C 与 C++中的异常处理

Robert Schmidt 著
无情 译

目 录

1. 异常和标准 C 对它的支持

（前言略）

1.1 异常分类

基于 Dr. GUI 的建议，我把我的第一个专栏投入到“程序异常”的系列上。我认识到，“exception”这个术语有些不明确并和上下文相关，尤其是 C++标准异常（C++ standard exceptions）和 Microsoft 的结构化异常（structured exception handling）。不幸的的是，“异常”一词太常见了，随时出现在语言的标准和常见的编程文献中。因为不想创造一个新名词，所以我将尽力在此系列的各部分中明确我对“异常”的用法。

- Part 1 概述通常意义上的异常的性质，和标准 C 库提供的处理它们的方法。
- Part 2 纵览 Microsoft 对这些标准 C 库方法的扩展：专门的宏和结构化异常处理。
- Part 3 及其余将致力于标准 C++异常处理体系。

（C 语言使用者可能在 Part2 后放弃，但我鼓励你坚持到底；我所提出的许多点子同样适用于 C，虽然不是很直接。）

本质上看，程序异常是指出现了一些很少发生的或出乎意料的状态，通常显示了一个程序错误或要求一个必须提供的回应。不能满足这个回应经常造成程序功能削弱或死亡，有时导致整个系统和它一起 down 掉。不幸的是，试图使用传统的防护方法来编制健壮的代码经常只是将一个问题（意外崩溃）换成了另外一个问题（更混乱的设计和代码）。

太多的程序员认为这个交换抵不上程序意外崩溃时造成的烦恼，于是选择了生活在危险之中。认识到这一点后，C++标准增加了一个优雅并且基本上不可见的“异常体系”到语言中；就这样，这个方法产生了。如同我们在 Part4 的开始部分将要看到的，这个方法大部分情况下很成功，但在很微妙的情况下可能失败。

1.2 异常的生命阶段

在这个系列里，我将展示 C 和 C++处理异常体系运行于异常整个生命期的每一阶段时的不同之处：

- 阶段 1：一个软件错误发生。这个错误也许产生于一个被底层驱动或内核映射为软件错误的硬件响应事件（如被 0 除）。
- 阶段 2：错误的原因和性质被一个异常对象携带。这个对象的类型可以简单的整数值到繁杂的 C++类对象。
- 阶段 3：你的程序必须检测这个异常对象：或者轮询它的存在，或者由其主动上报。
- 阶段 4：检测代码必须决定如何处理异常。典型的方法分成三类。
  a 忽略异常对象，并期望别人处理它。
  b 在这个对象上干些什么，并还允许别人再继续处理它。
  c 获得异常的全部所有权。
- 阶段 5：既然异常已经处理了，程序通常恢复并继续执行。恢复分成两种：
  a 恢复异常，从异常发生处继续执行。
  b 终止异常，从异常被处理处继续执行。

当在程序外面（由运行期库或操作系统）终止异常时，恢复经常是不可能的，程序将异常结束。

我故意忽略了硬件错误事件，因为它们完全是底层平台范围内的事。取而代之，我假定一些软件上的可检测错误已经发生，并产生了一个处于第一阶段的软件异常对象。

1.3 C 标准库异常处理体系

C 标准库提供了几个方法来处理异常。它们也全部在标准 C++中有效，只是相关的头文件名字变了：老的 C 标准头文件<name.h>映射到了新的 C++标准头文件<cname>。（头文件名的前缀“C”是个助记符，暗示着这些全是 C 库头文件。）

虽然基于向后兼容性，老的 C 头文件也被 C++保留，但我建议你尽可能使用新的头文件。对于绝大部分实际使用而言，最大的变化是在新的头文件中，申明的函数被包含在命名空间 std 内。举个例子，C 语言使用

```
#include <stdio.h>

FILE *f = fopen("blarney.txt", "r");
```

在 C++中被改成

```
#include <cstdio>

std::FILE *f = std::fopen("blarney.txt", "r");
```

或更 C 风格的

```
#include <cstdio>
using namespace std;

FILE *f = fopen("blarney.txt", "r");
```

不幸的是，Microsoft 的 Visual C++没有将这些新的头文件包含在命名空间 std 中，虽然这是 C++标准所要求的（subclause D.5）。除非 Visual C++在这些头文件中已经正确地支持了 std，我将一直在我的专栏中使用老式的 C 风格命名。

（象 MIcrosoft 这样的运行库卖主这么做是合理的，正确地实现这些 C 程序库的头文件极可能要求维护和测试两份完全不同的底层代码，这是不可能受欢迎的也不值得多花力气的工作。）

1.4 无条件终止

仅次于彻底忽略一个异常，大概最容易的异常处理方法是程序自我毁灭。有时，最懒的方法事实上是最正确的。

在你开始嘲笑以前，应该认识到，一些异常表示的状况是如此严重以致于怎么也不可能合理恢复的。也许最好的例子就是 malloc 时返回 NULL。如果空闲堆管理程序不能提供可用的连续空间，你程序的健壮性将严重受损，并且恢复的可能性是渺茫的。

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。

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。

一般调用 abort()处理灾难性的程序故障。因为 abort()的默认行为是立即终止程序，你就必须负责在调用 abort()前存储重要数据。（当我们谈论到<signal.h>时，你可以使得 abort()自动调用 clean up 代码。）

相反，exit()执行了挂接在 atexit()上的自定义 clean up 代码。这些代码被按照其挂接的反序执行，你可以把它们当作虚拟析构器。通过必要的 clean up 代码，你可以安全地终止程序而没有留下尾巴。例如：

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

static void atexit_handler_1(void)
   {
   printf("within 'atexit_handler_1'\n");
   }

static void atexit_handler_2(void)
   {
   printf("within 'atexit_handler_2'\n");
   }

int main(void)
   {
   atexit(atexit_handler_1);
   atexit(atexit_handler_2);
   exit(EXIT_SUCCESS);
   printf("this line should never appear\n");
   return 0;
   }

/* When run yields
      within 'atexit_handler_2'
      within 'atexit_handler_1'

   and returns a success code to calling environment.
*/
```

（注意，即使是程序从 main()正常返回而没有明确调用 exit()，所挂接的 atexit()代码仍然会被调用。）

无论 abort()还是 exit()都不会返回到它的调用者中，且都将导致程序结束。在这个意

义上来说，它们都表现为终止异常的最后一步。

1.5 有条件地终止

abort()和 exit()让你无条件终止程序。你还可以有条件地终止程序。其实现体系是每个程序员所喜爱的诊断工具：断言，定义于<assert.h>。这个宏的典型实现如下所示：

```
#if defined NDEBUG
   #define assert(condition) ((void) 0)
#else
   #define assert(condition) \
       _assert((condition), #condition, __FILE__, __LINE__)
#endif
```

如定义体所示，当宏 NDEBUG 被定义时断言是无行为的，这暗示了它只对调试版本有效。于是，断言条件从不在非调试版本中被求值，这会造成同样的代码在调试和非调试版本间有奇妙的差异。

```
/* debug version */
#undef NDEBUG
#include <assert.h>
#include <stdio.h>

int main(void)
   {
   int i = 0;
   assert(++i != 0);
   printf("i is %d\n", i);
   return 0;
   }
/* When run yields

      i is 1
*/
```

现在，通过定义 NDEBUG，从 debug 版变到 release 版：

```
/* release version */
#defing NDEBUG
#include <assert.h>
#include <stdio.h>

int main(void)
   {
   int i = 0;
   assert(++i != 0);
   printf("i is %d\n", i);
   return 0;
```

```
   }

/* When run yields

      i is 0
*/
```

要避免这个差异，必须确保断言表达式的求值不会包含有影响的副作用。

在仅供调试版使用的定义体中，断言变成呼叫_assert()函数。我起了这个名字，而你所用的运行库的实现可以调用任何它想调用的内部函数。无论它叫什么，这个函数通常有以下形式：

```
void _assert(int test, char const *test_image,
   char const *file, int line)
   {
   if (!test)
      {
      printf("Assertion failed: %s, file %s, line %d\n",
      test_image, file, line);
      abort();
      }
   }
```

所以，失败的断言在调用 abort()前显示出失败情况的诊断条件、出错的源文件名称和行号。我在这里演示的诊断机构“printf()”相当粗糙，你所用的运行库的实现可能产生更多的反馈信息。

断言处理了异常的阶段 3 到 5。它们实际上是一个带说明信息的 abort()并做了前提条件检查，如果检查失败，程序中止。一般使用断言调试逻辑错误和绝不可能出现在正确的程序中的情况。

```
/* 'f' never called by other programs */
static void f(int *p)
   {
   assert(p != NULL);
   /* ... */
   }
```

对比一下逻辑错误和可以存在于正确程序中的运行期错误：

```
/* ...get file 'name' from user... */
FILE *file = fopen(name, mode);
assert(file != NULL); /* questionable use */
```

这样的错误表示异常情况，但不是 bug。对这些运行期异常，断言大概不是个合适的处理方法，你应该用我下面将介绍的另一个体系来代替。

1.6 非局部的跳转

与刺激的 abort()和 exit()相比,goto 语句看起来是处理异常的更可行方案。不幸的是，

goto 是本地的：它只能跳到所在函数内部的标号上，而不能将控制权转移到所在程序的任意地点（当然，除非你的所有代码都在 main 体中）。

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。

原理非常简单：

- setjmp(j)设置“jump”点，用正确的程序上下文填充 jmp_buf 对象 j。这个上下文包括程序存放位置、栈和框架指针，其它重要的寄存器和内存数据。当初始化完 jump 的上下文，setjmp()返回 0 值。
- 以后调用 longjmp(j,r)的效果就是一个非局部的 goto 或“长跳转”到由 j 描述的上下文处(也就是到那原来设置 j 的 setjmp()处)。当作为长跳转的目标而被调用时，setjmp()返回 r 或 1（如果 r 设为 0 的话)。(记住，setjmp()不能在这种情况时返回 0。)

通过有两类返回值，setjmp()让你知道它正在被怎么使用。当设置 j 时，setjmp()如你期望地执行；但当作为长跳转的目标时，setjmp()就从外面“唤醒”它的上下文。你可以用 longjmp()来终止异常，用 setjmp()标记相应的异常处理程序。

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

jmp_buf j;
void raise_exception(void)
   {
   printf("exception raised\n");
   longjmp(j, 1); /* jump to exception handler */
   printf("this line should never appear\n");
   }

int main(void)
   {
   if (setjmp(j) == 0)
      {
      printf("'setjmp' is initializing 'j'\n");
      raise_exception();
      printf("this line should never appear\n");
      }
   else
      {
      printf("'setjmp' was just jumped into\n");
      /* this code is the exception handler */
      }
   return 0;
   }

/* When run yields:
```

```
   'setjmp' is initializing 'j'
   exception raised
   'setjmp' was just jumped into
*/
```

那个填充 jmp_buf 的函数不在调用 longjmp()之前返回。否则，存储在 jmp_buf 中的上下文就有问题了：

```
jmp_buf j;

void f(void)
   {
   setjmp(j);
   }

int main(void)
   {
   f();
   longjmp(j, 1); /* logic error */
   return 0;
   }
```

所以，你必须把 setjmp()处理成只是到其所在位置的一个非局部跳转。

Longjmp()和 setjmp()联合体运行于异常生命期的 2 和 3 阶段。longjmp(j,r)产生异常对象 r（一个整数），并且作为返回值传送到 setjmp(j)处。实际上，setjmp()函数通报了异常 r。

1.7 信号

C 函数库也提供了标准的（虽然原始的）“事件”处理包。这个包定义了一组事件和信号，以及标准的方法来触发和处理它们。这些信号或者表示了一个异常状态或者表示了一个不协调的外部事件；基于所谈论的主题，我将只集中讨论异常信号。

为了使用这些包，需要包含标准头文件<signal.h>。这个头文件申明了函数 raise()和 signal()，数据类型 sig_atomic_t，和以 SIG 开头的信号事件宏。标准要求有六个信号宏，也许你所用的运行库实的现会再附加一些。这些信号被固定死在<signal.h>中，你不能增加自定义的信号。信号通过调用 raise()产生并被处理函数捕获。运行时体系提供默认处理函数，但你能通过 signal()函数安装自己的处理函数。处理函数可以通过 sig_atomic_t 类型的对象和外部进行通讯;如类型名所示,对这样的对象的操作是原子操作或者说中断安全的。

当你挂接信号处理函数时,通常提供一个函数地址,这个的函数必须接受一个整型值(所要处理的信号事件)，并且无返回。这样，信号处理函数有些象 setjmp()；它们所收到的仅有的异常信息是单个整数：

```
void handler(int signal_value);

void f(void)
   {
   signal(SIGFPE, handler); /* register handler */
   /* ... */
```

```
raise(SIGFPE); /* invoke handler, passing it 'SIGFPE' */
}
```

只可其一地，你可以安装两个特别的处理函数：

- signal(SIGxxx,SIG_DFL)，为指定的信号挂接系统的缺省处理函数。
- signal(SIGxxx，SIG_IGN)，告诉系统忽略指定的信号。

signal()函数返回前次挂接的处理函数的地址（表明挂接成功），或返回 SIG_ERR（表明挂接失败）。

处理函数被调用表明信号正在试图恢复异常。当然，你可以在处理函数中随意地调用 abort()、exit()或 longjmp()，有效地将信号解释为终止异常。有趣的是，abort()自己事实上在内部调用了 raise(SIGABRT)。SIGABRT 的缺省处理函数发起了一个诊断并终止程序，当然你可以安装自己的处理函数来改变这个行为。不能改变的是 abort()的终止程序的行为。Abort()理论上的实现如下：

```
void abort(void)
{
raise(SIGABRT);
exit(EXIT_FAILURE);
}
```

也就是说，即使你的 SIGABRT 处理函数返回了，abort()仍然中止你的程序。

C 语言标准在信号处理函数的行为上增加了一些限制和解释。如果你有 C 语言标准，我建议你查阅条款 7.7.1.1 的细节。（很不幸,C 语言和 C++语言的标准在 Internet 都得不到。）

<signal.h>的申明覆盖了异常的整个生存期，从产生到死亡。在标准的 C 语言运行期库中，它们是最接近于异常完全解决方案的。

1.8 全局变量

<setjmp.h>和<signal.h>一般使用异常通知体系：当试图通知一个异常事件时唤醒一个处理函数。如果你更愿意使用轮询体系，C 标准库在<errno.h>提供了例子。这个头文件定义了 errno 及其一些可能的取值。标准要求这样三个值：EDOM、ERANGE 和 EILSEQ，分别适用于域、范围和多字节顺序错误，你的编译器可能又加了些其它的，它们全以字母“E”开头。

errno，通过由运行库的代码设置它而用户代码查询它的办法将二者联系起来，运行于异常生命期的 1 到 3：运行库产生异常对象（一个简单的整数），把值拷给 errno，然后依赖用户的代码去轮询和检测这个异常。

运行库主要在<math.h>和<stdio.h>的函数中使用 errno。errno 在程序开始时设为 0，函数库程序不会再次把它设为 0。因此，要检测错误，你必须先将 errno 设为 0，再调用运行库程序，调用完后检查 errno 的值：

```
#include <errno.h>
#include <math.h>
#include <stdio.h>

int main(void)
{
double x, y, result;
```

```
   /* ... somehow set 'x' and 'y' ... */
   errno = 0;
   result = pow(x, y);
   if (errno == EDOM)
      printf("domain error on x/y pair\n");
   else if (errno == ERANGE)
      printf("range error on result\n");
   else
      printf("x to the y = %d\n", (int) result);
   return 0;
   }
```

注意：errno 不一定要绑在一个对象上：

```
int *_errno_function()
   {
   static int real_errno = 0;
   return &real_errno;
   }

#define errno (*_errno_function())

int main(void)
   {
   errno = 0;
   /* ... */
   if (errno == EDOM)
      /* ... */
   }
```

你可以在自己的程序中采用这样的技巧，对 errno 及其值进行模拟。使用 C++的话，你当然可以把这种策略扩展到类或命名空间的对象和函数上。（实际上，在 C++中，这个技巧是 Singleton Pattern 的基础。）

1.9 返回值和回传参数

象 errno 这样的异常对象不是没有限制的：

- 所有相关联的部分必须一致，确保设置和检查同一个对象。
- 无关的部分可能意外地修改了对象。
- 如果没有在调用程序前重设对象，或在调用下一步前没有检查它们，你就可能漏了异常。
- 宏和内部代码中的对象在重名时将掩盖异常对象。
- 静态对象天生就不是（多）线程安全的。

总之，这些对象很脆弱：你太容易用错它们，编译器没有警告程序却有不可预测的行为。要排除这些不足，你需要这样的对象：

- 被两个正确的部分访问——一个产生异常，一个检测异常。

- 带有一个正确的值。
- 名字不能被掩盖

1.10 线程安全。

函数返回值满足这些要求，因为它们是无名的临时变量，由函数产生而只能被调用者访问。调用一完成，调用者就可以检查或拷贝返回值；然后原始的返回对象将消失而不能被重用。又因为是无名的，它不能被掩盖。

（对于 C++，我假设只有右值函数调用表达，也就是说不能返回引用。由于我限定现在只谈论 C 兼容的技巧，而 C 不支持引用，这样的假设是合理的。）

返回值出现在异常生命期的阶段 2。在调用和被调用函数的联合体中，这只是完整的异常处理的一部分：

```
int f()
   {
   int error;
   /* ... */
   if (error) /* Stage 1: error occurred */
      return -1; /* Stage 2: generate exception object */
   /* ... */
   }

int main(void)
   {
   if (f() != 0) /* Stage 3: detect exception */
      {
      /* Stage 4: handle exception */
      }
   /* Stage 5: recover */
   }
```

返回值是 C 标准库所喜欢的异常传播方法。看下面的例子：

```
if ((p = malloc(n)) == NULL)
   /* ... */

if ((c = getchar()) == EOF)
   /* ... */

if ((ticks = clock()) < 0)
   /* ... */
```

注意，典型的 C 习惯用法：在同一条语句中接收返回值和检测异常。这种压缩表达式重载一个通道（返回值对象）来携带两个不同的含义：合法的数据值和异常值。代码必须按两条路来解释这个通道，直到知道哪个是正确的。

这种函数返回值的用法常见于很多语言中，尤其是 Microsoft 开发的语言无关的 Component Object Model (COM)。COM 方法通过返回一类型为 HRESULT（特别安排的 32 位无

符号值）的对象提示异常。和刚讨论的例子不同，COM 的返回值只携带状态和异常信息；回传信息通过参数列表中的指针进行。

回传指针和 C++的引用型的参数是函数返回值的变形，但有些明显的不同：

- 你能忽略和丢弃返回值。回传参数则绑定到了相应的实参上，所以不可能完全忽略它们。和返回值相比，参数在函数和它们的调用者间形成了紧耦合。
- 通过回传参数可以返回任意个数的值，而通过返回值只能返回一个值。所以回传参数提供了多个返回值。
- 返回值是临时对象：它们在调用前不存在，并且在调用结束是消失。实参的生命期远长于函数的调用过程。

1.11 小结

这次大概地介绍了异常和标准 C 对它的传统支持。第二部分，我将研究 Microsoft 对标准 C 方法的扩展：特有的异常处理宏、结构化异常处理或说 SEH。我将总结所有 C 兼容方法（包括 SEH）的局限性，并在第三部分拉开 C++异常的序幕。

回到目录

2. Microsoft 对异常处理方法的扩展

前次，我概述了异常的分类和 C 标准库支持的处理方法。这次讨论 Microsoft 对这些方法的扩展：结构化异常处理（SEH）和 Microsoft Foundation Class （MFC）异常处理。SEH 对 C 和 C++都有效，MFC 异常体系只对 C++有效。

2.1 机构化异常处理

机构化异常处理是 Windows 提供的服务功能并对所有语言写的程序有效。在 Visual C++中，Microsoft 封装和简化了这些服务（通过非标准的关键字和库程序）。Windows 平台的其它编译器可能选择不同的方式来到达相似的结果。在这个专栏中，名词“Structured Exception Handling”和“SEH”专指 Visual C++对 Windows 异常服务的封装。

2.2 关键字

为了支持 SEH，Micorsoft 用四个新关键字扩展了 C 和 C++语言：

- __except
- __finally
- __leave
- __try

因为这是非标关键字，必须打开扩展选项后再编译（关掉/Fa）。

为什么这些关键字带下划线？C++标准（条款 17.4.3.1.2，“Global names”）规定：下列名字和函数总是保留给编译器：

- 所有带双下划线（__）或以一个下划线加一个大写字母开始的名字保留给编译器随意使用。
- 所有以一个下划线开始的名字保留给编译器作全局名称用。

C 标准有类似的申明。

既然 SEH 的关键字符合上面的规则，Microsoft 就有权这样使用它们。这也表明，你不被允许在自己的程序中使用保留的名字。你必须避免定义名字类似__MYHEADER_H__或_FatalError 的标识符。

有趣而又不幸地，Visual C++的 application wizards 产生的源代码使用了保留的标识符。例如，如果你用 ATL COM App Wizard 生成一个新的 service，结果框架代码定义了如_Handler 和_twinMain 的名字——标准所说的你的程序不能使用的保留名称。

要减少这个不合规定行为，你当然可以手工更改这些名称。还好，这些有疑问的名字都是类的私有变量，在类的定义外面是不可见的，在.h 和.cpp 中进行全局替换是可行的。不幸的是，有一个函数（_twinMain）和一个对象（_Module）被申明了 extern，也就是说程序的其它部分会假定你使用了这些名字。（事实上，Visual C++库 libc.lib 在连接时需要名字_twinMain 可用。）

我建议你保留 Wizard 生成的名字，不要在你自己的代码中定义这样的名字就可以了。另外，你应该将所有不合标准的定义写入文档并留给程序的维护人员；记住，Visual C++以后的版本（和现有的其它 C++编译器）可能以另外的方式使用这些名字，从而破坏了你的代码。

2.3 标识符

Microsoft 也在非标头文件 excpt.h 中定义了几个 SEH 的标识符，并且包含入 windows.h 中。在其内部，定义了：

- 供__except 的过滤表达式使用的过滤结果宏。
- Win32 对象和函数的别名宏，用于查询异常信息和状态。
- 伪关键字宏，和前面谈到的四个关键字有着相同名字和含义，但没有下划线。（例如，宏 leave 对应 SEH 关键字__leave。）

Microsoft 用这些宏令我抓狂。他们对同一个函数了定义多个别名。例如，excpt.h 有如下申明和定义：

```
unsigned long __cdecl    _exception_code(void);
#define GetExceptionCode _exception_code
#define exception_code   _exception_code
```

