调试标志、检查级别或者断言选择范围之类的东西。
* 断言应该易于编写（因为断言常常觉有通用性，会用在很多地方）。

不是每个系统都有上面的全部需求，淡然也无需处理每种需求。

C++标准提供两种简单地机制：

* 在<cassert>中，标注库提供了assert(A)宏。当且仅当未定义宏NDEBUG（非调试）时，它在运行时检查断言A，一旦断言失败，编译器将输出一条错误信息并终止程序。其中，输出的错误信息包含错误的断言、源文件的名字以及行号等。
* C++断言使用static\_assert(A, message)在编译时无条件检查断言A。一旦断言失败，编译器将输出message以及编译错误信息。

如果在某些情况下assert()和static\_assert()不适用，我们也可以使用普通的代码进行检查。例如：

void f(int n)

//n应该在[1:max]之间

{

if(2<debug\_level && (n<=0 || max<n))

throw Assert\_error(“rang problem”);

//…

}

然而，使用这样的普通代码会使得要检查的对象不太明显。我们在：

* 对我们的测试条件求值？（是的，在2<debug\_level的部分。）
* 对某些条件求值么？我们希望该条件对某些条用为真，对其他调试不为真？（不，因为我们抛出了异常。除非有人愿意把异常当成一种返回机制，否则答案是否定的；见13.1.4.2节。）
* 检查一个应该永远成立的前置条件吗？（是，此处的异常仅仅是我们的回应而已。）

更糟糕的是，检查前置条件（或者不变式）的代码经常散落在其他代码中，既难定位又容易出错。我们想要的是一种检查断言的易于识别的机制，接下来我们就介绍这种机制。它可能显得有一点繁琐，但是它既能表示大量断言，又能涵盖很多对于断言失败的响应。首先，我定义一些机制，它负责决定何时检查以及一旦失败应该做什么：

namespace Assert

{

enum class Mode{ throw\_, terminate\_, ignore\_};

constexpr Mode current\_mode = CURRENT\_MODE;

constexpr int current\_level = CURRENT\_LEVEL;

constexpr int default\_level = 1;

constexpr bool level(int n)

{

return n<=current\_level;

}

struct Error : runtime\_error

{

Error(const string& p) : runtime\_error(p) {}

};

//…

}

上述代码的主要目的是检查断言的“层级”在何时不高于current\_level。如果断言失败，则用current\_mode在三种响应模式中选择一种。我们的目的是在决定做什么之前对任何断言不生成代码，因此current\_level和current\_mode被设置成常量。CURRENT\_MODE和CURRENT\_LEVEL可以看成是在程序的编译环境中设置的编译选项。

程序员使用Assert::dynamic()设置断言：

namespace Assert

{

//…

string compose(const char\* file, int line, const string& message)

//混合生成包含文件名和行号的消息

{

ostringstream os(“(”);

os << file << “,” << line << “):” << message;

return os.str();

}

template<bool condition = level(default\_level), class Except = Error>

void dynamic(bool assertion, const string& message = “Assert::dynamic failed”)

{

if(assertion)

{

return;

}

if(current\_mode == Assert\_mode::throw\_)

{

throw Except{message};

}

if(current\_mode == Assert::terminate\_)

{

std::terminate();

}

}

template<>

void dynamic<false, Error>(bool, const string&) //什么也不做

{

}

void dynamic(bool b, const string& s) //默认操作

{

dynamic<true, Error>(b, s);

}

void dynamic(bool b) //默认消息

{

dynamic<true, Error>(b);

}

}

我选用Assert::dynamic这个名字（意思是“运行时求值”）来与static\_assert作为对比（意思是“编译时求值”）。

我们还可以通过其他一些实现技巧来使生成的代码量最少。如果对灵活性要求高的话，也可以在运行时做更多测试工作。上面这个Assert并不是标准的一部分，我把它列在此处主要是为了讲解有关的问题和技术。要想实现一种放之四海而皆准的断言机制，他需要满足的要求会非常非常多。

Assert::dynamic的用法如下：

void f(int n) //n应该在[1 : max]之间

{

Assert::dynamic<Assert::level(2), Assert::Error>(

(n<=0 || max<n), Assert::compose(\_\_FILE\_\_, \_\_LINE\_\_, “range problem”);

)

//…

}

其中，\_\_FILE\_\_和\_\_LINE\_\_是宏，它们会在源代码的对应位置展开（见12.6.2节）。我无法把它们放置在Assert的实现中以对用户隐藏其细节。