也就是说，你可以用三种方法调用同一函数。你用哪个？并且，这些别名会如你所期望地被维护吗？

在 Microsoft 的文档中，它看起来偏爱 GetExceptionCode，它的名字和其它全局 Windows API 函数风格一致。我在 MSDN 中搜索到 33 处 GetExceptionCode，两个_exception_code，而 exception_code 个数为 0。根据 Microsoft 的引导，推荐使用 GetExceptionCode 及类似名称的其它函数。

因为_exception_code 的两个别名是宏，所以你不能再使用同样的名字了。我曾经犯过这个错，当我在为这个专栏写例程的时候。我定义了一个局部对象叫 exception_code（大概是吧）。实际上我就是定义了一个局部对象叫_exception_code，这是我无意中使用的宏 exception_code 展开的结果。当我一想到是这个问题，解决方案就是简单地将我的对象名字从 exception_code 改为 code。

最后，excpt.h 定义了一个特别的宏——“try”——已经成为 C++真正的关键字的东西。这意味着你不能在包含了 excpt.h 的编译单元中简单地混合 SEH 和标准 C++的异常块，除非你愿意#undef 这个 try 宏。当这样 undef 而露出真正的 try 关键字时，要冒搞乱 SEH 的维

护人员大脑的危险。另一方面，精通标准 C++的程序员会将 try 理解为一个关键字而不是宏。

我认为，包含一个头文件（即使是象 excpt.h 这样的非标头文件）不应该改变符合语言标准的代码的行为。我更坚持掩盖或重定义掉语言标准定义的关键字是个坏习惯。我建议：＃undef try，同样不使用其它的伪关键字宏，直接使用真正的关键字（如__try）。

2.4 语法

最基本的 SEH 语法是 try 块。如下形式：

```
__try compound-statement handler
```

处理体：

```
__except ( filter-expression ) compound-statement
```

或：

```
__finally compound-statement
```

完整一点看，try 块如下：

```
__try
   {
   ...
   }
__except(filter-expression)
   {
   ...
   }
```

或：

```
__try
   {
   ...
   }
__finally
   {
   ...
   }
```

在__try 里面你必须使用一个 leave 语句：

```
__try
   {
   ...
   __leave;
   ...
   }
```

在更大的程序块中，一个 try 块被认为是个单条语句：

```
if (x)
   {
   __try
     {
     ...
     }
   __finally
     {
     ...
     }
   }
```

等价于：

```
if (x)
   __try
     {
     ...
     }
   __finally
     {
     ...
     }
```

其它注意点：

- 在给定的 try 块中你必须有一个正确的异常处理函数。
- 所有的语句必须合并。即使只有一条语句跟在__try、__except 或__finally 后面也必须将它放入{}中。
- 在异常处理函数中，相应的过滤表达式必须有一个或能转换为一个 int 型的值。

2.5 基本语意

上次我列举了异常生命期的 5 个阶段。在 SEH 体系下，这些阶段实现如下：

- 操作系统上报了一个硬件错误或检测到了一个软件错误，或用户代码检测到一个错误（阶段 1）。
- （通常是由用户调用 Win32 函数 RasieException 启动,)操作系统产生并触发一个异常对象（阶段 2）。这个对象是一个结构，其属性对异常处理函数可见。
- 异常处理函数“看到”异常，并且有机会捕获它（阶段 3 和 4）。取决于处理函数的意愿，异常将或者恢复或者终止。(阶段 5)。

一个简单的例子：

```
int filter(void)
   {
   /* Stage 4 */
   }
```

```
int main(void)
   {
   __try
     {
     if (some_error) /* Stage 1 */
       RaiseException(...); /* Stage 2 */
     /* Stage 5 of resuming exception */
     }
   __except(filter()) /* Stage 3 */
     {
     /* Stage 5 of terminating exception */
     }
   return 0;
   }
```

Microsoft 调用定义在__except 中的异常处理函数,和定义在__finally 中的终止函数。

一旦异常被触发，由__except 开始的异常处理函数被异常发生点顺函数调用链向外面询问。每个被发现的异常处理函数，其过滤表达式都被求值。每次求值后发生什么取决于其返回结果。

excpt.h 定义了 3 个过滤结果的宏，都是 int 型的：

- EXCEPTION_CONTINUE_EXECUTION = -1
- EXCEPTION_CONTINUE_SEARCH = 0
- EXCEPTION_EXECUTE_HANDLER = 1

前面我说过，过滤表达式必须兼容 int 型，所以它们和这 3 个宏的值匹配。这个说法太保守了：我的经验显示 Visual C++接受的过滤表达式可以具有所有的整型、指针型、结构、数组甚至是 void 型!（但我在尝试浮点指针时遇到了编译错误。）

更进一步，所有求出的值看来都有效（至少对整型如此）。所有非零且符号位为 0 的值效果相当于 EXCEPTION_EXECUTE_HANDLER，而符号位为 1 的相当于 EXCEPTION_CONTINUE_EXECUTION。这大概是按位取模的结果。

如果一个异常处理函数的过滤求值结果是 EXCEPTION_CONTINUE_SEARCH，这个处理函数拒绝捕获异常，将继续搜索下一个异常处理函数。

通过由过滤表达式产生一个非 EXCEPTION_CONTINUE_SEARCH 来捕获异常，一旦捕获，程序就恢复。怎么恢复仍然由过滤表达式的值决定：

- EXCEPTION_CONTINUE_EXECUTION：表现为恢复异常。从发生异常处下面开始执行。异常处理函数本身的代码不执行。
- EXCEPTION_EXECUTE_HANDLER：表现为终止异常。从异常发生处开始退栈，一路上所遇到终止函数都被执行。栈退到捕获异常的处理函数所在的一级为止。进入处理函数体并执行。

如名所示,终止处理函数(以__finally 开始的代码)在终止异常时被调用。里面是 clean up 代码，它们就象 C 标准库中的 atexit()函数和 C++的析构函数。终止处理函数在正常执行流程也会进入，就象不是捕获型代码。相反，异常处理函数总表现为捕获型：它们只在其过滤表达式求值为 EXCEPTION_EXECUTE_HANDLER 时才进入。

终止处理函数并不明确知道自己是从正常流程进入的还是在一个 try 块异常终止时进入的。要判断这点，可以调用 AbnormalTermination 函数。此函数返回一个 int，0 表明是

从正常流程进入的，其它值表明在异常终止时进入的。

AbnormalTermination 实际上是个指向_abnormal_termination()的宏。Visual C++将_abnormal_termination()设计为环境敏感的函数，就象一个关键字。你不能随便调用这个函数，只能在终止处理函数中调用。这意味着你不能在终止处理函数中调用一个中间函数，再在此中间函数中调用_abnormal_termination()，这样做会得到一个编译期错误。

2.6 例程

下面的 C 例子显示了不同的过滤表达式值和处理函数本身类型的相互作用。第一个版本是个小的完整程序，以后的版本都在它前面一个上有小小的改动。所有的版本都自解释的，你能看清流程和行为。

程序通过 RaiseException()触发一个异常对象。RaiseException()函数的第一个参数是异常的代码，类型是 32 位无符号整型（DWORD）；Microsoft 为用户自定义的错误保留了[0xE0000000,0xEFFFFFFF]的范围。其它参数一般填 0。

这里使用的异常过滤器很简单。实际使用中，大概要调用 GetExceptionCode()和 GetExceptionInformation()来查询异常对象的属性。

2.7 Version #1: Terminating Exception

用 Visual C++生成一个空的 Win32 控制台程序，命名为 SEH_test，选项为默认。将下列 C 源码加入工程文件：

```
#include <stdio.h>
#include "windows.h"
#define filter(level, status) \
   ( \
   printf("%s:%*sfilter => %s\n", \
         #level, (int) (2 * (level)), "", #status), \
   (status) \
   )
#define termination_trace(level) \
   printf("%s:%*shandling %snormal termination\n", \
         #level, (int) (2 * (level)), "", \
         AbnormalTermination() ? "ab" : "")
static void trace(int level, char const *message)
   {
   printf("%d:%*s%s\n", level, 2 * level, "", message);
   }
extern int main(void)
   {
   DWORD const code = 0xE0000001;
   trace(0, "before first try");
   __try
      {
      trace(1, "try");
```

```
    __try
       {
       trace(2, "try");
       __try
          {
          trace(3, "try");
          __try
             {
             trace(4, "try");
             trace(4, "raising exception");
             RaiseException(code, 0, 0, 0);
             trace(4, "after exception");
             }
          __finally
             {
             termination_trace(4);
             }
       end_4:
          trace(3, "continuation");
          }
       __except(filter(3, EXCEPTION_CONTINUE_SEARCH))
          {
          trace(3, "handling exception");
          }
       trace(2, "continuation");
       }
 __finally
       {
       termination_trace(2);
       }
    trace(1, "continuation");
    }
__except(filter(1, EXCEPTION_EXECUTE_HANDLER))
    {
    trace(1, "handling exception");
    }
trace(0, "continuation");
return 0;
}
```

现在编译代码。（可能会得到 label end_4 未用的警告；先忽略。）

注意：

- 程序有四个嵌套 try 块，两个有异常处理函数，两个有终止处理函数。为了更好地显示嵌套和控制流程，我把它们全部放入同一个函数中。实际编程中可能是放在多个函数或

多个编译单元中的。

- 追踪运行情况，输出结果显示当前块的嵌套层次。
- 异常过滤器被实现为宏。第一个参数是嵌套层次，第二个才是实际要处理的值。
- 终止处理函数通过 termination_trace 宏跟踪其执行情况，显示出调用它们的原因。（记住，终止处理函数即使没有发生异常也会进入的。）

运行此程序，将看到如下输出：

```
0:before first try
1:  try
2:    try
3:      try
4:        try
4:        raising exception
3:      filter => EXCEPTION_CONTINUE_SEARCH
1:  filter => EXCEPTION_EXECUTE_HANDLER
4:        handling abnormal termination2:    handling abnormal termination
1:  handling exception
0:continuation
```

事件链：

- 第四层 try 块触发了一个异常。这导致顺嵌套链向上搜索，查找愿意捕获这个异常的异常过滤器。
- 碰到的第一个异常过滤器（在第三层）得出了 EXCEPTION_CONTINUE_SEARCH，所以拒绝捕获这个异常。继续搜索下一个异常处理函数。
- 碰到的下一个异常过滤器（在第一层）得出了 EXCEPTION_EXECUTE_HANDLER。这次，这个过滤器捕获这个异常。因为它求得的值，异常将被终止。
- 控制权回到异常发生点，开始退栈。沿路所有的终止处理函数被运行，并且所有的处理函数都知道异常终止发生了。一直退栈到控制权回到捕获异常的异常处理函数（在第一层）。在退栈时，只有终止处理函数被执行，中间的其它代码被忽略。
- 控制权一回到捕获异常的异常处理函数（在第一层），将以正常状态继续执行。

注意，控制权在同一嵌套层传递了两次：第一次异常过滤表达式求值，第二次在退栈和执行终止处理函数时。这造成了一种危害可能：如果一个异常过滤表达式以某种终止处理函数不期望的方式修改了的什么。一个基本原则就是，你的异常过滤器不能有副作用；如果有，则必须为你的终止处理函数保存它们。

2.8 版本 2：未捕获异常

将例程中的这行：

```
__except(filter(1, EXCEPTION_EXECUTE_HANDLER))
```

改为

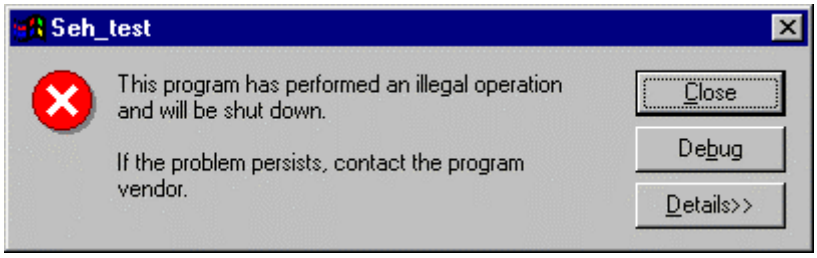
```
__except(filter(1, EXCEPTION_CONTINUE_SEARCH))
```

于是没有异常过滤器捕获这个异常。执行修改后的程序，你将看到：

```
0:before first try
1:  try
```

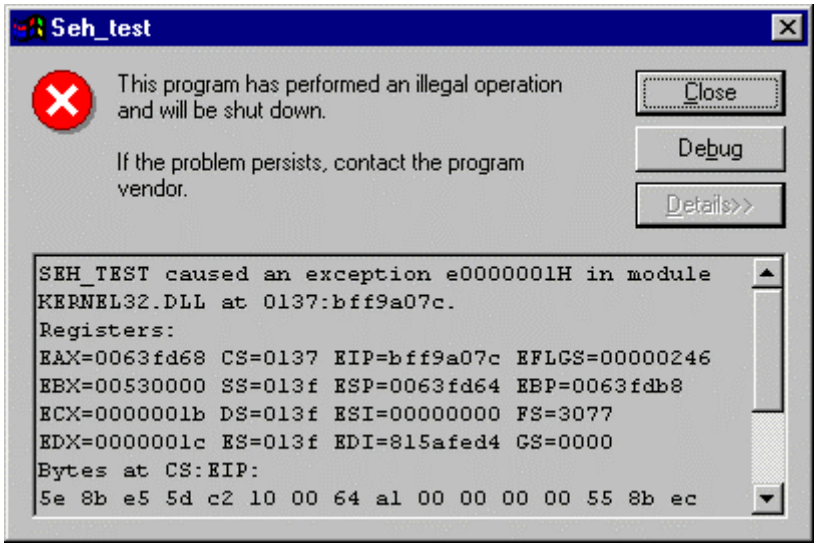
```
2:    try
3:      try
4:        try
4:        raising exception
3:      filter => EXCEPTION_CONTINUE_SEARCH
1:  filter => EXCEPTION_CONTINUE_SEARCH
```

接着出现这个对话框：



1. 用户异常对话框

点“Details”将其展开



2. 用户异常对话框的详细信息

在出错信息中可看到：出错程序是 SEH_TEST，通过 RaiseException 抛出的原始异常码是 e0000001H。

这个异常漏出了程序，最后被操作系统捕获和处理。有些象你的程序是这么写的：

```
__try
   {
   int main(void)
      {
      ...
      }
   }
__except(exception_dialog(), EXCEPTION_EXECUTE_HANDLER)
   {
   }
```

按对话框上的“Close”，所有的终止处理函数被执行，并退栈，直到控制权回到捕获异常的处理函数。你可以明显看到这些信息：

```
4:        handling abnormal termination
2:    handling abnormal termination
```

它们出现在关闭对话框之后。注意，你没有看到：

```
0:continuation
```

因为它的实现代码在终止处理函数之外，而退栈时只有终止处理函数被执行。

对我们的试验程序而言，捕获异常的处理函数在 main 之外，这意味着传递异常的行为到了程序范围外仍然在继续。其结果是，程序被终止了。

2.9 版本 3：恢复异常

接下来，改：

```
__except(except_filter(3, EXCEPTION_CONTINUE_SEARCH))
```

为：

```
__except(except_filter(3, EXCEPTION_CONTINUE_EXECUTION))
```

重新编译并运行。可以看到这样的输出：

```
0:before first try
1:  try
2:    try
3:      try
4:        try
4:        raising exception
3:      filter => EXCEPTION_CONTINUE_EXECUTION
4:        after exception
4:        handling normal termination
3:      continuation
2:    continuation
2:    handling normal termination
1:  continuation
0:continuation
```

因为第三层的异常过滤器已经捕获了异常，第一层的过滤器不会被求值。捕获异常的过滤器求值为 EXCEPTION_CONTINUE_EXECUTION，因此异常被恢复。异常处理函数不会被进入，将从异常发生点正常执行下去。

2.10 版本 4：异常终止

这样的结构：

```
__try
   {
   /* ... */
```

```
   return;
   }
```

或：

```
__try
   {
   /* ... */
   goto label;
   }
__finally
   {
   /* ... */
   }
/* ... */
label:
```

被认为是 try 块异常终止。以后调用 AbnormalTermination()函数的话将返回非 0 值，就象异常仍然存在。

要看其效果，改这两行：

```
trace(4, "raising exception");
RaiseException(exception_code, 0, 0, 0);
```

为：

```
trace(4, "exiting try block");
goto end_4;
```

第 4 层的 try 块不是被一个异常结束的，现在是被 goto 语句结束的。运行结果：

```
0:before first try
1:  try
2:    try
3:      try
4:        try
4:        exiting try block
4:        handling abnormal termination
3:      continuation
2:    continuation
2:    handling normal termination
1:  continuation
0:continuation
```

第 4 层的终止处理函数认为它正在处理异常终止，虽然并没有发生过异常。（如果发生过异常的话，我们至少能从一个异常过滤器的输出信息上看出来的。）

结论：你不能只依赖 AbnormalTermination()函数来判断异常是否仍存在。

2.11 版本 5：正常终止

如果想正常终止一个 try 块，也就是想要 AbnormalTermination() 函数返回 FALSE，应该使用 Microsoft 特有的关键字__leave。想验证的话，改：

```
goto end_4;
```

为：

```
__leave;
```

重新编译并运行，结果是：

```
0:before first try
1:  try
2:    try
3:      try
4:        try
4:        exiting try block
4:        handling normal termination
3:      continuation
2:    continuation
2:    handling normal termination
1:  continuation
0:continuation
```

和版本 4 的输出非常接近，除了一点：第 4 层的终止处理函数现在认为它是在处理正常结束。

2.12 版本 6：隐式异常

前面的程序版本处理的都是用户产生的异常。SEH 也可以处理 Windows 自己抛出的异常。

改这行：

```
trace(4, "exiting try block");
__leave;
```

为：

```
trace(4, "implicitly raising exception");
*((char *) 0) = 'x';
```

这导致 Windows 的内存操作异常（引用空指针）。接着改：

```
__except(except_filter(3, EXCEPTION_CONTINUE_EXECUTION))
```

为：

```
__except(except_filter(3, EXCEPTION_EXECUTE_HANDLER))
```

以使程序捕获并处理异常。

执行结果为：

```
0:before first try
1:  try
2:    try
3:      try
4:        try
4:        implicitly raising exception
3:      filter => EXCEPTION_EXECUTE_HANDLER
4:        handling abnormal termination
3:      handling exception
2:      continuation
2:    handling normal termination
1:  continuation
0:continuation
```

如我们所预料，Windows 在嵌套层次 4 中触发了一个异常，并被层次 3 的异常处理函数捕获。

如果你想知道捕获的精确异常码，可以让异常传到 main 外面去，就象版本 2 中做的。为此，改：

```
__except(except_filter(3, EXCEPTION_EXECUTE_HANDLER))
```

为：

```
__except(except_filter(3, EXCEPTION_CONTINUE_SEARCH))
```

结果对话框在按了“Details”后，显示的信息非常象用户异常。

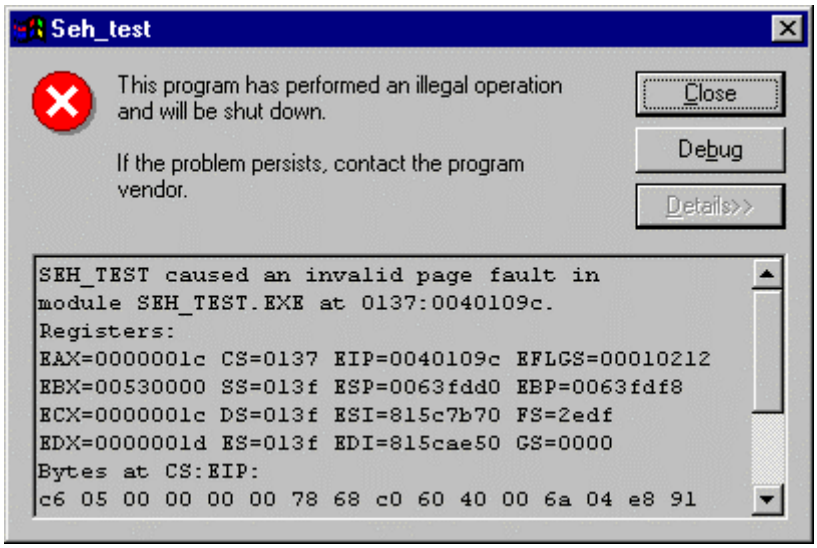


图 3　内存异常对话框

和版本 2 的对话框不同是，上次显示了特别的异常码，这次说了“invalid page fault”——更用户友好些吧。

2.13 C++考虑事项

在所有 C 兼容异常处理体系中，SEH 无疑是最完善和最灵活的（至少在 Windows 环境下）。具有讽刺意味的，它也是 Windows 体系以外的环境中最不灵活的，它将你和特殊的运行平台及 Visaul C++源码兼容的编译器牢牢绑在了一起。

如果只使用 C 语言，并且不考虑移植到 Windows 平台以外，SEH 很好。但如果使用 C++并考虑可移植性，我强烈建议你使用标准 C++异常处理而不用 SEH。你可以在同一个程序中同时使用 SEH 和标准 C++异常处理，只有一个限制：如果在有 SEH try 块的函数中定义了一个对象，而这个对象又没有 non-trivial（无行为的）析构函数，编译器会报错。在同一函数中同时使用这样的对象和 SEH 的__try，你必须禁掉标准 C++异常处理。

（Visual C++默认关掉标准 C++异常处理。你可以使用命令行参数/GX 或 Visual Studio 的 Project Settings 对话框打开它。）

在以后的文章中，我会在讨论标准 C++异常处理时回顾 SEH。我想将 SEH 整合入 C++的主流中，通过将结构化异常及 Windows 运行库支持映射为 C++异常和标准 C++运行库支持。

2.14 MFC 异常处理

说明：这一节我需要预先引用一点点标准 C++异常处理的知识，但要到下次才正式介绍它们。这个提前引用是不可避免的，也是没什么可惊讶的，因为 Microsoft 将它们的 MFC 异常的语法和语义构建在标准 C++异常的语法和语义的基础上。

我到现在为止所讲的异常处理方法对 C 和 C++都有效。在此之外，Microsoft 对 C++程序还有一个解决方案：MFC 异常处理类和宏。Microsoft 现在认为 MFC 异常处理体系过时了，并鼓励你尽可能使用标准 C++异常处理。然而 Visual C++仍然支持 MFC 异常类和及宏，所以我将给它个简单介绍。