Assert::Error是默认的异常，因此我们无需显示地提及它。类似地，如果我们想使用磨人的断言级别，也不需要显示地指出来：

void f(int n) //n应该在[1 : max]之间

{

Assert::dynamic((n<=0 || max<n), Assert::compose(\_\_FILE\_\_, \_\_LINE\_\_, “range problem”));

//…

}

我不认为过分纠结于表示断言的文本长度会有什么帮助，但是通过使用名字空间指示（见14.2.3）和默认消息，我们可以令文本长度最小化：

void f(int n) //n应该在[1 : max]之间

{

dynamic((n<=0 || max<n);

//…

}

我们可以通过构件选项（例如，控制条件编译）或程序代码选项控制要执行的测试以及对测试的响应。因此你既可以得到一个执行全部测试的程序版本以便对它进行测试，也可以得到一个几乎不做任何测试的产品级版本。

我个人的习惯是在程序的最终发布版中至少留一些测试程序。例如，保留Assert的作用之一是所有标记为0级的断言会被检查。字持续开发和维护一个大型程序的过程中，我们很难穷尽全部漏洞。而且，即使所有的事情都做的尽善尽美了，保留一些“清醒的检查”也有助于处理可能发生的硬件错误。

1. 抛出异常

我们可以throw任意类型的异常，前提是它能被复制和移动。例如：

class No\_copy

{

No\_cpoy(const No\_cpoy&) = delete; //禁止复制（见17.6.4节）

};

class My\_error

{

//…

};

void f(int n)

{

switch(n)

{

case 0 : throw My\_error{}; //OK

case 1 : throw No\_copy{}; //错误，不允许复制No\_copy

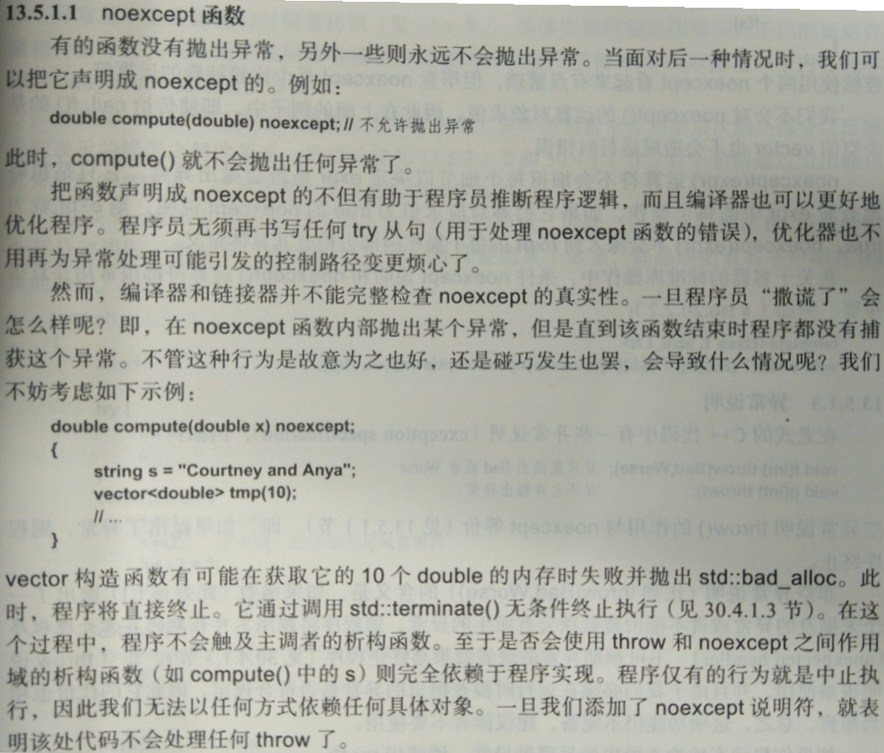
case 2 : throw My\_error; //错误，My\_error是一种类型，而非一个对象

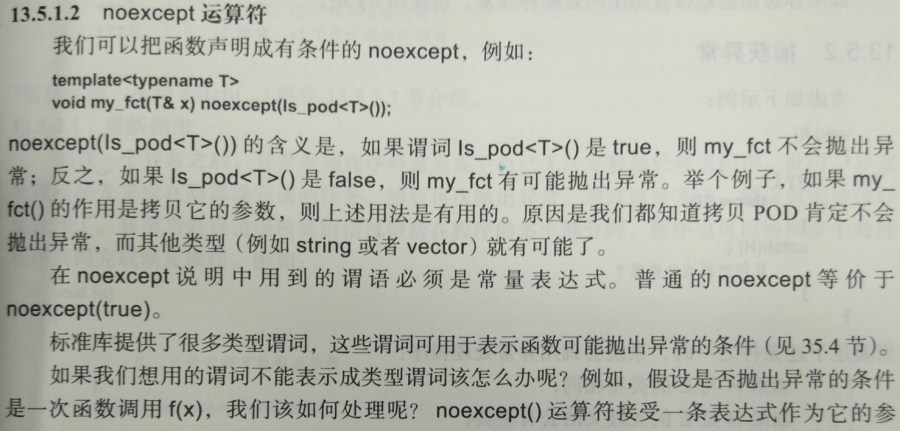
}

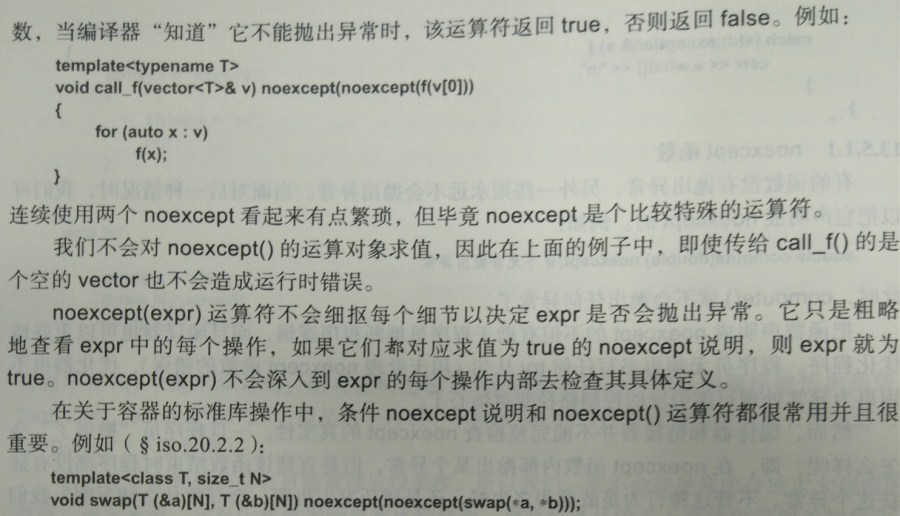
}

捕获的对象本质来说就是被抛出的对象的一分拷贝（尽管我们允许优化器最小化拷贝过程），换句话说，throw x用x初始化了一个x的类型的临时变量。

1. （p316）runtime\_errorhe out\_of\_range等标准库异常类接受一个字符串作为其构造函数的参数，然后用虚函数what()把该字符串的内容用在别处。
2. noexcept函数







1. 异常说明

在老式的C++代码中有些异常说明（exception specification），例如：

void f(int) throw(Bad, Worse); //只能抛出Bad或者Worse

void g(int) throw(); //不允许抛出异常

空异常说明throw()的作用与noexcpet等价。即，如果抛出异常，则程序终止。

非空异常说明（比如throw(Bad, Worse)）的含义是，如果函数（此处是f()）抛出了一个未提及的异常或者不能由参数项公有派生的异常，则程序将调用不可预测的异常处理程序（unecpected handler）。不可预期的缺省效果是终止程序。非空throw说明很难使用，并且由于我们必须在运行时检查抛出的异常是否符合规定，因此它的代价也相当昂贵。总之，这项功能仍不完备，建议读者不要使用。

如果你想动态地检查抛出的是哪种异常，请使用try块。

1. 异常捕获

考虑如下示例：

void f()

{

try

{

throw E{};

}

catch(H)