Microsoft 用标准 C++异常实现了 MFC3.0 及以后版本。所以你必须激活标准 C++异常才能使用 MFC，即使你不打算显式地使用这些异常。前面说过，你必须禁掉标准 C++异常来使用 SEH，这也意味着你不能同时使用 MFC 宏和 SEH。Microsoft 明文规定这两个异常体系是互斥的，不能在同一程序中混合使用。

SEH 是扩展了编译器关键字集，MFC 则定义了一组宏：

- TRY
- CATCH, AND_CATCH, 和 END_CATCH
- THROW 和 THROW_LAST

这些宏非常象 C++的异常关键字 try、catch 和 throw。

另外，MFC 提供了异常类体系。所有名字为 CXXXException 形式的类都是从抽象类 CException 派生的。这类似于标准 C++运行库在<setdxcept>中申明的从 std::exception 开始的派生体系。但，标准 C++的关键字可以处理绝大部分类型的异常对象，而 MFC 宏只能处理 CException 的派生类型对象。

对于每个 MFC 异常类 CXXXException，都有一个全局的辅助函数 AfxThrowXXXException()，它构造、初始化和抛出这个类的对象。你可以用这些辅助函数处理预定义的异常类型，用 THROW 处理自定义的对象（当然，它们必须是从 CException 派生的）。

基本的设计原则是：

- 用 TRY 块包含可能产生异常的代码。
- 用 CATCH 检测并处理异常。异常处理函数并不是真的捕获对象，它们其实是捕获了指向异常的指针。MFC 靠动态类型来辨别异常对象。比较一下，SEH 靠运行时查询异常码来辨别异常。
- 可以在一个 TRY 块上捆绑多个异常处理函数，每个捕获一个 C++静态类型不同的对象。第一个处理函数使用宏 CATCH，以后的使用 AND_CATCH，用 END_CATCH 结束处理函数队列。
- MFC 自己可能触发异常，你也可以显式触发异常（通过 THROW 或 MFC 辅助函数）。在异

常处理函数内部，可以用 THROW_LAST 再次抛出最近一次捕获的异常。

- 异常一被触发，异常处理函数就将被从里到外进行搜索，和 SEH 时一样。搜索停止于找到一个类型匹配的异常处理函数。所有异常都是终止。和 SEH 不一样，MFC 没有终止处理函数，你必须依赖于局部对象的析构函数。

一个小 MFC 例子，将大部分题目都包括了：

```
#include <stdio.h>
#include "afxwin.h"
void f()
   {
   TRY
      {
      printf("raising memory exception\n");
      AfxThrowMemoryException();
      printf("this line should never appear\n");
      }
   CATCH(CException, e)
      {
      printf("caught generic exception; rethrowing\n");
      THROW_LAST();
      printf("this line should never appear\n");
      }
   END_CATCH
   printf("this line should never appear\n");
   }
int main()
   {
   TRY
      {
      f();
      printf("this line should never appear\n");
      }
   CATCH(CFileException, e)
      {
      printf("caught file exception\n");
      }
   AND_CATCH(CMemoryException, e)
      {
      printf("caught memory exception\n");
      }
 /* ... handlers for other CException-derived types ... */
   AND_CATCH(CException, e)
      {
      printf("caught generic exception\n");
      }
```

```
   END_CATCH
   return 0;
   }
/*
   When run yields
   raising memory exception
   caught generic exception; rethrowing
   caught memory exception
*/
```

记住，异常处理函数捕获指向对象的指针，而不是实际的对象。所以，处理函数：

```
CATCH(CException, e)
   {
   // ...
   }
```

定义了一个局部指针 CException *e 指向了被抛出的异常对象。基于 C++的多态，这个指针可以引用任何从 CException 派生的对象。

如果同一 try 块有多个处理函数，它们按从上到下的顺序进行匹配搜索的。所以，你应该将处理最派生类的对象的处理函数放在前面，不然的话，更派生类的处理函数不会接收任何异常的（再次拜多态所赐）。

因为你典型地想捕获 CException，MFC 定义了几个 CException 特有宏：

- CATCH_ALL(e) 和 AND_CATCH_ALL(e) ，等价于 CATCH(CException, e) 和 AND_CATCH(CException, e)。
- END_CATCH_ALL ，结束 CATCH_ALL... AND_CATCH_ALL 队列。
- END_TRY 等价于 CATCH_ALL(e);END_CATCH_ALL。这让 TRY... END_TRY 中没有处理函数或说是接收所有抛出的异常。

这个被指的异常对象由 MFC 隐式析构和归还内存。这一点和标准 C++异常处理函数不一样，MFC 异常处理不会让任何人取得被捕获的指针的所有权。因此，你不能用 MFC 和标准 C++体系同时处理相同的异常对象；不然的话，将导致内存泄漏：引用已被析构的对象，并重复析构和归还同一对象。

2.15 小结

MSDN 在线还有另外几篇探索结构化异常处理和 MFC 异常宏的文章。

下次我将介绍标准 C++异常，概述它们的特点及基本原理。我还会将它们和到现在已经看到的方法进行比较。

回到目录

3. 标准 C++异常处理的基本语法和语义

这次，我来概述标准 C++异常处理的基本语法和语义。顺便，我会将它和前两次提到的技术进行比较。（在本文及以后，我将标准 C++异常处理简称为 EH，将微软的方法称为 SEH。）

3.1 基本语法和语义

EH 引入了 3 个新的 C++语言关键字：

- catch
- throw
- try

异常通过如下语句触发

```
throw [expression]
```

函数通过“异常规格申明”定义它将抛出什么异常：

```
throw([type-ID-list])
```

可选项 type-ID-list 包含一个或多个类型的名字，以逗号分隔。这些异常靠 try 块中的异常处理函数进行捕获。

```
try compound-statement handler-sequence
```

处理函数队列包含一个或多个处理函数，形式如下：

```
catch ( exception-declaration ) compound-statement
```

处理函数的“异常申明”指明了这个函数将捕获什么类型的异常。

和 SEH 一样，跟在 try 和 catch 后面的语句必须刮在{}内，而整个 try 块组成一条完整的大语句。

例子：

```
void f() throw(int, some_class_type)
   {
   int i;
   // ... generate an 'int' exception
   throw i;
   // ...
   }

int main()
   {
   try
      {
      f();
      }
   catch(int e)
      {
      // ... handle 'int' exception ...
      }
   catch(some_class_type e)
      {
      // ... handle 'some_class_type' exception ...
      }
   // ... possibly other handlers ...
   return 0;
   }
```

异常规格申明是 EH 特有的，SEH 和 MFC 都没有类似的东西。一个空的异常规格申明表明函数不抛出任何异常：

```
void f() throw()
   {
   // ... function throws no exceptions ...
   }
```

如果函数没有异常规格申明，它可以抛出任何类型的异常：

```
void f()
   {
   // ... function can throw anything or nothing ...
   }
```

当函数抛异常时，关键字 throw 通常后面带一个被抛出的对象：

```
throw i;
```

然而，throw 也可以不带对象：

```
catch(int e)
   {
   // ... handle 'int' exception ...
   throw;
   }
```

它的效果是再次抛出当前正被捕获的对象（int e）。因为空 throw 的作用是再次抛出已存在的异常对象，所以它必须位于 catch 语句块中。MFC 也有再次抛出异常的功能，SEH 则没有，它没有将异常对象交给过处理函数，所以没什么可再次抛出的。

就象函数原型中的参数申明一样，异常申明也可以是无名的：

```
catch(char *)
   {
   // ... handle 'char *' exception ...
   }
```

当这个处理函数捕获一个 char *型的异常对象时，它不能操作这个对象，因为这个对象没有名字。

异常申明还可以是这样的特殊形式：

```
catch(...)
   {
   // ... handle any type of exception ...
   }
```

就象不定参数中的“...”一样，异常申明中的“...”可以匹配任何异常的类型。

3.2 标准异常对象的类型

标准库函数可能报告错误。在 C 标准库中的报错方式在前面说过了。在 C++标准库中，有些函数抛出特定的异常，而另外一些根本不抛任何异常。

因为 C++标准中没有明确规定，所以 C++的库函数可以抛出任何对象或不抛。但 C++标准推荐运行库的实现通过抛出定义在<stdexecpt>中的异常类型或其派生类型来报告错误：

```
namespace std
   {
   class logic_error;         // : public exception
      class domain_error;     // : public logic_error
      class invalid_argument; // : public logic_error
      class length_error;     // : public logic_error
      class out_of_range;     // : public logic_error
   class runtime_error;       // : public exception
      class range_error;      // : public runtime_error
      class overflow_error;   // : public runtime_error
      class underflow_error;  // : public runtime_error
   }
```

这些（异常）类只对 C++标准库有约束力。在你自己的代码中，你可以抛出（和捕获）任何你所象要的类型。

3.3 标准中的其它申明

标准库头文件<exception>申明了几个 EH 类型和函数

```
namespace std
   {
   //
   // types
   //
   class bad_exception;
   class exception;
   typedef void (*terminate_handler)();
   typedef void (*unexpected_handler)();
   //
   // functions
   //
   terminate_handler set_terminate(terminate_handler) throw();
   unexpected_handler set_unexpected(unexpected_handler) throw();
   void terminate();
   void unexpected();
   bool uncaught_exception();
   }
```

提要：

- exception 是所有标准库抛出的异常的基类。
- uncaught_exception()函数在有异常被抛出却没有被捕获时返回 true，其它情况返回 false。它类似于 SEH 的函数 AbnormalTermination()。
- terminate()是 EH 的应急处理。它在异常处理体系陷入了不可恢复状态时被调用，经常

是因为试图重入（在前一个异常正处理过程中又抛了一个异常）。

- unexpected()在函数抛出一个它没有在“异常规格申明”中申明的异常时被调用。这个预料外的异常可能在退栈过程中被替换为一个 bad_excetion 对象。
- 运行库提供了缺省 terminate_handler()和 unexpected_handler() 函数处理对应的情况。你可以通过 set_terminate()和 set_unexpected()函数替换库的默认版本。

3.4 异常生命期

EH 运行于异常生命期的五个阶段：

- 程序或运行库遇到一个错误状况（阶段 1）并且抛出一个异常（阶段 2）。
- 程序的运行停止于异常点，开始搜索异常处理函数。搜索沿调用栈向上搜索（很象 SEH 终止异常时的行为）。
- 搜索结束于找到了一个异常申明与异常对象的静态类型相匹配（阶段 3）。于是进入相应的异常处理函数。
- 异常处理函数结束后,跳到此异常处理函数所在的 try 块下面最近的一条语句开始执行（阶段 5）。这个行为意味着 C++标准中异常总是终止。

这些步骤演示于这个简单的例子中：

```
#include <stdio.h>
static void f(int n)
   {
   if (n != 0) // Stage 1
      throw 123; // Stage 2
   }
extern int main()
   {
   try
      {
      f(1);
      printf("resuming, should never appear\n");
      }
   catch(int) // Stage 3
      {
      // Stage 4
      printf("caught 'int' exception\n");
      }
   catch(char *) // Stage 3
      {
      // Stage 4
      printf("caught 'char *' exception\n");
      }
   catch(...) // Stage 3
      {
      // Stage 4
      printf("caught typeless exception\n");
```

```
    }
  // Stage 5
  printf("terminating, after 'try' block\n");
  return 0;
  }
/*
  When run yields
    caught 'int' exception
    terminating, after 'try' block
*/
```

3.5 基本原理

C 标准库的异常体系处理 C++语言时有如下难题：

- 析构函数被忽略。既然 C 标准库异常体系是为 C 语言设计的，它们不知道 C++的析构函数。尤其，abort()、exit()和 longjmp()在退栈或程序终止时不调用局部对象的析构函数。
- 繁琐的。查询全局对象或函数返回值导致了代码混乱－你必须在所有可能发生异常的地方进行明确的异常情况检测，即使是异常情况可能实际上从不发生。因为这种方法是如此繁琐，程序员们可能会故意“忘了”检测异常情况。
- 无弹性的。Longjmp()“抛出”的只能是简单的 int 型。errno 和 signal()/raise()只使用了很小的一个值域集合，分辨率很低。Abort()和 exit()总是终止程序。Assert()只工作在 debug 版本中。
- 非固有的。所有的 C 标准库异常体系都需要运行库的支持，它不是语言内核支持的。

微软特有的异常处理体系也不是没有限制的：

- SEH 异常处理函数不是直接捕获一个异常对象，而是通过查询一个（概念性的）类似 errno 的全局值来判断什么异常发生了。
- SEH 异常处理函数不能组合，给定 try 块的唯有的一个处理函数必须在运行期识别和处理所有的异常事件。
- MFC 异常处理函数只能捕获 CException 及派生类型的指针。
- 通过包含定义了 MFC 异常处理函数的宏的头文件，程序包含了数百个无关的宏和申明。
- MFC 和 SEH 都是专属于与 Microsoft 兼容的开发环境和 Windows 运行平台的。

标准 C++异常处理避免了这些短处：

- 析构安全。在抛异常而进行退栈时，局部对象的析构函数被按正确的顺序调用。
- 不引人注目的。异常的捕获是暗地里的和自动的。程序员无需因错误检测而搞乱设计。
- 精确的。因为几乎任何对象都可以被抛出和捕获，程序员可以控制异常的内容和含义。
- 可伸缩的。每个函数可以有多个 try 块。每个 try 块可以有单个或一组处理函数。每个处理函数可以捕获单个类型，一组类型或所有类型的异常。
- 可预测的。函数可以指定它们将抛的异常类型，异常处理函数可以指定它们捕获什么类型的异常。如果程序违反了其申明，标准库将按可预测的、用户定义的方式运行。
- 固有的。EH 是 C++语言的一部分。你可以定义、throw 和 catch 异常而不需要包含任何库。

- 标准的。EH 在所有的标准 C++的实现中都可用。

基于更完备的想法,C++标准委员会考虑过两个 EH 的设计,在 D&E 的 16 章。(For a more complete rationale, including alternative EH designs considered by the C++ Standard's committee, check out Chapter 16 of the D&E.)

3.6 小结

下次，我将更深入挖掘 EH 的语言核心特性和 EH 的标准库支持。我也将展示 Microsoft Visual C++实现 EH 的内幕。我将开始标志出 EH 的那些 Visual C++只部分支持或完全不支持的特性，并且寻找绕过这些限制的方法。

在我相信设计 EH 的基本原理是健全的的同时,我也认为 EH 无意中包含了一些严重的后果。不用责备 C++标准的制订者的短视，我理解设计和实现有效的异常处理是多么的难。当我们遭遇到这些无意中的后果时，我将展示它们对你代码的微妙影响，并且推荐一些技巧来减轻其影响。

回到目录

4. 实例剖析 EH

到现在为止，我仍然逗留在 C 和 C++的范围内，但这次要稍微涉及一下汇编语言。目标：初步揭示 Visual C++对 EH 的 throw 和 catch 的实现。本文不是巨细无遗的，毕竟我的原则是只关注（C/C++）语言本身。然而，简单的揭示 EH 的实现对理解和信任 EH 大有帮助。

4.1 我们所害怕的唯一一件事

在 throw 过程中退栈时，EH 追踪哪个局部对象需要析构，预先安排必须的析构函数的调用，并且将控制权交给正确的异常处理函数。为了完成 EH 所需的记录和管理工作，编译器暗中在生成的代码中注入了数据、指令和库引用。

不幸的是，很多程序员（以及他们的经理）讨厌这种注入行为导致过分的代码膨胀。他们感到恐慌，认为 EH 会削弱程序的使用价值。所以，我认为 EH 触及了人们对未知的恐惧：因为源码中没有明确地表露出 EH 的工作，他们将作最坏的估算。

为了战胜这种恐惧，让我们通过短小的 Visual C++代码剖析 EH。

4.2 例 1：基线版本

生成一个新的 C++源文件 EH.cpp 如下：

```
class C
   {
public:
   C()
      {
      }
   ~C()
      {
      }
   };
```

```
void f1()
   {
   C x1;
   }

int main()
   {
   f1();
   return 0;
   }
```

然后，创建一个新的 Visual C++控制台项目，并包含 EH.CPP 为唯一的源文件。使用默认项目属性，但打开“生成源码/汇编混合的.asm 文件”选项。编译出 Debug 版本。在我机器上，得到的 EH.exe 是 23,040 字节。

打开 EH.asm 文件，你将发现 f1()函数非常接近预料：设置栈框架，调用 x1 的构造和析构函数，然后重设栈框架。特别地，你将注意到没有任何 EH 产物或记录——并不奇怪，因为程序没有抛出或捕获任何异常。

4.3 例 2：单异常处理函数

现在将 f1 改为如下形式：

```
void f1()
   {
   C x1;
   try
      {
      }
   catch(char)
      {
      }
   }
```

重新编译 EH.exe，然后注意文件大小。在我机器上，大小从 23,040 字节增到 29,696 字节。有些心跳吧，EH 导致了 29%的文件大小的增加。但看一下绝对增加，才 6,656 字节，并且绝大部分是来自于固定大小的库开销。剩下的少量才是额外注入到 EH.obj 中的代码和数据。

在 EH.asm 中，可以找到符号__$EHRec$定义了一个常量值，它表示对于栈框架的偏移量。每个函数都在其生成的代码中引用了__$EHRec$，编译器暗中定义了一个局部的“EH 记录”记录对象。

EH 记录是暂时的：和需要在代码中有个永久的静态记录相比，它们存在于栈中，在函数被进入时产生，在函数退出是消失。在且仅在函数需要提早析构局部对象时，编译器增加了 EH 记录（并且由局部代码维护它）。

隐含意思是，有些函数不需要 EH 记录。看这个，增加的第二个函数：

```
void f2()
   {
```

```
}
```
没有涉及对象和异常。重新编译程序。EH.asm 显示 f1()的栈中和以前一样包括一个 EH 记录，但 f2()的栈中没有。然而，如果将代码改成这样：

```
void f2()
   {
   C x2;
   f1();
   }
```

f2()现在定义了一个局部的 EH 记录，即使 f2()自己没有 try 块。为什么？因为 f2()调用了 f1(),而 f1()可能抛出异常而终止 f2()，因此需要提早析构 x2。

结论：如果一个包含局部对象的函数没有明确处理异常，但可能传递一个别人抛的异常，那么函数仍然需要一个 EH 记录和相应的维护代码。

这使你苦恼了吗？只要短路异常链就可以了。在我们的例子中，将 f1()的定义改成：

```
void f1() throw()
   {
   C x1;
   try
      {
      }
   catch(char)
      {
      }
   }
```

现在 f1()承诺不抛异常。结果，f2()不需要传递 f1()的异常，也就不需要 EH 记录了。你可以重新编译程序来核实，查看 EH.asm 并发现 f2()的代码不再提到__$EHRec$。

4.4 例 3：多个异常处理函数

EH 记录及其支撑代码不是编译所引入的唯有的记录。对给定 try 块的每个处理函数，编译器也都创建了入口表。想看得清楚些，将现在的 EH.asm 改名另存，并将 f1()扩展为：

```
void f1() throw()
   {
   C x1;
   try
      {
      }
   catch(char)
      {
      }
   catch(int)
      {
      }
   catch(long)
```

```
    {
    }
  catch(unsigned)
    {
    }
  }
```

重新编译，然后比较两次的 EH.asm。

（提醒：下面列出的 EH.asm，我没有忽略不相关的东西，也没有用省略号代替什么。精确的标号名在你的系统上可能不一样。并且不要以汇编语言分析器的眼光看这些代码。）

在我的 EH.asm 中，相关的名字、描述符和注释如下：

```
PUBLIC ??_R0D@8 ; char `RTTI Type Descriptor'
PUBLIC ??_R0H@8 ; int `RTTI Type Descriptor'
PUBLIC ??_R0J@8 ; long `RTTI Type Descriptor'
PUBLIC ??_R0I@8 ; unsigned int `RTTI Type Descriptor'

_DATA SEGMENT
??_R0D@8 DD FLAT:??_7type_info@@6B@ ; char `RTTI Type Descriptor'
         DD ...
         DB '.D', ...
_DATA ENDS

_DATA SEGMENT
??_R0H@8 DD FLAT:??_7type_info@@6B@ ; int `RTTI Type Descriptor'
         DD ...
         DB '.H', ...
_DATA ENDS

_DATA SEGMENT
??_R0J@8 DD FLAT:??_7type_info@@6B@ ; long `RTTI Type Descriptor'
         DD ...
         DB '.J', ...
_DATA ENDS

_DATA SEGMENT
??_R0I@8 DD FLAT:??_7type_info@@6B@ ; unsigned int `RTTI Type Descriptor'
         DD ...
         DB '.I', ...
_DATA ENDS
```

（对于“RTTI Type Descriptor”和“type_info”的注释提示我，Visual C++在 EH 和 RTTI 时使用了同样的类型名描述符。）

编译器同样生成了对在 xdata@x 段中定义的类型描述符的引用。每个类型对应一个捕获这种类型的异常处理函数的地址。这种描述符/处理函数对构成了 EH 库代码分发异常时的分

发表。这些也是从我的 EH.asm 下摘抄的，加上了注释和图表：

```
xdata$x SEGMENT

$T214 DD ...
      DD ...
      DD FLAT:$T217 ;---+
      DD ...        ;   |
      DD FLAT:$T218 ;---|---+
      DD 2 DUP(...) ;   |   |
      ORG $+4       ;   |   |
                    ;   |   |
$T217 DD ...        ;<--+   |
      DD ...        ;       |
      DD ...        ;       |
      DD ...        ;       |
                    ;       |
$T218 DD ...        ;<------+
      DD ...
      DD ...
      DD 04H        ; # of handlers
      DD FLAT:$T219 ;---+
      ORG $+4       ;   |
                    ;   |
$T219 DD ...        ;<--+
      DD FLAT:??_R0D@8 ; char RTTI Type Descriptor
      DD ...
      DD FLAT:$L206    ; catch(char) address

      DD ...
      DD FLAT:??_R0H@8 ; int RTTI Type Descriptor
      DD ...
      DD FLAT:$L207    ; catch(int) address

      DD ...
      DD FLAT:??_R0J@8 ; long RTTI Type Descriptor
      DD ...
      DD FLAT:$L208    ; catch(long) address

      DD ...
      DD FLAT:??_R0I@8 ; unsigned int RTTI Type Descriptor
      DD ...
      DD FLAT:$L209    ; catch(unsigned int) address

xdata$x ENDS
```

分发表表头（标号$T214、 $T217 和 $T218 处的代码）是 f1()专属的，并为 f1()的所有异常处理函数共享。$T219 出的分发表的每一个入口项都特属于 f1()的一个特定的异常处理函数。