{

//何时到达此处呢？

}

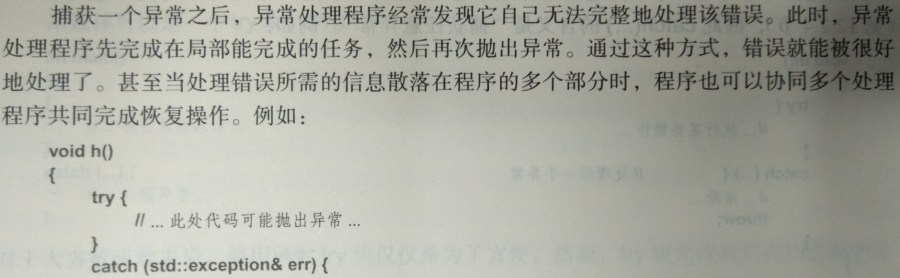
}

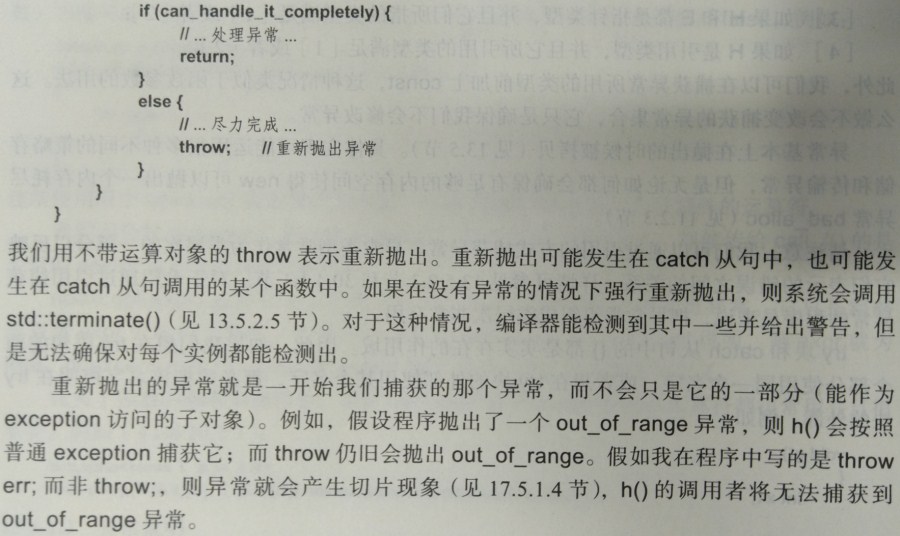
当时满足下述条件之一时，系统会调用异常处理程序：

1. 如果H与E的类型相同；
2. 如果H与E的无歧义的公有基类
3. 如果H和E都是指针类型，并且他们所指的类型满足[1]和[2]；
4. 如果H是应用类型，并且它所指向的引用的类型满足[1]和[2]；

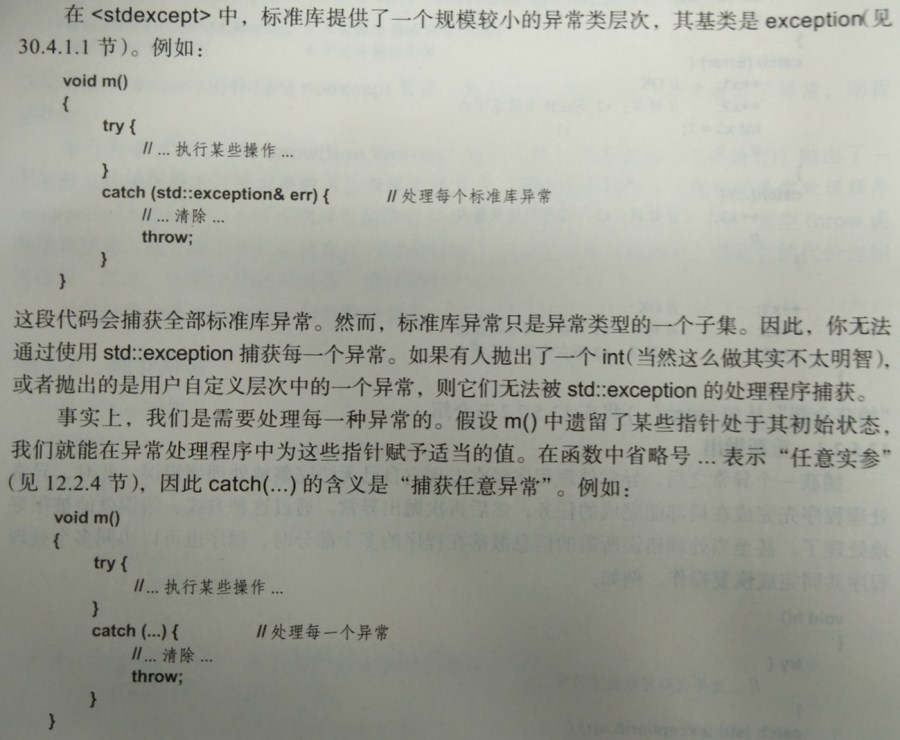
此外，我们可以在捕获异常所在的类型前加上const，这种情况类似于函数参数的用法。这麼做不会改变捕获的异常集合，他只是确保我们不会修改异常。