更一般地，编译器为每一带 try 块的函数生成一个分发表表头，为每一个异常处理函数增加一个入口项。类型描述符为程序的所有分发表共享。（例如，程序中所有 catch(long)的处理函数引用同样的??_R0J@8 类型描述符。）

提要：要减小 EH 的空间开销，应该将程序中捕获异常的函数数目减到最小，将函数中异常处理函数的数目减到最小，将异常处理函数所捕获的异常类型减到最小。

4.5 例四：抛异常

用“抛一个异常”来将所有东西融会起来。将 f1()的 try 语句改成这样：

```
try
   {
   throw 123; // type 'int' exception
   }
```

重新编译程序，打开 EH.asm，注意新出现的东西（我同样加了的注释和图表）。

```
; in these exported names, 'H' is the RTTI Type Descriptor
;   code for 'int' -- which matches the data type of
;   the thrown exception value 123
PUBLIC __TI1H
PUBLIC __CTA1H
PUBLIC __CT??_R0H@84

; EH library routine that actually throws exceptions
EXTRN __CxxThrowException@8:NEAR

; new static data blocks used by library
;   when throwing 'int' exception
xdata$x SEGMENT

__CT??_R0H@84 DD ...                ;<------+
              DD FLAT:??_R0H@8      ;       |   ??_R0H@8 is RTTI 'int'
                                    ;       |    Type Descriptor
              DD ...                ;       |
              DD ...                ;       |
              ORG $+4               ;       |
              DD ...                ;       |
              DD ...                ;       |
                                    ;       |
__CTA1H       DD ...                ;<--+   |
              DD FLAT:__CT??_R0H@84 ;---|---+
                                    ;   |
```

```
__TI1H        DD ...                ;   |  __TI1H is argument passed to
              DD ...                ;   |   __CxxThrowException@8
              DD ...                ;   |
              DD FLAT:__CTA1H       ;---+

xdata$x ENDS
```

和类型描述符一样，这些新的数据块为全部程序共享，例如，所有抛 int 异常代码引用 __TI1H. 。同样要注意：相同的类型描述符被异常处理函数和 throw 语句引用。

翻到 f1()处，相关部分如下：

```
;void f1() throw()
;   {
;   try
;      {

       ...
       push $L224 ; Address of code to adjust stack frame via handler
                  ;   dispatch table.  Invoked by __CxxThrowException@8.
       ...

;      throw 123;

       push OFFSET FLAT:__TI1H       ; Address of data area diagramed
                                     ;   above
       mov DWORD PTR $T213[ebp], 123 ; 123 is the exception's value
       lea eax, DWORD PTR $T213[ebp]
       push eax
       call __CxxThrowException@8    ; Call into EH library, which in
                                     ;   turn eventually calls $L224
                                     ;   and $L216 a.k.a. 'catch(int)'
;      }
;   // ...
;   catch(int)

    $L216:

;      {

       mov eax, $L182 ; Return to EH library, which jumps to $L182
       ret 0

;      }
;   // ...
```

```
    $L182:

;   // Call local-object destructors, clean up stack, return
;   }

$L224:                          ; This label referenced by 'try' code.
    mov eax, OFFSET FLAT:$T223  ; $T223 is handler dispatch table, what
                                ;   had previously been label $T214
                                ;   before we added 'throw 123'
    jmp ___CxxFrameHandler      ; internal library routine
```

当程序运行时，__CxxThrowException@8（EH 的库函数）调用了$L216，catch(int)处理函数的地址。当处理函数一结束，程序就继续顺 EH 库中的代码向下运行，跳到$L224，继续向下并最终跳到$L182。这个标号是 f1()的终止和 cleanup 代码的地址，在其中调用了 x1 的析构函数。你可以在调试器下用单步进行验证。

4.6 小结

所有的异常处理体系都导致开销。除非你愿意在没有任何异常安全体系的情况下执行代码，你必须同意付出速度和空间的代价。EH 作为语言的特性有优点的：编译器明确知道 EH 的实现并可以据此优化它。

除了编译器的优化，你自己还有很多方法来优化。在以后的文章中，我将揭示特定的方法来将 EH 的代价减到最小。有些方法是基于标准 C++的，其它则依赖于 Visual C++的具体实现。

回到目录

5. C++的 new 和 delete 操作时的异常处理

今天,我们开始学习 C++的 new 和 delete 操作时的异常处理。首先，我将介绍标准 C++运行库对 new 和 delete 操作的支持。然后，介绍伴随着这些支持的异常。

5.1 New 和 Delete 表达式

当写

```
B *p = new D;
```

这里，B 和 D 是 class 类型，并且有构造和析构函数，编译器实际产生的代码大约是这样的：

```
B *p = operator new(sizeof(D));
D::D(p);
```

过程是：

- new 操作接受 D 对象的大小（字节为单位）作为参数。
- new 操作返回一块大小足以容纳一个 D 对象的内存的地址。
- D 的缺省构造函数被调用。这个构造函数传入的 this 指针就是刚刚返回的内存地址。
- 最终结果：*p 是个完整构造了的对象，静态类型是 B，动态类型是 D。

相似的，语句

```
delete p;
```

差不多被编译为

```
D::~D(p);
operator delete(p);
```

D 的析构函数被调用，被传入的 this 指针是 p；然后 delete 操作释放被分配的内存。

new 操作和 delete 操作其实是函数。如果你没有提供自己的版本，编译器会使用标准 C++运行库头文件<new>中申明的版本：

```
void *operator new(std::size_t);
void operator delete(void *);
```

和其它标准运行库函数不同，它们不在命名空间 std 内。

因为编译器隐含地调用这些函数，所以它必须知道如何寻找它们。如果编译器将它们放在特别的空间内（如命名空间 std），你就无法申明自己的替代版本了。因此，编译器按绝对名字从里向外进行搜索。如果你没有申明自己的版本，编译器最终将找到在<new>中申明的全局版本。

这个头文件包含了 8 个 new/delete 函数：

```
//
// new and delete
//
void *operator new(std::size_t);
void delete(void *);
//
// array new and delete
//
void *operator new[](std::size_t);
void delete[](void *);
//
// placement new and delete
//
void *operator new(std::size_t, void *);
void operator delete[](void *, void *);
//
// placement array new and delete
//
void *operator new[](std::size_t, void *);
void operator delete[](void *, void *);
```

前两个我已经介绍了。接下来两个分配和释放数组对象，而最后四个根本不分配和释放任何东西！

5.2 数组 new 和数组 delete

new[]操作被这样的表达式隐含调用：

```
B *p = new D[N];
```

编译器对此的实现是：

```
B *p = operator new[](sizeof(D) * N + _v);
for (std::size_t _i(0); _i < N; ++_i)
```

```
   D::D(&p[_i]);
```

前一个例子分配和构造单个 D 对象，这个例子分配和构造一个有 N 个 D 对象的数组。注意，传给 new[]操作的字节大小是 sizeof(D)*N + _v，所有对象的总大小加_v。在这里， _v 是数组分配时的额外开销。

如你所想，

```
delete[] p;
```

实现为：

```
for (std::size_t _i(_N_of(p)); _i > 0; --_i)
   D::~D(&p[i-1]);
operator delete[](p);
```

这里，_N_of(p)是个假想词，它依赖于你的编译器在检测*p 中的元素个数时的实现体系。

和 p = new D[N]不同（它明确说明了*p 包含 N 个元素），delete[] p 没有在编译期明确说明*p 元素个数。你的程序必须在运行期推算元素个数。C++标准没有强制规定推算的实现体系，而我所见过的编译器共有两种实现方法：

- 在*p 前面的字节中保存元素个数。其存储空间来自于 new[]操作时_v 字节的额外开销。
- 由标准运行库维护一个私有的 N 对 p 的映射表。

5.3 Placement New 和 Placement Delete

关键字 new 可以接受参数：

```
p = new(arg1, arg2, arg3) D;
```

（C++标准称这样的表达式为 “new with placement”或“placement new”，我马上会简单地解释原因。）这些参数会被隐含地传给 new 操作函数：

```
p = operator new(sizeof(D), arg1, arg2, arg3);
```

注意，第一个参数仍然是要生成对象的字节数，其它参数总是跟在它后面。

标准运行库定义了一个 new 操作的特别重载版本，它接受一个额外参数：

```
void *operator new(std::size_t, void *);
```

这种形式的 new 操作被如下的语句隐含调用：

```
p = new(addr) D;
```

这里，addr 是某些数据区的地址，并且类型兼容于 void *。

addr 传给这个特别的 new 操作，这个特别的 new 操作和其它 new 操作一样返回将被构造的内存的地址，但不需要在自由内存区中再申请内存，它直接将 addr 返回：

```
void *operator new(std::size_t, void *addr)
   {
   return addr;
   }
```

这个返回值然后被传给 D::D 作构造函数的 this 指针。

就这样，表达式

```
p = new(addr) D;
```

在 addr 所指的内存上构造了一个 D 对象，并将 p 赋为 addr 的值。这个方法让你有效地指定新生成对象的位置，所以被叫作“placement new”。

这个 new 的额外参数形式最初被设计为控制对象的位置的，但是 C++标准委员会认识到这样的传参体系可以被用于任意用途而不仅是控制对象的位置。不幸的是，术语“placement”已经被根据最初目的而制订，并适用于所有 new 操作的额外参数的形式，即使它们根本不试图控制对象的位置。

所以，下面每个表达式都是 placement new 的一个例子：

```
new(addr) D;     // calls operator new(std::size_t, void *)
new(addr, 3) D; // calls operator new(std::size_t, void *, int)
new(3) D;        // calls operator new(std::size_t, int)
```

即使只有第一个形式是一般被用作控制对象位置的。

5.4 placement Delete

现在，只要认为 placement delete 是有用处的就行了。我肯定会讲述理由的，可能就在接下来的两篇内。

Placement new 操作和 placement delete 操作必须成对出现。一般来说，每一个

```
void *operator new(std::size_t, p1, p2, p3, ..., pN);
```

都对应一个

```
void operator delete(void *, p1, p2, p3, ..., pN);
```

根据这条原则，标准运行库定义了

```
void operator delete(void *, void *);
```

以对应我刚讲的 placement new 操作。

5.5 数组 New 和数组 Delete

基于对称，标准运行库也申明了 placement new[]操作和 placement delete[]操作：

```
void *operator new[](std::size_t, void *);
void operator delete[](void *, void *);
```

如你所料：placement new[]操作返回传入的地址，而 placement delete[]操作的行为和我没有细述的 placement delete 操作行为几乎一样。

5.6 异常

现在，我们把这些 new/delete 和异常结合起来。再次考虑这条语句：

```
B *p = new D;
```

当其调用 new 操作而没有分配到足够内存时将发生什么？

在 C++的黑暗年代（1994 年及以前），对大部分编译器而言，new 操作将返回 NULL。这曾经是对 C 的 malloc 函数的合理扩展。幸运的是，我们现在生活在光明的年代，编译器强大了，类被设计得很漂亮，而编译运行库的 new 操作会抛异常了。

前面，我展示了在<new>中出现的 8 个函数的申明。那时，我做了些小手脚；这里是它们的完整形式：

```
namespace std
   {
   class bad_alloc
      {
      // ...
      };
```

```
   }

//
// new and delete
//
void *operator new(std::size_t) throw(std::bad_alloc);
void operator delete(void *) throw();
//
// array new and delete
//
void *operator new[](std::size_t) throw(std::bad_alloc);
void operator delete[](void *) throw();
//
// placement new and delete
//
void *operator new(std::size_t, void *) throw();
void operator delete(void *, void *) throw();
//
// placement array new and delete
//
void *operator new[](std::size_t, void *) throw();
void operator delete[](void *, void *) throw();
```

在这些 new 操作族中，只有非 placement 形式的会抛异常（std::bad_alloc）。这个异常意味着内存耗尽状态，或其它内存分配失败。你可能奇怪为什么 placement 形式不抛异常；但记住，这些函数实际上根本不分配任何内存，所以它们没有分配问题可报告。

没有 delete 操作抛异常。这不奇怪，因为 delete 不分配新内存，只是将旧内存还回去。

5.7 异常消除

相对于会抛异常的 new 操作形式，<new>中也申明了不抛异常的重载版本：

```
namespace std
   {
   struct nothrow_t
      {
      // ...
      };
   extern const nothrow_t nothrow;
   }

//
// new and delete
//
void *operator new(std::size_t, std::nothrow_t const &) throw();
void operator delete(void *, std::nothrow_t const &) throw();
```

```
//
// array new and delete
//
void *operator new[](std::size_t, std::nothrow_t const &) throw();
void operator delete[](void *, std::nothrow_t const &) throw();
```

这几个函数也被认为是 new 操作和 delete 操作的 placement 形式，因为它们也接收额外参数。和前面的控制对象分配位置的版本不同，这几个只是让你分辨出抛异常的 new 和不抛异常的 new。

```
#include <iostream>
#include <new>
using namespace std;

int main()
   {
   int *p;
   //
   // 'new' that can throw
   //
   try
      {
      p = new int;
      }
   catch(bad_alloc &)
      {
      cout << "'new' threw an exception";
      }
   //
   // 'new' that can't throw
   //
   try
      {
      p = new(nothrow) int;
      }
   catch(bad_alloc &)
      {
      cout << "this line should never appear";
      }
   //
   return 0;
   }
```

注意两个 new 表达式的重要不同之处：

```
p = new int;
```

在分配失败时抛 std::bad_alloc，而
p = new(nothrow) int;
在分配失败时不抛异常，它返回 NULL（就象 malloc 和 C++黑暗年代的 new）。

如果你不喜欢 nothrow 的语法，或你的编译器不支持，你可以这样达到同样效果：

```
#include <new>

//
// function template emulating 'new(std::nothrow)'
//
template<typename T>
T *new_nothrow() throw()
   {
   T *p;
   try
      {
      p = new T;
      }
   catch(std::bad_alloc &)
      {
      p = NULL;
      }
   return p;
   }

//
// example usage
//
int main()
   {
   int *p = new_nothrow<int>(); // equivalent to 'new(nothrow) int'
   return 0;
   }
```

这个模板函数与它效仿的 new(nothrow)表达式同有一个潜在的异常安全漏洞。现在，我将它作为习题留给你去找出来。（恐怕没什么用的提示：和 placement delete 有关。）

5.8 小结

new 和 delete 是怪兽。和 typeid 一起，它们是 C++中仅有的会调用标准运行库中函数的关键字。即使程序除了 main 外不明确调用或定义任何函数，new 和 delete 语句的出现就会使程序调用运行库。如我在这儿所示范的，调用运行库将经常可能抛异常或处理异常。

本篇的例程中的代码和注释是用于我对 C++标准的解释的。不幸的是，如我以前所说，Microsoft 的 Visual C++经常不遵守 C++标准。在下一篇中，我将揭示 Visual C++的运行库对 new 和 delete 的支持在什么地方背离了 C++标准。我将特别注意在对异常的支持上的背离，并且将展示怎么绕过它们。

回到目录

6. Microsoft 对于<new>的实现版本中的异常处理

上次，我讲述了标准运行库头文件<new>中申明的 12 个全局函数中的异常行为。这次我将开始讨论 Microsoft 对这些函数的实现版本。

在 Visual C++ 5 中，标准运行库头文件<new>提供了这些申明：

```
namespace std
   {
   class bad_alloc;
   struct nothrow_t;
   extern nothrow_t const nothrow;
   };

void *operator new(size_t) throw(std::bad_alloc);
void operator delete(void *) throw();

void *operator new(size_t, void *);

void *operator new(size_t, std::nothrow_t const &) throw();
```

和在第五部分中讲述的标准所要求的相比，Microsoft 的<new>头文件版本缺少：

- 所有（三种）形式的 operator new[]
- 所有（三种）形式的 operator delete[]
- Placement operator delete(void *, void *)
- Placement operator delete(void *, std::nothrow_t const &)

并且，虽然运行库申明了 operator new 抛出 std::bad_alloc，但函数的行为并不符合标准。

如果你使用 Visaul C++ 6，<new>头文件有同样的缺陷，只是它申明了 operator delete(void *, void *)。

6.1 数组

Visual C++在标准运行库的实行中没有定义 operator new[]和 operator delete[]形式的版本。幸好，你可以构建自己的版本：

```
#include <stdio.h>

void *operator new(size_t)
   {
   printf("operator new\n");
   return 0;
   }

void operator delete(void *)
   {
```

```
   printf("operator delete\n");
   }

void *operator new[](size_t)
   {
   printf("operator new[]\n");
   return 0;
   }

void operator delete[](void *)
   {
   printf("operator delete[]\n");
   }

int main()
   {
   int *p;
   p = new int;
   delete p;
   p = new int[10];
   delete[] p;
   }

/* When run should yield

      operator new
      operator delete
      operator new[]
      operator delete[]
*/
```

为什么 Visual C++的标准运行库缺少这些函数？我不能肯定，猜想是“向后兼容”吧。

operator new[]和 operator delete[]加入 C++标准比较晚，并且许多年来编译器们还不支持它，所有支持分配用户自定义对象的编译器都定义了 operator new 和 operator delete，并且即使是分配数组对象也将调用它们。

如果一个以前不支持 operator new[]和 operator delete[]的编译器开始支持它们时，用户自定义的全局 operator new 和 operator delete 函数将不再在分配数组对象时被调用。程序仍然能编译和运行，但行为却变了。程序员甚至没法知道变了什么，因为编译器没有报任何错。

6.2 无声的变化

这些无声的变化给写编译器的人（如 Microsoft）出了个难题。要知道，C++标准发展了近 10 年。在此期间，编译器的卖主跟踪标准的变化以确保和最终版本的最大程度兼容。同时，用户依赖于当前可用的语言特性，即使不能确保它们在标准化的过程中得以幸存。

如果标准的一个明显变化造成了符合前标准的程序的行为的悄然变化，编译器的卖主有三种选择：

1. 坚持旧行为，不理符合新标准的代码
2. 改到新行为，不理符合旧标准的代码
3. 让用户指定他们想要的行为

在此处的标准运行库提供 operator new[]和 operator delete[]的问题上，Micrsoft 选择了 1。我自己希望他们选择 3，对这个问题和其它所有 Visual C++不符合标准之处。他们可以通过#pragmas、编译选项或环境变量来判断用户的决定的。

Visual C++长期以来通过形如/Za 的编译开关来实行选择 3，但这个开关有一个未公开的行为：它关掉了一些标准兼容的特性，然后打开了另外一些。我期望的（想来也是大部分人期望的）是一个完美的调节方法来打开和关闭标准兼容的特性！

（在这个 operator new[]和 operator delete[]的特例中，我建议你开始使用容器类（如 vector）来代替数组，但这是另外一个专栏的事情了。 ）

6.3 异常规格申明

Microsoft 的<new>头文件正确地申明了非 placement 的 operator new：

```
void *operator new(std::size_t) throw(std::bad_alloc);
```

你可以定义自己的 operator new 版本来覆盖运行库的版本，你可能写成：

```
void *operator new(std::size_t size) throw(std::bad_alloc)
   {
   void *p = NULL;
   // ... try to allocate '*p' ...
   if (p == NULL)
      throw std::bad_alloc();
   return p;
   }
```

如果你保存上面的函数，并用默认选项编译，Visual C++不会报错。但，如果你将警告级别设为 4，然后编译，你将遇到这个信息：

```
warning C4290: C++ Exception Specification ignored
```

那么好，如果你自己的异常规格申明不能工作，肯定，运行库的版本也不能。保持警告级别为 4，然后编译：

```
#include <new>
```

我们已经知道，它申明了一个和我们的程序同样的异常规格的函数。

奇怪啊，奇怪！编译器没有警告，即使在级别 4！这是否意味着运行库的申明有些奇特属性而我们的没有？不，它事实上意味着 Micorsoft 的欺骗行为：

- <new>包含了标准运行库头文件<exception>。
- <exception>包含了非标头文件 xstddef。
- xstddef 包含了另一个非标头文件 yvals.h。
- yvals.h 包含了指令#pragma warning(disable:4290)。
- #pragma 关闭了特定的级别 4 的警告，我们在自己的代码中看到的那条。

结论：Visual C++在编译期检查异常规格申明，但在运行期忽略它们。你可以给函数加上异常申明（如 throw(std::bad_alloc)），编译器会正确地分析它们，但在运行期这个申明没有效果，就象根本没有写过。

6.4 怎么会这样

在这个专栏的第三部分，我讲述了异常规格申明的形式，却没有解释其行为和效果。Visual C++对异常规格申明的不完全支持给了我一个极好的机会来解释它们。

异常规格申明是函数及其调用者间契约的一部分。它完整列举了函数可能抛出的所有异常。（用标准中的说法，被称为函数 “允许”特定的异常。）

换句话说就是，函数不允许（承诺不抛出）其它任何不在申明中的异常。如果申明有但为空，函数根本不允许任何异常；相反，如果没有异常规格申明，函数允许任何异常。

除非函数与调用者间的契约是强制性的，否则它根本就不值得写出来。于是你可能会想，编译器应该在编译时确保函数没有撒谎：

```
void f() throw() // 'f' promises to throw no exceptions...
   {
   throw 1; // ... yet it throws one anyway!
   }
```

惊讶的是，它在Visual C++中编译通过了。

不要认为Visual c++有病，这个例子可以用任何兼容C++的编译器编译通过。我从标准（sub clause 15.4p10）中引下来的：

C++的实现版本不该拒绝一个表达式，仅仅是因为它抛出或可能抛出一个其相关函数所不允许的异常。例如：

```
extern void f() throw(X, Y);

void g() throw(X)
   {
   f(); //OK
   }
```

调用f()的语句被正常编译，即使当调用时f()可能抛出g()不允许的异常Y。

是不是有些特别？那么好，如果编译器不强制这个契约，将发生什么？

6.5 运行期系统

如果函数抛出了一个它承诺不抛的异常，运行期系统调用标准运行库函数unexpected()。运行库的缺省unexpected()的实现是调用terminate()来结束函数。你可以调用set_unexpected()函数安装新的unexpected()处理函数而覆盖其缺省行为。

这只是理论。但如前面的Visual C++警告所暗示，它忽略了异常规格申明。因此，Visual C++运行期系统不会调用unexpected()函数，当一个函数违背其承诺时。

要试一下你所喜爱的编译器的行为，编译并运行下面这个小程序：

```
#include <exception>
#include <stdio.h>
using namespace std;

void my_unexpected_handler()
   {
   throw bad_exception();
```

```
   }

void promise_breaker() throws()
   {
   throw 1;
   }

int main()
   {
set_unexpected(my_unexpected_handler);
   try
      {
      promise_breaker();
      }
   catch(bad_exception &)
      {
printf("Busted!");
      }
   catch(...)
      {
printf("Escaped!");
      }
   return 0;
   }
```

如果程序输出是：

```
Busted!
```

则，运行期系统完全捕获了违背异常规格申明的行为。反之，如果输出是：

```
Escaped!
```

则运行期系统没有捕获违背异常规格申明的行为。

在这个程序里，我安装了 my_unexepected_handler()来覆盖运行库的缺省 unexpected()处理函数。这个自定义的处理函数抛出一个 std::bad_exception 类型的异常。此类型有特别的属性：如果 unexpected()异常处理函数抛出此类型，此异常能够被（外面）捕获，程序将继续运行而不被终止。在效果上，这个 bad_exception 对象代替了原始的抛出对象，并向外传播。

这是假定了编译器正确地检测了 unexpected 异常，在 Visual C++中，my_unexpected_handler() 没有并调用，原始的 int 型异常抛到了外面，违背了承诺。

6.6 模拟异常规格申明

如果你愿意你的设计有些不雅，就可以在 Visual C++下模拟异常规格申明。考虑一下这个函数的行为：

```
void f() throw(char, int, long)
   {
   // ... whatever
```

```
}
```

假设一下会发生什么？

- 如果 f()没有发生异常，它正常返回。
- 如果 f()发生了一个允许的异常，异常传到 f()外面。
- 如果 f()发生了其它（不被允许）的异常，运行期系统调用 unexpected()函数。

要在 Visual C++下实现这个行为，要将函数改为：

```
void f() throw(char, int, long)
   {
   try
      {
      // ... whatever
      }
   catch(char)
      {
      throw;
      }
   catch(int)
      {
      throw;
      }
   catch(long)
      {
      throw;
      }
   catch(...)
      {
      unexpected();
      }
   }
```

Visual C++一旦开始正确支持异常规格申明，它的内部代码必然象我在这儿演示的。这意味着异常规格申明将和 try/catch 块一样导致一些代价，就象我在第四部分中演示的。

因此，你应该明智地使用异常规格申明，就象你使用其它异常部件。任何时候你看到一个异常规格申明，你应该在脑子里将它们转化为 try/catch 队列以正确地理解其相关的代价。

6.7 预告

placement delete 的讨论要等到下次。将继续讨论更多的通行策略来异常保护你的设计。

回到目录

7. 部分构造及 placement delete

讨论在一般情况下的部分构造、动态生成对象时的部分构造，以及用 placement delete 来解决部分构造问题。

C++标准要求标准运行库头文件<new>提供几个 operator delete 的重载形式。在这些重载形式中，Visual C++ 6 缺少：

- void operator delete(void *, void *)

而 Visual C++ 5 缺少：

- void operator delete(void *, void *)
- void operator delete(void *, std::nothrow_t const &)

这些重载形式支持 placement delete 表达式，并解决了一个特殊问题：释放部分构造的对象。在这次和接下来一次，我将给出一般情况下的部分构造、动态生成对象时的部分构造，以及用 placement delete 来解决部分构造问题的例子。

7.1 部分构造

看这个例子：

```
// Example 1

#include <iostream>

class A
   {
public:
   A()
      {
      throw 0;
      }
   };

int main()
   {
   try
      {
      A a;
      }
   catch(...)
      {
      std::cout <<"caught exception" << std::endl;
      }
   return 0;
   }
```

因为 A 的构造函数抛出了一个异常，a 对象没有完全构造。在这个例子中，没有构造函

数有可见作用：因为 A 没有子对象，构造函数实际上没有任何操作。但，考虑这样的变化：

```
// Example 2

#include <iostream>

class B
  {
public:
  B()
     {
     throw 0;
     }
  };

class A
   {
private:
  B const b;
   };

// ... main same as before ...
```

现在，A 的构造函数不是无行为的，因为它构造了一个 B 成员对象，而它里面会抛异常。程序对这个异常作出什么反应？

从 C++标准中摘下了四条（稍作了简化）原则：

- 一个对象被完全构造，当且仅当它的构造函数已经完全执行，而它的析构函数还没开始执行。
- 如果一个对象包含子对象，包容对象的构造函数只有在所有子对象被完全构造后才开始执行。
- 一个对象被析构，当且仅当它被完全构造。
- 对象按它们被构造的反序进行析构。

因为抛出了一个异常，B::B 没有被完全执行。因此，B 的对象 A::b 既没有被完全构造也没有被析构。

要证明这点，跟踪相应的类成员：

```
// Example 3

#include <iostream>

class B
   {
public:
   B()
      {
     std::cout << "B::B enter" << std::endl;
      throw 0;
```

```
    std::cout << "B::B exit" << std::endl;
     }
  ~B()
     {
     std::cout << "B::~B" << std::endl;
     }
   };

class A
   {
public:
  A()
     {
     std::cout << "A::A" << std::endl;
     }
  ~A()
     {
      std::cout << "A::~A"<< std::endl;
     }
private:
   B const b;
   };

// ...  main same as before ...
```

当运行时，程序将只输出

```
B::B enter
caught exception
```

从而显示出对象 a 和 b 既没有完全构造也没有析构。

7.2 多对象

使例子变得更有趣和更有说明力，把它改得允许部分（不是全部）对象被完全构造：

```
// Example 4

#include <iostream>

class B
   {
public:
   B(int const ID) : ID_(ID)
      {
      std::cout << ID_ << " B::B enter" <<std::endl;
     if (ID_ > 2)
         throw 0;
```

```
      std::cout << ID_ << " B::B exit" <<std::endl;
      }
   ~B()
      {
      std::cout << ID_ << " B::~B" <<std::endl;
      }
private:
  int const ID_;
   };

class A
   {
public:
   A() : b1(1), b2(2), b3(3)
      {
      std::cout <<"A::A" << std::endl;
      }
   ~A()
      {
      std::cout <<"A::~A" << std::endl;
      }
private:
   B const b1;
   B const b2;
   B const b3;
   };

// ...  main same asbefore ...
```

注意 B 的构造函数现在接受一个对象 ID 值的参数。用它作 B 的对象的唯一标记并决定对象是否完全构造。大部分跟踪信息以这些 ID 开头，显示为：

```
1 B::B enter
1 B::B exit
2 B::B enter
2 B::B exit
3 B::B enter
2 B::~B
1 B::~B
caught exception
```

b1 和 b2 完全构造而 b3 没有。所以，b1 和 b2 被析构而 b3 没有。此外，b1 和 b2 的析构按其构造的反序进行。最后，因为一个子对象（b3）没有完全构造，包容对象 a 也没有完全构造和析构。

7.3 动态分配对象

将类 A 改为其成员变量是动态生成的：

```
// Example 5

#include <iostream>

// ... class B same as before ...

class A
   {
public:
   A() : b1(new B(1)), b2(new B(2)), b3(new B(3))
      {
      std::cout <<"A::A" << std::endl;
      }
   ~A()
      {
     delete b1;
     delete b2;
     delete b3;
      std::cout <<"A::~A" << std::endl;
      }
private:
   B * const b1;
   B * const b2;
   B * const b3;
   };

// ... main same as before ...
```

这个形式符合 C++习惯用法：在包容对象的构造函数里分配成员变量，并对其填充数据，然后在包容对象的析构函数里释放它们。

编译并运行例 5。输出是：

```
1 B::B enter
1 B::B exit
2 B::B enter
2 B::B exit
3 B::B enter
caught exception
```

其结果与例 4 相似，但有一个巨大的不同：因为~A 没有被执行，其中的 delete 语句也就没有执行，被成功分配的*b1 和*b2 的析构函数也没有调用。例四中的不妙状况（三个对象析构了两个）现在更差了（三个对象一个都没有析构）。

实际上，没有比这更坏的了。记住，delete b1 语句有两个作用：

- 调用*b1 的析构函数~b。
- 调用 operator delete 释放*b1 所占有的内存。

所以我们不光是遇到~B 没有被调用所导致的问题，还有每个 B 对象造成的内存泄漏问题。这不是件好事。

B 对象是 A 私有的，它们是实现细节，对程序的其它部分是不可见的。用动态生成 B 的子对象来代替自动生成 B 的子对象不该改变程序的外在行为，这表明了我们的例子在设计上的缺陷。

7.4 析构动态生成的对象

为了最接近例 4 的行为，我们需要在任何情况强迫 delete 语句的执行。将它们放入~A 明显不起作用。我们需要找个能起作用的地方，我们知道它能被执行的地方。跳入脑海的解决方法中，最优雅的方法来自于 C++标准运行库：

```
// Example 6

#include <iostream>
#include <memory>

// ... class B same as before ...

class A
   {
public:
   A() : b1(new B(1)), b2(new B(2)), b3(new B(3))
      {
      std::cout << "A::A" << std::endl;
      }
   ~A()
      {
      std::cout << "A::~A" << std::endl;
      }
private:
  std::auto_ptr<B> const b1;
  std::auto_ptr<B> const b2;
  std::auto_ptr<B> const b3;
   };

// ... main same as before ...
```

auot_ptr 读作“auto-pointer”。如名所示，auoto-pointer 表现为通常的指针和自动对象的混合体。

std::auto_ptr 是在<memory>中申明的类模板。一个 std::auto_ptr<B>类型的对象的表现非常象一个通常的 B*类型对象，关键的不同是：auto_ptr 是一个实实在在的类对象，它有析构函数，而这个析构函数将在 B*所指对象上调用 delete。最终结果是：动态生成的 B 对象如同是个自动 B 对象一样被析构。

可以把一个 auto_ptr<B>对象当作对动态生成的 B 对象的简单包装。在包装消失（析构）时，它也将被包装对象带走了。要实际看这个魔术戏法，编译并运行例 6。结果是：

```
1 B::B enter
1 B::B exit
2 B::B enter
2 B::B exit
3 B::B enter
2 B::~B
1 B::~B
caught exception
```

Bingo! 输出和例 4 相同。

你可能会奇怪为什么没有为 b3 调用~B。这表明了 auto_ptr 包装上的失败？根本不是。我们所读过的规则还在起作用。对 b3 进行的构造函数的调用接受了 new B(3)传过来的参数。于是发生了一个异常终止了 b3 的构造。因为 b3 没有完全构造，它同样不会析构。

藏在 atuo-pointer 后面的想法没有新的地方；string 对象实际上就是 char 数组的 auto-pointer 型包装。虽然如此，我仍然期望有一天我能更详细的讨论 auto_ptr 及其家族，目前只要把 auto_ptr 当作一个保证发生异常时能析构动态生成的对象的简单方法。

7.5 预告

既然 b3 的析构函数没有被调用，也就没有为其内存调用 delete。如前面所见，被包装的 B 对象受到两个影响：

- 析构函数~B 没有被调用。这是意料中的甚至是期望中的，因为 B 对象在先前没有完全构造。
- 内存没有被通过 operator delete 释放。不管是不是意料中的，它绝不是期望中的，因为 B 对象所占用的内存被分配了，即使 B 对象没有在此内存中完全构造。

我需要 operator delete 被调用，即使~B 没有被调用。要实现这点，编译器必须在脱离 delete 语句的情况下调用 operator delete。因为我知道 b3 是我的例子中的讨厌对象，我可以显式地为 b3 的内存调用 operator delete；但要知道这只是教学程序，通常情况下我们不能预知哪个构造函数将失败。

不，我们所需要的是编译器检测到动态生成对象时的构造函数失败时隐含调用 operator delete 来释放对象占用的内存。这有些效仿编译器在自动对象构造失败时的行为：对象的内存如同程序体中的无用单元一样，是可回收的。

幸好，它有个大喜结局。要看这个结局，需到下回。在下回结束时，我将揭示 C++语言如何提供了这个完美特性，为什么标准运行库申明了 placement operator delete，以及为什么你可能想在自己的库或类中做同样的事。

回到目录

8. 自动删除，类属 new 和 delete、placement new 和 placement delete

在上次结束时，我期望道：当一个新产生的对象在没有完全构造时，它所占用的内存能自动释放。很幸运，C++标准委员会将这个功能加入到了语言中（而不幸的是，这个特性加得太晚了，许多编译器还不支持它）。Visual C++ 5 和 6 都支持这个“自动删除”特性（但，如我们将要看到的，Visual C++ 5 的支持是不完全的）。

8.1 自动删除

要实际验证它，在上次的例 6 中增加带跟踪信息的 operator new 和 operator delete 函数：

```
// Example 7

#include <iostream>
#include <memory>
#include <stdio.h>
#include <stdlib.h>

void *operator new(size_t const n)
   {
   printf("   ::operator new\n");
   return malloc(n);
   }

void operator delete(void *const p)
   {
   std::cout << "   ::operator delete" << std::endl;
   free(p);
   }

class B
   {
public:
   B(int const ID) : ID_(ID)
      {
      std::cout << ID_ << " B::B enter" << std::endl;
      if (ID_ > 2)
         {
         std::cout << std::endl;
         std::cout << "  THROW" << std::endl;
         std::cout << std::endl;
         throw 0;
         }
      std::cout << ID_ << " B::B exit" << std::endl;
      }
   ~B()
      {
      std::cout << ID_ << " B::~B" << std::endl;
      }
private:
   int const ID_;
   };
```

```
class A
   {
public:
   A() : b1(new B(1)), b2(new B(2)), b3(new B(3))
      {
      std::cout << "  A::A" << std::endl;
      }
   ~A()
      {
      std::cout << "  A::~A" << std::endl;
      }
private:
   std::auto_ptr<B> const b1;
   std::auto_ptr<B> const b2;
   std::auto_ptr<B> const b3;
   };

int main()
   {
   try
      {
      A a;
      }
   catch(...)
      {
      std::cout << std::endl;
      std::cout << "  CATCH" << std::endl;
      std::cout << std::endl;
      }
   return 0;
   }
```

程序将用我们自己的 operator new 和 operator delete 代替标准运行库提供的版本。这样，我们将能跟踪所有的动态创建对象时的分配和释放内存操作。（我同时小小修改了其它的跟踪信息，以便输出信息更容易读。）

注意，我们的 operator new 调用了 printf 而不是 std::cout。本来，我确实使用了 std::cout，但程序在运行库中产生了一个无效页错误。调试器显示运行库在初始化 std::cout 前调用了 operator new,而 operator new 又试图调用还没有初始化的 std::cout，程序于是崩溃了。

我在 Visual C++ 6 中运行程序，得到了头大的输出：

```
::operator new
::operator new
::operator new
::operator new
```

```
   ::operator new
   ::operator new
   ::operator delete
   ::operator delete
   ::operator new
   ::operator new
   ::operator new
   ::operator new
   ::operator new
   ::operator new
   ::operator delete
   ::operator delete
1 B::B enter
1 B::B exit
   ::operator new
2 B::B enter
2 B::B exit
   ::operator new
3 B::B enter

  THROW

   ::operator delete
2 B::~B
   ::operator delete
1 B::~B
   ::operator delete

  CATCH

   ::operator delete
   ::operator delete
   ::operator delete
   ::operator delete
   ::operator delete
Blech.
```

我无法从中分辨出有用的信息。原因很简单：我们的代码，标准运行库的代码，以及编译器暗中生成的代码都调用了 operator new 和 operator delete。我们需要一些方法来隔离出我们感兴趣的调用过程，并只输出它们的跟踪信息。

8.2 类属 new 和 delete

C++又救了我们。不用跟踪全局的 operator new 和 operator delete，我们可以跟踪其

类属版本。既然我们感兴趣的是 B 对象的分配和释放过程，我们只需将 operator new 和 operator delete 移到类 B 中去：

```
// Example 8

#include <iostream>
#include <memory>

class B
   {
public:
   void *operator new(size_t const n)
      {
      std::cout << "  B::operator new" << std::endl;
      return ::operator new(n);
      }
   void operator delete(void *const p)
      {
      std::cout << "  B::operator delete" << std::endl;
      operator delete(p);
      }
   // ... rest of class B unchanged
};

// ... class A and main unchanged
```

编译器将为 B 的对象调用这些函数，而为其它对象的分配和释放调用标准运行库中的函数版本。

通过在你自己的类这增加这样的局部操作函数，你可以更好的管理动态创建的此类型对象。例如，嵌入式系统的程序员经常在特殊映射的设备或快速内存中分配某些对象，通过其类型特有的 operator new 和 operator delete，可以控制如何及在哪儿分配这些对象。

对我们的例子，特殊的堆管理是没必要的。因此，我在类属 operator new 和 operator delete 中调用了其全局版本而不再是 malloc 和 free，并去除了对头文件<stdlib.h>的包含。这样，所有对象的分配和释放的实际语义保持了一致。

同时，因为我们的 operator new 不在在全局范围内，它不会被运行库在构造 std::cout 前调用，于是我可以在其中安全地调用 std::cout 了。因为不再调用 printf，我也去掉了<stdio.h>。

编译并运行例 8。将发现输出信息有用多了：

```
  B::operator new
1 B::B enter
1 B::B exit
  B::operator new
2 B::B enter
2 B::B exit
  B::operator new
3 B::B enter
```

```
  THROW

  B::operator delete
2 B::~B
  B::operator delete
1 B::~B
  B::operator delete

  CATCH
```

三个 B::operator new 的跟踪信息对应于 a.b1、a.b2 和 a.b3 的构造。其中，a.b1 和 a.b2 被完全构造（它们的构造函数都进入并退出了），而 a.b3 没有（它的构造函数只是进入了而没有退出）。注意这个：

```
3 B::B enter

  THROW

  B::operator delete
```

它表明，调用 a.b3 的构造函数，在其中抛出了异常，然后编译器自动释放了 a.b3 占用的内存。接下来的跟踪信息：

```
2 B::~B
  B::operator delete
1 B::~B
  B::operator delete
```

表明被完全构造的对象 a.b2 和 a.b1 在释放其内存前先被析构了。

结论：所有完全构造的对象的析构函数被调用，所有对象的内存被释放。

8.3 Placement new

例 8 使用了“普通的”非 Placement new 语句来构造三个 B 对象。现在考虑这个变化：

```
// Example 9

// ... preamble unchanged

class B
   {
public:
   void *operator new(size_t const n, int)
      {
      std::cout << "  B::operator new(int)" << std::endl;
      return ::operator new(n);
      }
   // ... rest of class B unchanged
};
```

```
class A
   {
public:
   A() : b1(new(0) B(1)), b2(new(0) B(2)), b3(new(0) B(3))      {
      std::cout << "  A::A" << std::endl;
      }
   // ... rest of class A unchanged
   };


// ... main unchanged
```

这个 new 语句

```
new(0) B(1)
```

有一个 placement 参数 0。因为参数的类型是 int，编译器需要 operator new 的一个接受额外 int 参数的重载版本。我已经增加了一个满足要求的 B::operator new 函数。这个函数实际上并不使用这个额外参数，此参数只是个占位符，用来区分 placement new 还是非 placement new 的。

因为 Visual C++ 5 不完全支持 placement new 和 placement delete，例 9 不能在其下编译。程序在 Visual C++ 6 下能编译，但在下面这行上生成了三个 Level 4 的警告：

```
A() : b1(new(0) B(1)), b2(new(0) B(2)), b3(new(0) B(3))
```

内容都是：

```
'void *B::operator new(unsigned int, int)':
   no matching operator delete found;
   memory will not be freed if initialization
   throws an exception
```

想知道编译器为什么警告，运行程序，然后和例 8 比较输出：

```
  B::operator new(int)
1 B::B enter
1 B::B exit
  B::operator new(int)
2 B::B enter
2 B::B exit
  B::operator new(int)
3 B::B enter

  THROW

2 B::~B
  B::operator delete
1 B::~B
  B::operator delete

  CATCH
```

输出是相同的，只一个关键不同：
3 B::B enter

THROW
和例 8 一样的是，a.b3 的构造函数进入了并在其中抛出了异常；但和例 8 不同的是，a.b3 的内存没有自动删除。我们应该留意编译器的警告的！

8.4 最后，Placement delete!