1. 重新抛出（p319）





1. 捕获每个异常



1. 多异常处理程序（p321）

一个try块可以对应多个catch从句（异常处理程序）。因此派生的异常能被多种异常类型的处理程序捕获，所有try语句中异常处理程序的书写顺序显得非常重要。程序将一次尝试每段处理代码。例如：

void f()

{

try

{

//…

}

catch(std::ios\_base::failure)

{

//…处理各种输入输出错误（见30.4.1.1节）…

}

catch(std::exception& e)

{

//…处理各种标准库异常（见30.4.1.1节）…

}

catch(…)

{

//…处理其他异常（见13.5.2.2）…

}

}

编译器了解层次的情况，因此它可以发现并报告很多逻辑错误。例如：

void g()

{

try

{

//…

}

catch(…)

{

//…处理各种错误（见30.4.2.2节）…

}

catch(std::exception& e)

{

//…处理各种标准库异常（见30.4.1.1节）…

}

catch(std::bad\_cast)

{

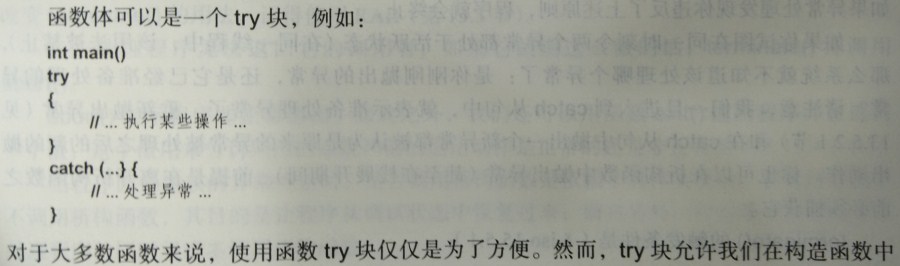
//…处理动态类型转换异常（见22.2.1）…

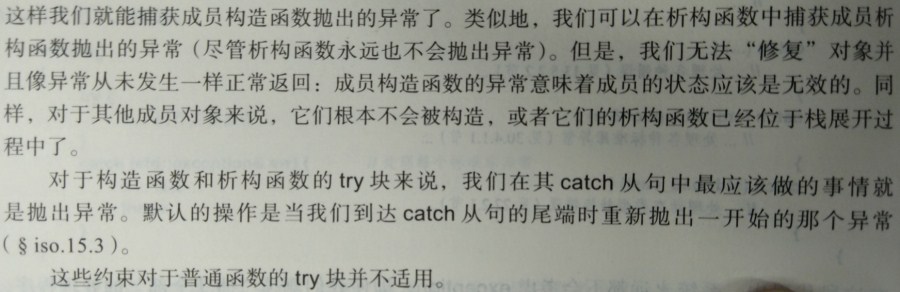
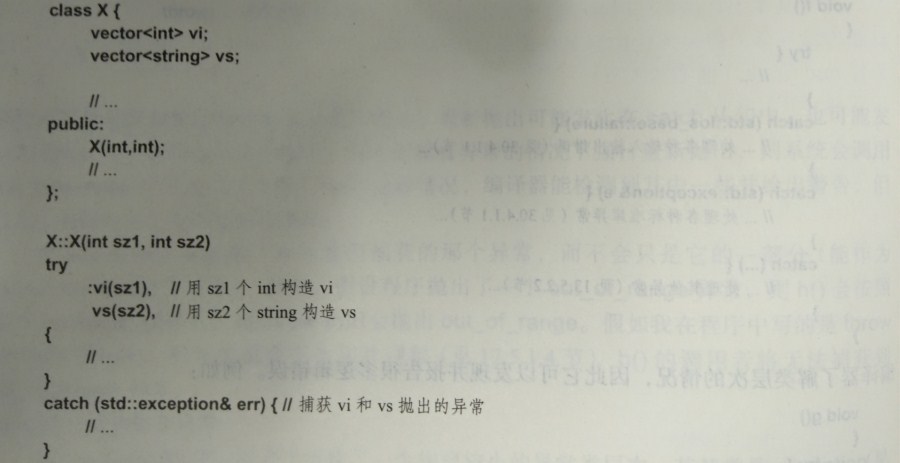
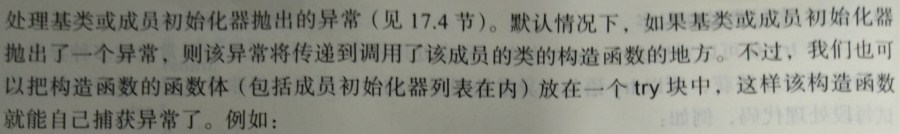
}