想要“自动删除”能工作，一个匹配抛异常的对象的 operator new 的 operator delete 的重载版本必须可用。摘自 C++标准 (subclause 5.3.4p19, "New")：

如果参数的数目相同并且除了第一个参数外其类型一致（在作了参数的自动类型转换后），一个 placement 的释放函数与一个 placement 的分配函数相匹配。所有的非 palcement 的释放函数匹配于一个非 placement 的分配函数。如果找且只找到一个匹配的释放函数，这个函数将被调用；否则，没有释放函数被调用。

因此，对每个 placement 分配函数

```
void operator new(size_t, P2, P3, ..., Pn);
```

都有一个对应的 placement 释放函数

```
void *operator delete(void *, P2, P3, ..., Pn);
```

这里

```
P2, P3, ..., Pn
```

一般是相同的参数队列。我说“一般”是因为，根据标准的说法，可以对参数进行一些转换。再引于标准(subclause 8.3.5p3, "Functions")，基于可读性稍作了修改：

在提供了参数类型列表后，将对这些类型作一些转换以决定函数的类型：

- **所有参数类型的 const/volatile 描述符修饰将被删除。这些 cv 描述符修饰只影响形参在函数体中的定义，不影响函数本身的类型。**
  例如：类型
  void (*)(const int)
  变为
  void (*)(int)
- **如果一个存储类型描述符修饰了一个参数类型，此描述符被删除。这存储类型描述符修饰只影响形参在函数体中的定义，不影响函数本身的类型。**
  例如：
  register char *
  变成
  char *

转换后的参数类型列表才是函数的参数类型列表。

顺便提一下，这个规则同样影响函数的重载判断，signatures 和 name mangling。基本上，函数参数上的 cv 描述符和存储类型描述符的出现不影响函数的身份。例如，这意味着下列所有申明引用的是同一个函数的定义。

- void f(int)

- void f(const int)
- void f(register int)
- void f(auto const volatile int)

增加匹配于我们的 placement operator new 的 placement operator delete 函数：

```
// Example 10

// ... preamble unchanged

class B
   {
public:
    void operator delete(void *const p, int)
      {
      std::cout << "  B::operator delete(int)" << std::endl;
      ::operator delete(p);
      }
   // ... rest of class B unchanged
};

// ... class A and main unchanged
```

然后重新编译并运行。输出是：

```
  B::operator new(int)
1 B::B enter
1 B::B exit
  B::operator new(int)
2 B::B enter
2 B::B exit
  B::operator new(int)
3 B::B enter

  THROW

  B::operator delete(int)
2 B::~B
  B::operator delete
1 B::~B
  B::operator delete

  CATCH
```

和例 8 非常相似，每个 operator new 匹配一个 operator delete。

一个可能奇怪的地方：所有 B 对象通过 placement operator new 分配，但不是全部通过 placement operator delete 释放。**记住，placement operator delete 只（在 plcaement operator new 失败时）被调用于自动摧毁部分构造的对象。**完全构造的对象将通过 delete

语句手工摧毁，而 delete 语句调用非 placement operator delete。（WQ 注：没有办法调用 placement delete 语句，只能调用 plcaement operator delete 函数，见 9.2。）

8.5 光阴似箭

在第九部分，我将展示 placement delete 是多么地灵巧（远超过现在展示的），但有小小的隐瞒和简化。并示范一个新的机制来在构造函数（如 A::A）中更好地容忍异常。
回到目录

9. placement new 和 placement delete，及处理构造函数抛出的异常

当被调用了来清理部分构造时，operator delete 的第一个 void *参数带的是对象的地址（刚刚由对应的 operator new 返回的）。operator delete 的所有额外 placement 参数都和传给 operator new 的相应参数的值相匹配。

在代码里，语句

```
p = new(n1, n2, n3) T(c1, c2, c3);
```

的效果是

```
p = operator new(sizeof(T), n1, n2, n3);
T(p, c1, c2, c3);
```

如果 T(p, c1, c2, c3)构造函数抛出了一个异常，程序暗中调用

```
operator delete(p, n1, n2, n3);
```

原则：当释放一个部分构造的对象时，operator delete 从原始的 new 语句知道上下文。

9.1 Placement operator delete 的参数

要证明这点，增强我们的例子来跟踪相应的参数值：

```
// Example 11

#include <iostream>
#include <memory>

class B
   {
public:
   B(int const ID) : ID_(ID)
      {
      std::cout << ID_ << " B::B enter" << std::endl;
      if (ID_ > 2)
         {
         std::cout << std::endl;
         std::cout << "  THROW" << std::endl;
         std::cout << std::endl;
         throw 0;
```

```
        }
    std::cout << ID_ << " B::B exit" << std::endl;
    }
~B()
    {
    std::cout << ID_ << " B::~B" << std::endl;
    }
//
// non-placement
//
void *operator new(size_t const n)
    {
    void *const p = ::operator new(n);
    std::cout << "  B::operator new(" << n <<
          ") => " << p << std::endl;
    return p;
    }
void operator delete(void *const p)
    {
    std::cout << "  B::operator delete(" << p <<
          ")" << std::endl;
    ::operator delete(p);
    }
//
// placement
//
void *operator new(size_t const n, int const i)
    {
    void *const p = ::operator new(n);
    std::cout << "  B::operator new(" << n <<
          ", " << i << ") => " << p << std::endl;
    return p;
    }
void operator delete(void *const p, int const i)
    {
    std::cout << "  B::operator delete(" << p <<
          ", " << i << ")" << std::endl;
    ::operator delete(p);
    }
private:
int const ID_;
};

class A
```

```
   {
public:
   A() : b1(new(11) B(1)), b2(new(22) B(2)), b3(new(33) B(3))
      {
      std::cout << "  A::A" << std::endl;
      }
   ~A()
      {
      std::cout << "  A::~A" << std::endl;
      }
private:
   std::auto_ptr<B> const b1;
   std::auto_ptr<B> const b2;
   std::auto_ptr<B> const b3;
   };

int main()
   {
   try
      {
      A a;
      }
   catch(...)
      {
      std::cout << std::endl;
      std::cout << "  CATCH" << std::endl;
      std::cout << std::endl;
      }
   return 0;
   }
```

用 Visual C++ 6 编译并运行。在我的机器上的输出是：

```
  B::operator new(4, 11) => 007E0490
1 B::B enter
1 B::B exit
  B::operator new(4, 22) => 007E0030
2 B::B enter
2 B::B exit
  B::operator new(4, 33) => 007E0220
3 B::B enter

  THROW

  B::operator delete(007E0220, 33)
2 B::~B
```

```
  B::operator delete(007E0030)
1 B::~B
  B::operator delete(007E0490)

  CATCH
```

注意这些数字：

- 4 是每个被分配的 B 对象的大小的字节数。这个值在不同的 C++实现下差异很大。
- 如 007E0490 这样的值是 operator new 返回的对象的地址，作为 this 指针传给 T 的成员函数的，并作为 void *型指针传给 operator delete。你看到的值几乎肯定和我的不一样。
- 11，22 和 33 是最初传给 operator new 的额外 placement 参数，并在部分构造时传给相应的 placement operator delete。

9.2 手工调用 operator delete

所有这些 operator new 和 operator delete 的自动匹配是很方便的，但它只在部分构造时发生。对通常的完全构造，operator delete 不是被自动调用的，而是通过明确的 delete 语句间接调用的：

```
p = new(1) B(2); // calls operator new(size_t, int)
// ...
delete p;        // calls operator delete(void *)
```

这样的顺序其结果是调用 placement operator new 和非 placement operator delete，即使你有对应的（placement）operator delete 可用。

虽然你很期望，但你不能用这个方法强迫编译器调用 placement operator delete：

```
delete(1) p; // error
```

而必须手工写下 delete 语句将要做的事：

```
p->~B();                  // call *p's destructor
B::operator delete(p, 1); // call placement
                          //  operator delete(void *, int)
```

要和自动调用 operator delete 时的行为保持完全一致，你必须保存通过 new 语句传给 operator new 的参数，并将它们手工传给 operator delete。

```
p = new(n1, n2, n3) B;
// ...
p->~B();
B::operator delete(p, n1, n2, n3);
```

9.3 其它非 placement delete

贯穿整个这个专题，我说了 operator new 和 operator delete 分类如下：

函数对

- void *operator new(size_t)
- void operator delete(void *)

是非 placement 分配和释放函数。
所有如下形式的函数对

- void *operator new(size_t, P1, ..., Pn)
- void operator delete(void *, P1, ..., Pn)

是 placement 分配和释放函数。
我这样说是因为简洁，但我现在必须承认撒了个小谎：
void operator delete(void *, size_t)
也可以是一个非 placement 释放函数而匹配于
void *operator new(size_t)
虽然它有一个额外参数。如你所猜想，operator delete 的 size_t 参数带的是传给 operator new 的 size_t 的值。和其它额外参数不同，它是提供完全构造的对象用的。
在我们的例子中，将这个 size_t 参数加到非 placement operator delete 上：

```
// Example 12

// ... preamble unchanged

class B
   {
   void operator delete(void * const p, size_t const n)
      {
      std::cout << "  B::operator delete(" << p <<
            ", " << n << ")" << std::endl;
      ::operator delete(p);
      }
   // ... rest of class B unchanged
   };

// ... class A and main unchanged
```

The results:

```
  B::operator new(4, 11) => 007E0490
1 B::B enter
1 B::B exit
  B::operator new(4, 22) => 007E0030
2 B::B enter
2 B::B exit
  B::operator new(4, 33) => 007E0220
3 B::B enter

  THROW

  B::operator delete(007E0220, 33)
2 B::~B
  B::operator delete(007E0030, 4)
```

```
1 B::~B
  B::operator delete(007E0490, 4)

  CATCH
```

注意，为完全构造的对象，将额外的参数 4 提供给了 operator delete。

9.4 显而易见的矛盾

你可能奇怪：C++标准允许非 placement operator delete 自动知道一个对象的大小，却否定了 placement operator delete 可具有相同的能力。要想使它们保持一致，一个 placement 分配函数

```
void *operator new(size_t, P1, P2, P3)
```

应该匹配于这样一个 placement 释放函数

```
void operator delete(void *, size_t, P1, P2, P3)
```

但事实不是这样，这两个函数不匹配。为什么语言被这样设计？我猜有两个原因：效率和清晰。

大部分情况下，operator delete 不需要知道一个对象的大小；强迫函数任何时候都接受大小参数是低效的。并且，如果标准允许 size_t 参数可选，这样的含糊将造成：

```
void operator delete(void *, size_t, int)
```

在不同的环境下有不同的意义，决定它将匹配哪个：

```
void *operator new(size_t, int)
```

还是

```
void *operator new(size_t, size_t, int)
```

如果因下面的语句抛了个异常而被调用：

```
p = new(1) T; // calls operator new(size_t, int)
```

operator delete 的 size_t 参数将是 sizeof(T)；但如果是被调用时是

```
p = new(1, 2) T; // calls operator new(size_t, size_t, int)
```

operator delete 的 size_t 参数将是 new 语句的第一个参数值（这里是 1）。于是，operator delete 将不知道怎么解释它的 size_t 值。

我估计，你可能想知道是否非 placement 的函数

```
void operator delete(void *, size_t)
```

同时作为一个 placement 函数匹配于

```
void *operator new(size_t, size_t)
```

如果它被允许，operator delete 将遇到前面讲的同样问题。而不被允许的话， C++标准将需要其规则的一个例外。

我没发现规则的这样一个例外。我试过几个编译器，— including EDG's front end, my expert witness on such matters — 并认为：

void operator delete(void *, size_t)

实际上能同时作为一个 placement 释放函数和一个非 placement 释放函数。这是个重要的提醒。

如果你怀疑我，就将例 12 的 placement operator delete 移掉。

```
// Example 13

// ... preamble unchanged
```

```
class B
   {
//   void operator delete(void *const p, int const i)
//     {
//      std::cout << "  B::operator delete(" << p <<
//            ", " << i << ")" << std::endl;
//     ::operator delete(p);>
//      }
   // ... rest of class B unchanged
   };

// ... class A and main unchanged
```

现在，类里有一个 operator delete 匹配于两个 operator new。其输出结果和例 12 仍然相同。（WQ 注：结论是正确的，但不同的编译器下对例 12 到例 14 的反应相差很大，很是有趣!）

9.5 结束

两个最终要点：

- 贯穿我整个对::operator new 和 B::operator delete 的讨论，我总是将函数申明为非 static。通常这样的申明意味着有 this 指针存在，但这些函数的行为象它们没有 this 指针。实际上，在这些函数来试图引用 this，你将发现代码不能编译。不象其它成员函数，operator new 和 operator delete 始终是 static 的，即使你没有用 static 关键字。
- 无论我在哪儿提到 operator new 和 operator delete，你都可以用 operator new[] 和 operator delete[]代替。相同的模式，相同的规则，和相同的观察结果。（虽然 Visual C++标准运行库的<new>中缺少 operator new[]和 operator delete[]，编译器仍然允许你定义自己的数组版本。）

我想，这个结束了我对 plcement new 和 delete 及它们在处理构造函数抛出的异常时扮演的角色的解释。下次，我将介绍给你一个不同的技巧来容忍构造函数抛出的异常。

回到目录

10. 从私有子对象中产生的异常

几部分来，我一直展示了一些技巧来捕获从对象的构造函数中抛出的异常。这些技巧是在异常从构造函数中漏出来后处理它们。有时，调用者需要知道这些异常，但通常（如我所采用的例程中）异常是从调用者并不关心的私有子对象中爆发的。使得用户要关心“不可见”的对象表明了设计的脆弱。

在历史上，（可能抛异常）的构造函数的实现者没有简单而健壮的解决方法。看这个简单的例子：

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

```
class buffer
   {
public:
   explicit buffer(size_t);
   ~buffer();
private:
   char *p;
   };

buffer::buffer(size_t const count)
      : p(new char[count])
   {
   }

buffer::~buffer()
   {
   delete[] p;
   }

static void do_something_with(buffer &)
   {
   }

int main()
   {
   buffer b(100);
   do_something_with(b);
   return 0;
   }
```

buffer 的构造函数接受字符数目并从自由空间分配内存，然后初始化 buffer::p 指向它。如果分配失败，构造函数中的 new 语句产生一个异常，而 buffer 的用户（这里是 main 函数）必须捕获它。

10.1 try 块

不幸的是，捕获这个异常不是件容易事。因为抛出来自 buffer::buffer，所有 buffer 的构造函数的调用应该被包在 try 块中。没脑子的解决方法：

```
try
   {
   buffer b(count);
   }
catch (...)
```

```
   {
   abort();
   }
do_something_with(b); // ERROR. At this point,
                      //   'b' no longer exists
```
是不行的。do_something_with()的调用必须在 try 块中：
```
try
   {
   buffer b(100);
   do_something_with(b);
   }
catch (...)
   {
   abort();
   }
//do_something_with(b);
```
（免得被说闲话：我知道调用 abort()来处理这个异常有些过份。我只是用它做个示例，因为现在关心的是捕获异常而不是处理它。）

虽然有些笨拙，但这个方法是有效的。接着考虑这样的变化：
```
static buffer b(100);

int main()
   {
//   buffer b(100);
   do_something_with(b);
   return 0;
   }
```
现在，b 被定义为全局对象。试图将它包入 try 块
```
try // um, no, I don't think so
   {
   static buffer b;
   }
catch (...)
   {
   abort();
   }

int main()
   {
   do_something_with(b);
   return 0;
   }
```
将不能被编译。

10.2 暴露实现

每个例子都显示了 buffer 设计上的基本缺陷:buffer 的接口以外的实现细节被暴露了。在这里，暴露的细节是 buffer 的构造函数中的 new 语句可能失败。这个语句用于初始化私有子对象 buffer::p——一个 main 函数和其它用户不能操作甚至根本不知道的子对象。当然，这些用户更不应该被要求必须关注这样的子对象抛出的异常。

为了改善 buffer 的设计，我们必须在构造函数中捕获异常：

```
#include <stdlib.h>

class buffer
   {
public:
   explicit buffer(size_t);
   ~buffer();
private:
   char *p;
   };

buffer::buffer(size_t const count)
      : p(NULL)
   {
   try
      {
      p = new char[count];
      }
   catch (...)
      {
      abort();
      }
   }

buffer::~buffer()
   {
   delete[] p;
   }

static void do_something_with(buffer &)
   {
   }

int main()
   {
   buffer b(100);
   do_something_with(b);
   return 0;
```

```
   }
```

异常被包含在构造函数中。用户，比如 main()函数，从不知道异常存在过，世界又一次清静了。

10.3 常量成员

也这么做？注意，buffer::p 一旦被设置过就不能再被改动。为避免指针被无意改动，谨慎的设计是将它申明为 const：

```
class buffer
   {
public:
   explicit buffer(size_t);
   ~buffer();
private:
   char * const p;
   };
```

很好，但到了这步时：

```
buffer::buffer(size_t const count)
   {
   try
      {
      p = new char[count]; // ERROR
      }
   catch (...)
      {
      abort();
      }
   }
```

一旦被初始化，常量成员不能再被改变，即使是在包含它们的对象的构造函数体中。常量成员只能被构造函数的成员初始化列表设置一次。

```
buffer::buffer(size_t const count)
      : p(new char[count]) // OK
```

这让我们回到了段落一中，又重新产生了我们最初想解决的问题。

OK,这么样如何：不用 new 语句初始化 p，换成用内部使用 new 的辅助函数来初始化它：

```
char *new_chars(size_t const count)
   {
   try
      {
      return new char[count];
      }
   catch (...)
      {
```

```
      abort();
      }
   }

buffer::buffer(int const count)
      : p(new_chars(count))
   {

//   try
//      {
//      p = new char[count]; // ERROR
//      }
//   catch (...)
//      {
//      abort();
//      }
   }
```

这个能工作，但代价是一个额外函数却仅仅用来保护一个几乎从不发生的事件。

10.4 函数 try 块

（WQ 注：后面会讲到，function try 块不能阻止构造函数的抛异常动作，它其实只起异常过滤的功能!!! 见 P14.3）

我在上面这些建议中没有发现哪个能确实令人满意。我所期望的是一个语言级的解决方案来处理部分构造子对象问题，而又不引起上面说到的问题。幸运的是，语言中恰好包含了这样一个解决方法。

在深思熟虑后，C++标准委员会增加了一个叫做“function try blocks”的东西到语言规范中。作为 try 块的堂兄弟，函数 try 块捕获整个函数定义中的异常，包括成员初始化列表。不用奇怪，因为语言最初没有被设计了支持函数 try 块，所以语法有些怪：

```
buffer::buffer(size_t const count)
try
      : p(new char[count])
   {
   }
catch
   {
   abort();
   }
```

看起来想是通常的 try 块后面的{}实际上是划分构造函数的函数体的。在效果上，{}有双重作用，不然，我们将面对更别扭的东西：

```
buffer::buffer(int const count)
try
      : p(new char[count])
   {
```

```
   {
   }
   }
catch
   {
   abort();
   }
```

（注意：虽然嵌套的{}是多余的，这个版本能够编译。实际上，你可以嵌套任意重{}，直到遇到编译器的极限。）

如果在初始化列表中有多个初始化，我们必须将它们放入同一个函数 try 块中：

```
buffer::buffer()
try
   : p(...), q(...), r(...)
   {
   // constructor body
   }
catch (std::bad_alloc)
   {
   // ...
   }
```

和普通的 try 块一样，可以有任意个异常处理函数：

```
buffer::buffer()
try
   : p(...), q(...), r(...)
   {
   // constructor body
   }
catch (std::bad_alloc)
   {
   // ...
   }
catch (int)
   {
   // ...
   }
catch (...)
   {
   // ...
   }
```

古怪的语法之外，函数 try 块解决了我们最初的问题：所有从 buffer 子对象的构造函数抛出的异常留在了 buffer 的构造函数中。

因为我们现在期望 buffer 的构造函数不抛出任何异常，我们应该给它一个异常规格申明：

```
explicit buffer(size_t) throw();
```

接着一想，我们应该是个更好点的程序员，于是给我们所有函数加了异常规格申明：

```
class buffer
   {
public:
   explicit buffer(size_t) throw();
   ~buffer() throw();
   // ...
   };

// ...

static void do_something_with(buffer &) throw()

// ...
```

Rounding Third and Heading for Home

对我们的例子，最终版本是：

```
#include <stdlib.h>

class buffer
   {
public:
   explicit buffer(size_t) throw();
   ~buffer() throw();
private:
   char *const p;
   };

buffer::buffer(size_t const count)
try
      : p(new char[count])
   {
   }
catch (...)
   {
   abort();
   }

buffer::~buffer()
   {
   delete[] p;
   }
```

```
static void do_something_with(buffer &) throw()
   {
   }

int main()
   {
   buffer b(100);
   do_something_with(b);
   return 0;
   }
```

用 Visual C++编译，自鸣得意地坐下来，看着 IDE 的提示输出。

```
syntax error : missing ';' before 'try'
syntax error : missing ';' before 'try'
'count' : undeclared identifier
'<Unknown>' : function-style initializer appears
   to be a function definition
syntax error : missing ';' before 'catch'
syntax error : missing ';' before '{'
missing function header (old-style formal list?)
```

噢！

Visual C++还不支持函数 try 块。在我测试过的编译器中，只有 Edison Design Group C++ Front End version 2.42 认为这些代码合法。

（顺便提一下，我特别关心为什么编译将第一个错误重复了一下。可能它的计算你第一次会不相信。）

如果你坚持使用 Visual C++，你可以使用在介绍函数 try 块前所说的解决方法。我喜欢使用额外的 new 封装函数。如果你认同，考虑将它做成模板：

```
template <typename T>
T *new_array(size_t const count)
   {
   try
      {
      return new T[count];
      }
   catch (...)
      {
      abort();
      }
   }

// ...

buffer::buffer(size_t const count)
      : p(new_array<char>(count))
```

```
{
}
```

这个模板比原来的 new_chars 函数通用得多，对 char 以外的类型也有能工作。同时，它有一个隐蔽的异常相关问题，而我将在下次谈到。

回到目录

11. 异常规格申明

现在是探索 C++标准运行库和 Visual C++在头文件<exception>中申明的异常支持的时候了。根据 C++标准（subclause 18.6，“Exception handling” ）上的描述，这个头文件申明了：

- 从运行库中抛出的异常对象的基类。
- 任何抛出的违背异常规格申明的对象的可能替代物。
- 在违背异常规格申明的异常被抛出是被调用的函数，以及在其行为上增加东西的钩子（“hook”）。
- 在异常处理过程被终止时被调用的函数，以及在其行为上增加东西的钩子。

我从分析异常规格申明及程序违背它时遭到什么可怕后果开始。分析将针对上面提到的主题，以及通常 C++异常处理时的一些杂碎。

11.1 异常规格申明回顾

异常规格申明是 C++函数申明的一部分，它们指定了函数可以抛出什么异常。例如，函数

```
void f1() throw(int)
```

可以抛出一个整型异常，而

```
void f2() throw(char *, E)
```

可以抛出一个 char *或一个 E（这里 E 是用户自定义类型）类型的异常。一个空的规格申明

```
void f3() throw()
```

表明函数不抛出异常，而没有规格申明

```
void f4()
```

表明函数可以抛出任何东西。注意语法

```
void f4() throw(...)
```

比前面的“抛任何东西”的函数更好，因为它类似“捕获任何东西”

```
catch(...)
```

然而，认可“抛任何东西” 的函数就允许了那些在异常规格申明存在前写下的函数。

11.2 违背异常规格申明

迄今为止，我写的都是：函数可能抛出在它的异常规格申明中描述的异常。“可能”有些单薄，“必须”则有力些。“可能”表示了函数可以忽略它们的异常规格。你也许认为编译器将禁止这种行为：

```
void f() throw() // Promises not to throw...
   {
   throw 1;      // ...but does anyway - error?
```

```
   }
```

但你错了。用 Visual C++试一下，你将发现编译器保持沉默，它没有发现编译期错误。实际上，在我所用过的编译器中，没有一个报了编译期错误。

话虽这么说，但异常规格申明有它的规则的，函数违背它将遭受严重后果的。不幸的是，这些后果表现在运行期错误而不是编译期。想看的话，把上面的小段代码放到一个完整程序中：

```
void f() throw()
   {
   throw 1;
   }

int main()
   {
   f();
   return 0;
   }
```

当程序运行时将发生什么？f()抛出一个 int 型异常，违背了它的契约。你可能认为这个异常将从 main()中漏入运行期库。基于这个假设，你倾向于使用一个简单的 try 块：

```
#include <stdio.h>

void f() throw()
   {
   throw 1;
   }

int main()
   {
   try
      {
      f();
      }
   catch (int)
      {
      printf("caught int\n");
      }
   return 0;
   }
```

来捕获这个异常，以防止它漏出去。

实际上，如果你用 Visual C++ 6 编译并运行，你将得到：

```
caught int
```

你再次奇怪 throw()异常规格实际做了什么有用的事，除了增加了源代码的大小和看起来比较快感。你的奇怪感觉将变得迟钝，只要一回想到前面说了多少 Visual C++违背 C++

标准的地方，只不过再多一个新问题：Visaul C++正确地处理了违背异常规格申明的情况了吗？

11.3 调查说明……

没有！

这个程序的行为符合标准吗？catch 语句不该进入的。来自于标准（subclauses 15.5.2 and 18.6.2.2）：

- **一个异常规格申明保证只有被列出的异常被抛出。**
- **如果带异常规格申明的函数抛出了一个没有列出的异常，函数**
- **void unexpected()在退完栈后立即被调用。**
- **函数 unexpected()将不会返回……**

当一个函数试图抛出没有列出的异常时，通过 unexpected()函数调用了一个异常处理函数。这个异常处理函数的默认实现是调用 terminate() 来结束程序。

在我给你一个简短的例程后，我将展示 Visual C++的行为怎么样地和标准不同。

11.4 unexpected()函数指南

unexpected()函数是标准运行库在头文件<exception>中申明的函数。和其它大部分运行库函数一样，unexpected()函数存在于命名空间 std 中。它不接受参数，也不返回任何东西，实际上 unexpected()函数从不返回，就象 abort()和 exit()一样。如果一个函数违背了它自己的异常规格申明，unexpected()函数在退完栈后被立即调用。

基于我对标准的理解，运行库的 unexpected()函数的实现理论上是这样的：

```
void _default_unexpected_handler_()
   {
   std::terminate();
   }

std::unexpected_handler _unexpected_handler =
      _default_unexpected_handler;

void unexpected()
   {
   _unexpected_handler();
   }
```

（_default_unexpected_handler 和_unexpected_handler 是我虚构的名字。你的运行库的实现可能使用其它名称，完全取决于其实现。 ）

std::unexpected()调用一个函数来真正处理 unexpected 的异常。它通过一个隐藏的指针（_unexpected_handler，类型是 std::unexpected_handler）来引用这个处理函数的。运行库提供了一个默认处理函数（default_unexpected_handler()），它调用 std::terminate()来结束程序。

因为是通过指针_unexpected_handler 间接调用的，你可以将内置的调用 _default_unexpected_handler 改为调用你自己的处理函数，只要这个处理函数的类型兼容

于 std::unexpected_handler：

```
typedef void (*unexpected_handler)();
```

同样，处理函数必须不返回到它的调用者（std::unexpected()）中。没人阻止你写一个会返回的处理函数，但这样的处理函数不是标准兼容的，其结果是程序的行为有些病态。

你可以通过标准运行库的函数 std::set_unexpected()来挂接自己的处理函数。注意，运行库只维护一个处理函数来处理所有的 unexpected 异常;一旦你调用了 set_unexpected()函数，运行库将不再记得前一次的处理函数。（和 atexit()比较一下，atexit()至少可以挂 32 重 exit 处理函数。）要克服这个限制，你要么在不同的时间设置不同的处理函数，要么使你的处理函数在不同的上下文时有不同的行为。

11.5 Visual C++ vs unexpected

试一下这个简单的例子：

```
#include <exception>
#include <stdio.h>
#include <stdlib.h>
using namespace std;

void my_unexpected_handler()
   {
   printf("in unexpected handler\n");
   abort();
   }

void throw_unexpected_exception() throw(int)
   {
   throw 1L; // violates specification
   }

int main()
   {
   set_unexpected(my_unexpected_handler);
   throw_unexpected_exception();
   printf("this line should never appear\n");
   return 0;
   }
```

用一个标准兼容的编译器编译并运行，程序结果是：

```
in unexpected handler
```

可能接下来是个异常异常终止的特殊（因为有 abort()的调用）。但用 Visual C++编译并运行，程序会抛出“Unhandled exception”对话框。关闭对话框后，程序输出：

```
this line should never appear
```

必须承认，Visual C++没有正确实现 unexpected()。这个函数被申明在<exception>中，运行期库中有其实现，只不过这个实现不做任何事。

实际上，Visual C++甚至没有正确地申明，用这个理论上等价的程序可以证明：

```
#include <exception>
#include <stdio.h>
#include <stdlib.h>
//using namespace std;

void my_unexpected_handler()
   {
   printf("in unexpected handler\n");
   abort();
   }

void throw_unexpected_exception() throw(int)
   {
   throw 1L; // violates specification
   }
int main()
   {
   std::set_unexpected(my_unexpected_handler);
   throw_unexpected_exception();
   printf("this line should never appear\n");
   return 0;
   }
```

Visual C++不能编译这个程序。查看<exception>表明：set_unexpected_handler()被申明为全局函数而不是在命名空间 std 中。实际上，所有的 unexpected 族函数都被申明为全局函数。

底线：Visual c++能编译使用 unexpected()等函数的程序，但运行时的行为是不正确的。

我希望Microsoft 能在下一版中改正这些问题。在未改正前，当讨论涉及到unexpected()时，我建议你使用标准兼容的 C++编译器。

11.6 维持程序存活

在我所展示的简单例子中，程序在 my_unexpected_handler()里停止了。有时，让程序停止是合理和正确的；但更多情况下，程序停止是太刺激了，尤其是当 unexpected 异常表明的是程序只轻微错误。

假定你想处理 unexpected 异常，并恢复程序，就象对大多数其它“正常”异常一样。因为 unexpected()从不返回，程序恢复似乎不可能，除非你看了标准的 subclause 15.5.2：

unexpected()不该返回，但它可以 throw（或 re-throw）一个异常。如果它抛出一个新异常，而这异常是异常规格申明允许的，搜索另外一个异常处理函数的行为在调用 unexpected()的地方继续进行。

太好了！如果 my_unexpected_handler()抛出一个允许的异常，程序就能从最初的违背异常规格申明的地方恢复了。在我们的例子里，最初的异常规格申明允许 int 型的异常。根据上面的说法，如果 my_unexpected_handler 抛出一个 int 异常，程序将能继续了。

基于这种猜测，试一下：

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

void my_unexpected_handler()
   {
   printf("in unexpected handler\n");
   throw 2; // allowed by original specification
   //abort();
   }
```

用标准兼容的编译器编译运行，程序输出：

```
in unexpected handler
program resumed
```

和期望相符。

抛出的 int 异常和其它异常一样顺调用链传递，并被第一个相匹配的异常处理函数捕获。在我们的例子里，程序的控制权从 my_unexpected_handler()向 std::unexpected()再向 main()回退，并在 main()中捕获异常。用这种方法，my_unexpected_handler()变成了一个异常转换器，将一个最初的“坏”的 long 型异常转换为一个“好”的 int 型异常。

结论：通过转换一个 unexpected 异常为 expected 异常，你能恢复程序的运行。

11.7 预告

下次，我将结束 std::unexpected()的讨论：揭示在 my_unexpected_handler()中抛异常的限制，探索运行库对这些限制的补救，并给出处理 unexpected 异常的通行指导原则。我也将开始讨论运行库函数 std::terminate()的相关内容。

```
void throw_unexpected_exception() throw(int)
   {
   throw 1L; // violates specification
   }

int main()
   {
   std::set_unexpected(my_unexpected_handler);
   try
      {
      throw_unexpected_exception();
      printf("this line should never appear\n");
      }
   catch (int)
      {
      printf("program resumed\n");
      }
   return 0;
```

```
   }
```
回到目录

12. unexpected()的实现上固有的限制

上次，我介绍了 C++标准运行库函数 unexpected()，并展示了 Visual C++的实现版本中的限制。这次，我想展示所有 unexpected()的实现上固有的限制，以及绕开它们的办法。

12.1 异常处理函数是全局的、通用的

我在上次简要地提过这点，再推广一点：过滤 unexpected 异常的异常处理函数 unexpected()是全局的，对每个程序是唯一的。

所有 unexpected 异常都被同样的一个 unexpected()异常处理函数处理。标准运行库提供默认的处理函数来处理所有 unexpected 异常。你可以用自己的版本覆盖它，这时，运行库会调用你提供的处理函数来处理所有的 unexpected 异常。

和普通的异常处理函数，如：

```
catch (int)
   {
   }
```

不同，unexpected 异常处理函数不“捕获”异常。一旦被进入，它就知道有 unexpected 异常被抛出，但不知道类型和起因，甚至没法得到运行库的帮助：运行库中没有程序或对象保存这些讨厌的异常。

在最好的情况下，unexpected 异常处理函数可以把控制权交给程序的其它部分，也许它们有更好的办法。例如：

```
#include <exception>
using namespace std;

void my_unexpected_handler()
   {
   throw 1;
   }

void f() throw(int)
   {
   throw 1L; // oops -- *bad* function
   }

int main()
   {
   set_unexpected(my_unexpected_handler);
   try
      {
      f();
      }
```

```
   catch (...)
      {
      }
   return 0;
   }
```

f()抛出了一个它承诺不抛的异常，于是 my_unexpected_handler()被调用。这个处理函数没有任何办法来判断它被进入的原因。除了结束程序外，它唯一可能有些用的办法是抛出另外一个异常，希望新异常满足被老异常违背的异常规格申明，并且程序的其它部分将捕获这个新异常。

在这个例子里，my_unexpected_handler()抛出的 int 异常满足老异常违背的异常规格申明，并且 main()成功地捕获了它。但稍作变化：

```
#include <exception>
using namespace std;

void my_unexpected_handler()
   {
   throw 1;
   }

void f() throw(char)
   {
   throw 1L; // oops -- *bad* function
   }

int main()
   {
   set_unexpected(my_unexepected_handler);
   try
      {
      f();
      }
   catch (...)
      {
      }
   return 0;
   }
```

my_unexpected_handler()仍然在 unexpected 异常发生后被调用，并仍然抛出了一个 int 型异常。不幸的是，int 型异常现在和老异常违背的异常规格申明相违背。因此，我们现在两次违背了同一异常规格申明：第一次是 f()，第二次是 f()的援助者 my_unexpected_handler()。

12.2 Terminate

现在，程序放弃了，并调用运行库的程序 terminate()自毁。terminate()函数是标准

运行库在异常处理上的最后一道防线。当程序的异常处理体系感到无望时，C++标准要求程序调用 terminate()函数。C++标准的 Subclause 15.5.1 列出了调用 terminate()的情况：

和 unexpected()处理函数一样，terminate()处理函数也可以用户定义。但和 unexpected()处理函数不同的是，terminate()处理函数必须结束程序。记住：当你的 terminate()处理函数被进入时，异常处理体系已经无效了，此是程序所需要的最后一件事是找一个 terminate()处理函数来丢弃异常。

在能避免时就不要让你的程序调用 terminate()。terminate()其实是个叫得好听点的 exit()。如果 terminate()被调用了，你的程序就会以一种不愉快的方式死亡。

就如同不能完全支持 unexpected()一样，Visual c++也不能完全支持 terminate()。要在实际运行中验证的话，运行：

```
#include <exception>
#include <stdlib.h>
#include <stdio.h>
using namespace std;

void my_terminate_handler()
   {
   printf("in my_terminate_handler\n");
   abort();
   }

int main()
   {
   set_terminate(my_terminate_handler);
   throw 1; // nobody catches this
   return 0;
   }
```

根据 C++标准，抛出了一个没人捕获的异常将导致调用 terminate()（这是我前面提到的 Subclause 15.5.1 中列举的情况之一）。于是，上面的程序一个输出：

```
in my_terminate_handler
```

但，用 Visual C++编译并运行，程序没有输出任何东西。

12.3 避免 terminate

在我们的 unexpected()例子中，terminate()最终被调用是因为 f()抛出了 unexpected 异常。我们的 unexpected_handler()试图阻住这个不愉快的事，通过抛出一个新异常，但没成功；这个抛出行为因再度产生它试图解决的那个问题而结束。我们需要找到一个方法以使得 unexpected()处理函数将控制权传给程序的其它部分（假定那部分程序是足够聪明的，能够成功处掉异常）而不导致程序终止。

很高兴，C++标准正好提供了这样一个方法。如我们所看过的，从 unexpected()处理函数中抛出的异常对象必须符合（老异常违背的）异常规格申明。**这个规则有一个例外：如果如果被违背的异常规格申明中包含类型 bad_exception，一个 bad_exception 对象将替代**

unexpected()处理函数抛出的对象。例如：

```
#include <exception>
#include <stdio.h>
using namespace std;
void my_unexpected_handler()
   {
   throw 1;
   }

void f() throw(char, bad_exception)
   {
   throw 1L; // oops -- *bad* function
   }

int main()
   {
   set_unexpected(my_unexpected_handler);
   try
      {
      f();
      }
   catch (bad_exception const &)
      {
      printf("caught bad_exception\n");
      // ... even though such an exception was never thrown
      }
   return 0;
   }
```

当用C++标准兼容的编译器编译并运行，程序输出：

```
caught bad_exception
```

当用Visual C++编译并运行，程序没输出任何东西。因为Visual c++并没有在第一次抛异常的地方捕获unexpected异常，它没有机会进行bad_exception的替换。

和前面的例子相同的是，f()仍然违背它的异常规格申明，而my_unexpected_handler()仍然抛出一个int。不同之处是：f()的异常规格申明包含bad_exception。结果，程序悄悄地将 my_unexpected_handler()原来抛出的 int 对象替换为 bad_exception 对象。因为bad_exception异常是允许的，terminate()没有被调用，并且这个bad_exception异常能被程序的其它部分捕获。

最终结果：最初从f()抛出的long异常先被映射为int，再被映射为bad_exception。这样的映射不但避免了前面导致terminate的再次异常问题，还给程序的其它部分一个修正的机会。bad_exception异常对象的存在表明了某处最初抛出了一个unexpected异常。通过在问题点附近捕获这样的对象，程序可以得体地恢复。

我也注意到一个奇怪的地方。在代码里，你看到 f()抛出了一个 long，my_unexpected_handler()抛出了一个int，而没人抛出bad_exception，但main()确实捕

获到一个 bad_exception。是的，程序捕获了一个它从没抛出的对象。就我所知，唯一被允许发生这种行为的地方就是unexpected异常处理函数和bad_exception异常间的相互作用。

12.4 一个更特别的函数

C++标准定义了 3 个“特别”函数来捕获异常。其中，你已经看到了 terminate()和unexpected()。最后，也是最简单的一个是 uncaght_exception()。摘自 C++标准（15.5.3）：

函数 bool uncaught_exception()在被抛出的异常对象完成赋值到匹配的异常处理函数的异常申明完成初始化之间返回 true。包括其中的退栈过程。如果异常被再次抛出，uncaught_exception() 从再抛点到再抛对象被再次捕获间返回 true。

uncaught_exception() 让你查看是否程序抛出了一个异常而还没有被捕获。这个函数对析构函数有特别意义：

```
#include <exception>
#include <stdio.h>
using namespace std;

class X
   {
public:
   ~X();
   };

X::~X()
   {
   if (uncaught_exception())
      printf("X::~X called during stack unwind\n");
   else
      printf("X::~X called normally\n");
   }

int main()
   {
   X x1;
   try
      {
      X x2;
      throw 1;
      }
   catch (...)
      {
      }
   return 0;
   }
```

在 C++标准兼容的环境下，程序输出：

X::~X called during stack unwind
X::~X called normally

x1 和 x2 在 main()抛出异常前构造。退栈时调用 x2 的析构函数。因为一个未被捕获的异常在析构函数调用期间处于活动状态，uncaught_exception()返回 true。然后，x1 的析构函数被调用（在 main()退出时），异常已经恢复，uncaught_exception()返回 false。

和以前一样，Visual C++在这里也不支持 C++标准。在其下编译，程序输出：

X::~X called normally
X::~X called normally

如果你了解 Microsoft 的 SEH（我在第二部分讲过的），就知道 uncaught_exception()类似于 SEH 的 AbnormalTermination()。在它们各自的应用范围内，两个函数都是检测是否一个被抛出的异常处于活动状态而仍然没有被捕获。

12.5 小结

大多数函数不直接抛异常，但将其它函数抛的异常传递出来。决定哪些异常被传递是非常困难的，尤其是来自于没有异常规格申明的函数的。bad_exception ()是一个安全的阀门，提供了一个方法来保护那些你不能进行完全解析的异常。

这些保护能工作，但，和普通的异常处理函数一样，需要你明确地设计它。对每个可能违背其异常规格申明的函数，你都必须记得在其异常规格申明中加一个 bad_exception 并在某处捕获它。bad_exception 和其它异常没有什么不同：如果你不想捕获它，不去产生它就行了。一个没有并捕获的 bad_exception 将导致程序终止，就象在最初的地方你没有使用 bad_exception 进行替换一样。

异常规格申明使你意图明确。它说“这是我允许这个函数抛出的异常的集合；如果函数抛出了其它东西，不是我的设计错了就是程序有神经病（the program is buggy)”。一个 unexpected 异常，不管它怎么出现的，都表明了一个逻辑错误。我建议你最好让错误以一种可预见的方式有限度地发生。

所有这些表明你可以描绘你的代码在最开始时的异常的行为。不幸的是，这样的描绘接近于巫术。下次，我将给出一些指导方针来分析你的代码中的异常。

回到目录

13. 异常安全

接下来两次，我将讨论“异常安全”，C++标准中使用了（在 auto_ptr 中）却没有定义的术语。在 C++范围内，不同的作者使用这个术语却表达不同的含义。在我的专题中，我从两个方面来定义“异常安全”：

- 如果一个实体捕获或抛出一个异常，但仍然维持它公开保证的语义，它就是“接口安全”的。依赖于它保证的力度，实体可能不允许将任何异常漏给其用户。
- 如果异常没有导致资源泄漏或产生未定义的行为，实体就是“行为安全”的。“行为安全”一般是强迫的。幸运的是，如果做到了“行为安全”，通常也间接提供了“接口安全”。

异常安全有点象 const：好的设计必须在一开始就考虑它，它不能够事后补救。但是我们开始使用异常还没有多少年，所以还没有“异常安全问题集”这样的东西来指导我们。实际上，我期望大家通过一条艰辛的道路来掌握异常安全：通过经历异常故障在编码时绕过它们；或关闭异常特性，认为它们“太难”被正确掌握。

我不想撒谎：分析设计上的异常安全性太难了。但是，艰辛的工作也有丰厚的回报。不过，这个主题太难了，想面面俱到的话将花我几个月的时间。我最小的目标是：通过缺乏异常安全的例子来展示怎么使它们变得安全，并激励你在此专题之外去看和学更多的东西。

13.1 构造函数

如果一个普通的成员函数

```
x.f()
```

抛出一个异常，你可以容忍此异常并试图再次调用它：

```
X x;
bool done;
do
   {
   try
      {
      done = true;
      x.f();
      }
   catch (...)
      {
      // do something to recover, then retry
      done = false;
      }
   }
while (!done);
```

但，如果你试图再次调用一个构造函数，你实际上是调用了一个完全不同的对象：

```
bool done(false);
while (!done)
   {
   try
      {
      done = true;
      X x; // calls X::X()
      }
   // from this point forward, `x` does not exist
   catch (...)
      {
      // do something to recover, then retry
      done = false;
      }
   }
```

你不能挽救一个构造函数抛异常的对象；异常的存在表明那个对象已经死了。

当一个构造函数抛异常时，它杀死了其宿主对象而没有调用析构函数。这样的抛异常行为危害了“行为安全”：如果这个抛异常的构造函数分配了资源，你无法依赖析构函数释放

它们。一般构造和析构是成对的，并期待后者清理前者。如果析构函数没有被调用，这个期望是不满足的。

最后，如果你从构造函数中抛了一个异常，并且你的类是用户类的一个基类或子对象，那么用户类的构造函数必须处理你抛出的异常。或者它将异常抛给另外一个用户类的构造函数，如此递推下去，直到程序调用 terminate()。实际上用户必须做你没有做的工作（维持构造函数的安全性）。

13.2 关于取舍的问题

构造函数抛异常同时降低了接口安全和行为安全。除非有迫不得以的理由，不要让构造函数抛异常。

也有不同的意见认为：异常应该被本来就做这事的专门代码捕获的。那些只是静静地接收异常而没有处理它们的异常处理函数违背了这些异常的初衷。如果一个函数没有准备好正确地处理一个异常，它应该将这个异常传递下去。

最低事实是：必须有人处理异常；如果所有人都放过它，程序将终止。还必须同时捕获触发异常的条件；如果没人标记它，程序可能以任何方式终止，并且恐怕不怎么文雅。

一个异常对象警示我们存在一个不该忽略的错误状况。不幸的是，这个对象的存在可能导致一个全新的不同的错误状况。在设计异常安全的时候，你必须在两个有时冲突的设计原则间进行取舍。

1.在错误发生时进行通报

2.防止这个通报行为导致其它错误。

因为构造函数抛异常可能有有害的副作用，你必须小心权衡这两个原则。我不允许我写的构造函数中抛异常，这样设计倾向于原则 2；但我不想将它推荐为普遍原则，在其它情况下这两个原则是等重的。自己好自判断吧。

13.3 析构函数

析构函数抛异常可能使程序有奇怪的反应。它可能彻底地杀死程序。根据 C++标准（subclause 15.1.1，“the terminate() function” ），简述如下：

在某些情况下，异常处理必须被抛弃以减少一些微妙的错误。这些情况中包括：当因为异常而退栈过程中将要被析构的对象的析构函数。在这些情况下，函数 void terminate()被调用。退栈不会完成。

简而言之，析构函数不该提示是否发生了异常。但，如我上次所说，新的 C++标准运行库程序 uncaught_exception()可以让析构函数确定其所处的异常环境。不幸的是，我上次也说了，Visual C++未能正确地支持这个函数。

问题比我提示的还要糟。我上次写到，Microsoft 的 uncaught_exception()函数版本一定返回 false，所以 Visaul C++总告诉你的析构函数当前没有发生异常，在其中抛异常是可以的。如果你从一个支持 uncaught_exception 的环境转到 Visual C++，以前正常工作的代码可能开始调用 terminate()了。

要尝试一下的话，试下面的例子：

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

```
#include <stdlib.h>

using namespace std;

static void my_terminate_handler(void)
   {
   printf("Library lied; I'm in the terminate handler.\n");
   abort();
   }

class X
   {
public:
   ~X()
      {
      if (uncaught_exception())
         printf("Library says not to throw.\n");
      else
         {
         printf("Library says I'm OK to throw.\n");
         throw 0;
         }
      }
   };

int main()
   {
   set_terminate(my_terminate_handler);
   try
      {
      X x;
      throw 0;
      }
   catch (...)
      {
      }
   printf("Exiting normally.\n");
   return 0;
   }
```

在C++标准兼容的环境下，你得到：

```
Library says not to throw.
Exiting normally.
```

但 Visual C++下，你得到：

```
Library says I'm OK to throw.
Library lied; I'm in the terminate handler.
```

并跟随一个程序异常终止。