}

在这段代码中，系统永远不会考虑exception。即使我们删掉“捕获全部”的处理程序，程序也仍然有错，因为bad\_cast是从exception中派生来的，所以它永远不会执行。异常类型与catch从句的匹配过程是一种快速的运行时操作，其机理与编译时的重载解析并不一样。

1. 函数try块





1. 终止（p322）

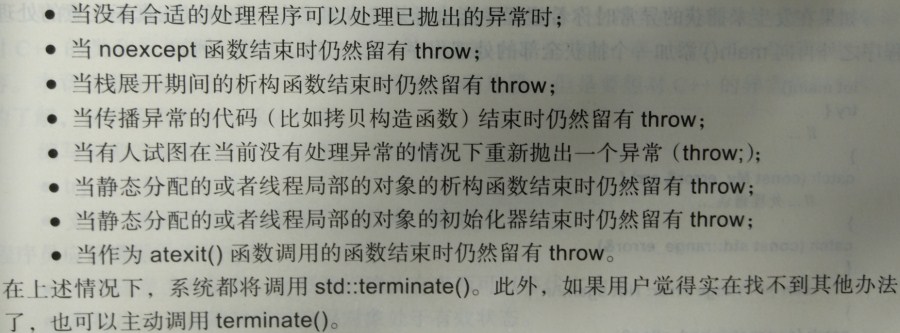
在有些情况下最好不要使用异常处理，基本的原则是：

* 处理异常的时候不要抛出异常。
* 不要抛出一个无法捕获的异常。

如果异常处理发现一违反了上述原则，程序就会终止。

如果你试图在同一时刻令两个异常都处于活跃状态（在同一线程中，该用法被禁止），那么系统就不知道该处理哪个异常了：是你刚刚抛出的异常还是它已经准备处理的异常？请注意，我们一旦进入到catch从句中，就表示准备处理异常了。重新抛出异常（见13.5.2.1节）和在catch从句中抛出一个新异常常被认为是原来的异常被处理之后的新的抛出动作。你也可以在析构函数中抛出异常（甚至在栈展开的期间），前提是在离开析构函数之前必须捕获它。

terminate()的触发条件是（§iso.15.5.1）:



“结束时留有throw”的意思是在某处抛出异常，但是该异常未被捕获，因而运行时系统试图把该异常从函数传递给函数调用者。

默认情况下，terminate()会调用abort()（见15.4.3节）。对大多数用户来说，这个默认选项也是最佳选择，特别在调试期间更是如此。如果用户不接受该选项，也可以通过调用<exception>中的std::set\_terminate()提供一个终止处理程序（terminate handler）：

using terminate\_handler = void(\*)(); //源于<exception>

[[noreturn]] void my\_handler() //终止处理程序无法返回任何值

{

//自行处理终止

}

void dangerous() //非常危险！

{

terminate\_handler old = set\_terminate(my\_handler);

//…

set\_terminate(old); //修复旧的终止处理程序

}

返回值是提供给set\_terminate()的函数。

举个例子，终止处理程序可用于中断一个处理过程或者重新初始化一个系统。terminate()的出发点是当异常处理机制实现错误恢复策略时，必须采用跟积极的措施；是时候切换到新的容错策略了。一旦进入了终止处理程序，任何关于程序数据结构的假设 都不成立，我们无法保证他们不被干扰和损坏。甚至用cerr输出错误消息都可能出错。同样，就像上面的的dangerous()说的那样，它并非异常安全的。set\_terminate(old)前面的一个throw甚至是一个return都有可能把my\_handler置于意料之外的境地。如果你一定要改变terminate()的用法，记得使用RAII（见13.3节）。

终止处理无法返回它的调用者，如果他试图这么做的话，terminate()将调用abort()。