```
And with six you get egg roll.
```

建议：除非你确切知道你现在及以后所用的平台都正确支持 uncaught_exception()，不要调用它。

13.4 部分删除

即使你知道当前不在处理异常，你仍然不应该在析构函数中抛异常。考虑如下的例子：

```
class X
   {
public:
   ~X()
      {
      throw 0;
      }
   };

int main()
   {
   X *x = new X;
   delete x;
   return 0;
   }
```

当 main()执行到 delete x，如下两步将依次发生：

```
x 的析构函数被调用。
operator delete 被调用了来释放 x 的内存空间。
```

但因为 x 的析构函数抛了异常，operator delete 没有被调用。这危及了行为安全。如果还不信，试一下这个更完整的例子：

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

class X
   {
public:
   ~X()
      {
      printf("destructor\n");
      throw 0;
```

```
    }
  void *operator new(size_t n) throw()
    {
    printf("new\n");
    return malloc(n);
    }
  void operator delete(void *p) throw()
    {
    printf("delete\n");
    if (p != NULL)
      free(p);
    }
  };

int main()
  {
  X *x = new X;
  try
    {
    delete x;
    }
  catch (...)
    {
    printf("catch\n");
    }
  return 0;
  }
```

如果析构函数没有抛异常，程序输出：

```
new
destructor
delete
```

实际上程序输出：

```
new
destructor
catch
```

operator delete 没有进入，x 的内存空间没有被释放，程序有资源泄漏，the press hammers your product for eating memory, and you go back to flipping burgers for a living。

原则：异常安全要求你不能在析构函数中抛异常。和在构造函数抛异常上有不同意见不一样，这条是绝对的。为了明确表明意图，应该在申明析构函数时加上异常规格申明throw()。

13.5 预告

我本准备覆盖模板安全的，但没地方了。我将留到下次介绍，并开出推荐读物表。

回到目录

14. 模板安全

上次，我开始讨论异常安全。这次，我将探究模板安全。

模板根据参数的类型进行实例化。因为通常事先不知道其具体类型，所以也无法确切知道将在哪儿产生异常。你大概最期望的就是去发现可能在哪儿抛异常。这样的行为很具挑战性。

看一下这个简单的模板类：

```
template <typename T>
class wrapper
   {
public:
   wrapper()
      {
      }
   T get()
      {
      return value_;
      }
   void set(T const &value)
      {
      value_ = value;
      }
private:
   T value_;
   wrapper(wrapper const &);
   wrapper &operator=(wrapper const &);
   };
```

如名所示，wrapper 包容了一个 T 类型的对象。方法 get()和 set()得到和改变私有的包容对象 value_。两个常用方法——拷贝构造函数和赋值运算符没有使用，所以没有定义，而第三个——析构函数由编译器隐含定义。

实例化的过程很简单，例如：

```
wrapper<int> i;
```

包容了一个 int。i 的定义过程导致编译器从模板实例化了一个定义为 wrapper<int>的类：

```
template <>
class wrapper<int>
   {
public:
```

```
   wrapper()
      {
      }
   int get()
      {
      return value_;
      }
   void set(int const &value)
      {
      value_ = value;
      }
private:
   int value_;
   wrapper(wrapper const &);
   wrapper &operator=(wrapper const &);
   };
```

因为 wrapper<int>只接受 int 或其引用（一个内嵌类型或内嵌类型的引用），所以不会触及异常。wrapper<int>不抛异常，也没有直接或间接调用任何可能抛异常的函数。我不进行正规的分析了，但相信我：wrapper<int>是异常安全的。

14.1 class 类型的参数

现在看：

```
wrapper<X> x;
```

这里 X 是一个类。在这个定义里，编译器实例化了类 wrapper<X>：

```
template <>
class wrapper<X>
   {
public:
   wrapper()
      {
      }
   X get()
      {
      return value_;
      }
   void set(X const &value)
      {
      value_ = value;
      }
private:
   X value_;
   wrapper(wrapper const &);
```

```
wrapper &operator=(wrapper const &);
};
```

粗一看，这个定义没什么问题，没有触及异常。但思考一下：

- wrapper<X>包容了一个 X 的子对象。这个子对象需要构造，意味着调用了 X 的默认构造函数。这个构造函数可能抛异常。
- wrapper<X>::get()产生并返回了一个 X 的临时对象。为了构造这个临时对象，get()调用了 X 的拷贝构造函数。这个构造函数可能抛异常。
- wrapper<X>::set()执行了表达式 value_ = value，它实际上调用了 X 的赋值运算。这个运算可能抛异常。

在 wrapper<int>中针对不抛异常的内嵌类型的操作现在在 wrapper<X>中变成调用可能抛异常的函数了，同样的模板，同样的语句，但极其不同的含义。

由于这样的不确定性，我们需要采用保守的策略：假设 wrapper 会根据类来实例化，而这些类在其成员上没有异常规格申明，它们可能抛异常。

14.2 使得包容安全

再假设 wrapper 的异常规格申明承诺其成员不产生异常。至少，我们必须在其成员上加上异常规格申明 throw()。我们需要修补掉这些可能导致异常的地方：

- 在 wrapper::wrapper()中构造 value_的过程。
- 在 wrapper::get()中返回 value_的过程。
- 在 wrapper::set()中对 value_赋值的过程。

另外，在违背 throw()的异常规格申明时，我们还要处理 std::unexpected。

14.3 Leak #1：默认构造函数

对 wrapper 的默认构造函数，解决方法看起来是采用 function try 块：

```
wrapper() throw()
   try : T()
      {
      }
   catch (...)
      {
      }
```

虽然很吸引人，但它不能工作。根据 C++标准（paragraph 15.3/16，“Handling an exception”）：

对构造或析构函数上的 function-try-block，当控制权到达了异常处理函数的结束点时，被捕获的异常被再次抛出。对于一般的函数，此时是函数返回，等同于没有返回值的 return 语句，对于定义了返回类型的函数此时的行为为未定义。

换句话说，上面的程序相当于是：

```
X::X() throw()
   try : T()
      {
      }
```

```
   catch (...)
      {
      throw;
      }
```

这不是我们想要的。

我想过这样做：

```
X::X() throw()
   try
      {
      }
   catch (...)
      {
      return;
      }
```

但它违背了标准的 paragraph 15：

如果在构造函数上的 function-try-block 的异常处理函数体中出现了 return 语句,程序是病态的。

我被标准卡死了，在用支持 function try 块的编译器试验后，我没有找到让它们以我所期望的方式运行的方法。不管我怎么尝试，所有被捕获的异常都仍然被再次抛出，违背了 throw()的异常规格申明，并打败了我实现接口安全的目标。

原则：无法用 function try 块来实现构造函数的接口安全。

引申原则 1：尽可能使用构造函数不抛异常的基类或成员子对象。

引申原则 2：为了帮助别人实现引申原则 1，不要从你的构造函数中抛出任何异常。（这和我在 Part13 中所提的看法是矛盾的。）

我发现 C++标准的规则非常奇怪，因为它们减弱了 function try 的实际价值：在进入包容对象的构造函数（wrapper::wrapper()）前捕获从子对象（T::T()）构造函数中抛出的异常。实际上，function try 块是你捕获这样的异常的唯一方法；但是你只能捕获它们却不能处理掉它们！

（WQ 注：下面的文字原载于 Part15 上，我把提前了。

上次我讨论了 function try 块的局限性，并承诺要探究其原因的。我所联系的业内专家没人知道确切答案。现在唯一的共识是：

- 如我所猜测，标准委员会将 function try 块设计为过滤而不是捕获子对象构造函数中发生的异常的。
- 可能的动机是：确保没人误用没有构造成功的包容对象。

我写信给了 Herb Sutter，《teh Exceptional C++》的作者。他从没碰过这个问题，但很感兴趣，以至于将其写入“Guru of the Week”专栏。如果你想加入这个讨论，到新闻组 comp.lang.c++.moderated 上去看“Guru of the Week #66: Constructor Failures”。

）

注意 function try 可以映射或转换异常：

```
X::X()
   try
```

```
      {
      throw 1;
      }
   catch (int)
      {
      throw 1L; // map int exception to long exception
      }
```

这样看，它们非常象 unexpected 异常的处理函数。事实上，我现在怀疑这才是它们的设计目的（至少是对构造函数而言）：更象是个异常过滤器而不是异常处理函数。我将继续研究下去，以发现这些规则后面的原理。

现在，至少，我们被迫使用一个不怎么直接的解决方法：

```
template <typename T>
class wrapper
   {
public:
   wrapper() throw()
         : value_(NULL)
      {
      try
         {
         value_ = new T;
         }
      catch (...)
         {
         }
      }
   // ...
private:
   T *value_;
   // ...
   };
```

被包容的对象，原来是在 wrapper::wrapper()进入前构造的，现在是在其函数体内构造的了。这个变化可以让我们使用普通的方法来捕获异常而不用 function try 块了。

因为 value_现在是个 T *而不是 T 对象了，get()和 set()必须使用指针的语法了：

```
T get()
   {
   return *value_;
   }

void set(T const &value)
   {
   *value_ = value;
```

```
}
```

14.4 Leak #1A：operator new

在构造函数内的 try 块中，语句

```
value_ = new T;
```

隐含地调用了 operator new 来分配*value_的内存。而这个 operator new 函数可能抛异常。

幸好，我们的 wrapper::wrapper()能同时捕获 T 的构造函数和 operator new 函数抛出的异常，因此维持了接口安全。但，记住这个关键性的差异：

- 如果 T 的构造函数抛了异常，operator delete 被隐含调用了来释放分配的内存。（对于 placement new，这取决于是否存在匹配的 operator delete，我在 part 8 和 9 说过了的。）
- 如果 operator new 抛了异常，operator delete 不会被隐含调用。

第二点本不该有什么问题：如果 operator new 抛了异常，通常是因为内存分配失败，operator delete 没什么需要它去释放的。但，如果 operator new 成功分配了内存但因为其它原因而仍然抛了异常，它必须负责释放内存。换句话说，operator new 自己必须是行为安全的。

（同样的问题也发生在通过 operator nwe[]创建数组时。）

14.5 Leak #1B：Destructor

想要 wrapper 行为安全，我们需要它的析构函数释放 new 出来的内存：

```
~wrapper() throw()
   {
   delete value_;
   }
```

这看起来很简单，但请等一下说大话！delete value_调用*value_的析构函数，而这个析构函数可能抛异常。要实现~wrapper()的接口异常，我们必须加上 try 块：

```
~wrapper() throw()
   {
   try
      {
      delete value_;
      }
   catch (...)
      {
      }
   }
```

但这还不够。如果*value_的析构函数抛了异常，operator delete 不会被调用了来释放*value_的内存。我们需要加上行为安全：

```
~wrapper() throw()
   {
   try
```

```
    {
    delete value_;
    }
  catch (...)
    {
    operator delete(value_);
    }
  }
```

仍然没结束。C++标准运行库申明的 operator delete 为

```
void operator delete(void *) throw();
```

它是不抛异常了，但自定义的 operator delete 可没说不抛。要想超级安全，我们应该写：

```
~wrapper() throw()
  {
  try
    {
    delete value_;
    }
  catch (...)
    {
    try
      {
      operator delete(value_);
      }
    catch (...)
      {
      }
    }
  }
```

但这还存在危险。语句

```
delete value_;
```

隐含调用了 operator delete。如果它抛了异常，我们将进入 catch 块，一步步执行下去并再次调用同样的 operator delete！我们将程序连续暴露在同样的异常下。这不会是个好程序的。

最后，记住：operator delete 在被 new 出对象的构造函数抛异常时被隐含调用。如果这个被隐含调用的 operator delete 也抛了异常，程序将处于两次异常状态并调用 terminate()。

原则：不要在一个可能在异常正被处理过程被调用的函数中抛异常。尤其是，不要从下列情况下抛异常：

- destructors
- operator delete
- operator delete[]

几个小习题：用 auto_ptr 代替 value_，然后重写 wrapper 的构造函数，并决定其虚构

函数的角色（如果需要的话），条件是必须保持异常安全。

14.6 题外话

我本准备一次完成异常安全的。但现在是第二部分，并仍然有足够的素材写成第三部分（我发誓那是最后的部分）。下次，我将讨论 get()和 set()上的异常安全问题，和今天的内容同样精彩。

回到目录

15. 模板安全（续）

在异常安全的第二部分，我讲了在构造函数和析构函数中导致资源泄漏的问题。这次将探索另外两个问题。并且以推荐读物列表结束。

15.1 Problem #2：get

上次，我定义 X::get()为：

```
T get()
   {
   return *value_;
   }
```

这个定义有点小小的不足。既然 get()不改变 wrapper 对象，我应该将它申明为 const 成员的：

```
T get() const
   {
   return *value_;
   }
```

get()返回了一个 T 的临时对象。这个临时对象通过 T 的拷贝构造函数根据*value_隐式生成的，而这个构造函数可能抛异常。要避开这点，我们应该将 get()修改为不返回任何东西：

```
void get(T &value) const throw()
   {
  value = *value_;
   }
```

现在，get()接受一个事先构造好的 T 对象的引用，并通过引用“返回”结果。因为 get()现在不调用 T 的构造函数了，它是异常安全的了。

真的吗？

很不幸，答案是“no”。我们只是将一个问题换成了另外一个问题而已，因为语句

```
value = *value_;
```

实际上是

```
value.operator=(*value_);
```

而它可能抛异常。更完备的解决方法是

```
void get(T &value) const throw()
   {
   try
      {
```

```
      value = *value_;
      }
   catch (...)
      {
      }
   }
```

现在，get()不会将异常漏出去了。

不过，工作还没完成。在 operator=给 value 赋值时抛异常的话，value 将处于不确定状态。get()想要有最大程度的健壮接口的话，它必须两者有其一：

- value 根据*value_进行了完全设置，或
- value 没有被改变。

这两条要将我们弄跳起来了：无论我们用什么方法来解决这个问题，我们都必须调用 operator=来设置 value，而如果 operator=抛了异常，value 将只被部分改变。

我们的这个强壮接口看起来美却不实在。我们无法简单地实现它，只能提供一个弱些的承诺了：

- value 根据*value_进行了完全设置，或
- value 处于一个不确定的（错误）状态。

但还有一个问题没解决：让调用者知道回传的 value 是否是“好的”。一个可能的解决方法（也很讽刺的）是抛出一个异常。另外一个可能方法，也是我在这儿采用的方法是返回一个错误码。

修改后的 get()是：

```
bool get(T &value) const throw()
   {
   bool error(false);
  try
     {
     value = *value_;
     }
  catch (...)
     {
      error = true;
     }

   return error;
   }
```

提供了一个较弱的承诺的这个新接口是安全的。它行为安全吗？是的。wrapper 所拥有的唯一资源是分配给*value_的内存，而它是受保护的，即使 operator=抛了异常。

符合最初的说明，get()有了一个健壮的异常安全承诺，即使 T 没有这个承诺。最终，我们过于加强了 get()的承诺（这取决于 value），而应该将它降低到 T 的承诺层次。我们用一个警告修正 get()的承诺，基于我们不能控制或不能预知 T 的状态。In the end, we over-committed get's guarantee (the determinism of value), and had to bring it down to T's level. We amended get's contract with a caveat, based on conditions in T we couldn't control or predict.

原则：程序的健壮性等于它最弱的承诺。尽可能提供最健壮的承诺，同时在行为和接口

上。

推论：如果你自己的接口的承诺比其他人的接口健壮，你通常必须将你的接口减弱到相匹配的程度。

15.2 Problem #3：set

我们现在的 X::set()的实现是：

```
void set(T const &value)
   {
   *value_ = value;
   }
```

（和 get()不同，set()确实修改 wrapper 对象，所以不能申明为 cosnt。）

语句

```
*value_ = value;
```

应该看起来很熟悉：她只是前面 Problem #2 中提到的语句

```
value = *value_;
```

的反序。注意到这个变化，Problem #3 的解决方案就和 Problem #2 的一样了：

```
bool set(T const &value) throw()
   {
   bool error(false);
   try
      {
     *value = value_;
      }
   catch (...)
      {
      error = true;
      }
   return error;
   }
```

和我们在 get()中回传 value 遇到的问题一样：如果 operator=抛了异常，我们无法知道*value_的状态。我们对 get()的承诺的警告在这儿同样适用。

get()和 set()现在有这同样的操作但不同的用途：get()将当前对象的值赋给另外一个对象，而 set()将另外一个对象的值赋给当前对象。由于这种对称性，我们可以将共同的代码放入一个 assign()函数：

```
static bool assign(T &to, T const &from) throw()
   {
   bool error(false);
   try
      {
      to = from;
      }
```

```
   catch (...)
      {
      error = true;
      }
   return error;
   }
```

使用了这个辅助函数后，get()和 set()缩短为

```
bool get(T &value) const throw()
   {
   return assign(value, *value_);
   }

bool set(T const &value) throw()
   {
   return assign(*value_, value);
   }
```

15.3 最终版本

wrapper 的最终版本是

```
template <typename T>
class wrapper
   {
public:
   wrapper() throw()
         : value_(NULL)
      {
      try
         {
         value_ = new T;
         }
      catch (...)
         {
         }
      }
   ~wrapper() throw()
      {
      try
         {
         delete value_;
         }
      catch (...)
         {
```

```
            operator delete(value_);
            }
         }
      bool get(T &value) const throw()
         {
         return assign(value, *value_);
         }
      bool set(T const &value) throw()
         {
         return assign(*value_, value);
         }
private:
      bool assign(T &to, T const &from) throw()
         {
         bool error(false);
         try
            {
            to = from;
            }
         catch (...)
            {
            error = true;
            }
         return error;
         }
      T *value_;
      wrapper(wrapper const &);
      wrapper &operator=(wrapper const &);
      };
```

（哇！52 行，原来只有 20 行的！而且这还只是一个简单的例子。）

注意，所有的异常处理函数只是吸收了那些异常而没有做任何处理。虽然这使得 wrapper 异常安全，却没有纪录下导致这些异常的原因。

我在 Part13 中讲的在构造函数上的相冲突的原则在这儿同样适用。异常安全是不够的，并且实际上是达不到预期目的的，如果它掩盖了最初的异常状态的话。同时，如果异常对象在被捕获前就弄死了程序的话，大部分的异常恢复方案都将落空。最后，良好的设计必须满足下两个原则：

- 通过异常对象的存在来注视异常状态，并适当地做出反应。
- 确保创造和传播异常对象不会造成更大的破坏。（别让治疗行为比病本身更糟糕。）

15.4 其它说法

在过去 3 部分中，我剖析了异常安全。我强烈建议你读一下这些文章：

- The first principles of C++ exception safety come from Tom Cargill's "Exception

Handling: A False Sense of Security," originally published in the November and December 1994 issues of C++ Report. This article, more than any other, alerted us to the true complexities and subtleties of C++ exception handling.

- C++ Godfather Bjarne Stroustrup is writing an exception-safety Appendix for his book The C++ Programming Language (Third Edition) (http://www.research.att.com/~bs/3rd.html). Bjarne's offering a draft version (http://www.research.att.com/~bs/3rd_safe0.html) of that chapter on the Internet.

- I tend to think of exception safety in terms of contracts and guarantees, ideas formalized in Bertrand Meyer's "Design by Contract" (http://www.eiffel.com/doc/manuals/technology/contract/page.html) programming philosophy. Bertrand realizes this philosophy in both his seminal tome Object-Oriented Software Construction (http://www.eiffel.com/doc/oosc.html) and his programming language Eiffel (http://www.eiffel.com/eiffel/page.html).

- Herb Sutter has written the most thorough C++ exception-safety treatise I've seen. He's published it as Items 8-19 of his new book Exceptional C++ (http://www1.fatbrain.com/asp/bookinfo/bookinfo.asp?theisbn=0201615622). If you've done time on Usenet's comp.lang.c++.moderated newsgroup, you've seen Herb's Guru of the Week postings. Those postings inspired the bulk of his book. Highly recommended.

- Herb's book features a forward written by Scott Meyers. Scott covers exception safety in Items 9-15 of his disturbingly popular collection More Effective C++ (http://www1.fatbrain.com/asp/bookinfo/bookinfo.asp?theisbn=020163371X). If you don't have this book, you simply must acquire it; otherwise Scott's royalties could dry up, and he'd have to get a real job like mine.

Scott（在他的 Item14）认为，不应该将异常规格申明加到模板成员上，和我的正相反。事实是无论用不用异常规格申明，总有一部分程序需要保护所有异常，以免程序自毁。Scott 公正地指出不正确的异常规格申明将导致 std::unexpected——这正是他建议你避开的东西；但，在本系列的 Part11，我指出 unexpected 比不可控的异常传播要优越。

最后要说的是，这儿不会只有一个唯一正确的答案的。我相信异常规格申明可以导致更可预知和有限度的异常行为，即使是对于模板。我也得坦率地承认，在异常/模板混合体上我也没有足够经验，尤其是对大系统。我估计还很少有人有这种经验，因为（就我所知）还没有哪个编译器支持 C++标准在异常和模板上的全部规定。

回到目录

16. 指导方针

根据读者们的建议，经过反思，我部分修正在 Part14 中申明的原则：

- 只要可能，使用那些构造函数不抛异常的基类和成员子对象。
- 不要从你的构造函数中抛出任何异常。

这次，我将思考读者的意见，C++先知们的智慧，以及我自己的新的认识和提高。然后将它们转化为指导方针来阐明和引申那些最初的原则。

（关键字说明：我用“子对象”或“被包容对象”来表示数组中元素、无名的基类、有名的数据成员；用“包容对象”来表示数组、派生类对象或有数据成员的对象。）

16.1 C++的精髓

你可能认为构造函数在遇到错误时有职责抛异常以正确地阻止包容对象的构造行为。Herb Sutter 在一份私人信件中写道：

一个对象的生命期始于构造完成。

推论：一个对象当它的构造没有完成时，它从来就没存在过。

推论：通报构造失败的唯一方法是用异常来退出构造函数。

我估计你正在做这种概念上就错误的事（“错”是因为它不符合 C++的精髓），而这也正是做起来困难的原因。

“C++的精髓”是主要靠口头传授的 C++神话。它是我们最初的法则，从 ISO 标准和实际中得出的公理。如果没有存在过这样的 C++精髓的圣经，混乱将统治世界。Given that no actual canon for the Spirit exists, confusion reigns over what is and is not within the Spirit, even among presumed experts.

C 和 C++的精髓之一是“trust the programmer”。如同我写给 Herb 的：

最终，我的“完美”观点是：在错误的传播过程中将异常映射为其它形式应该是系统设计人员选定的。这么做不总是最佳的，但应该这么做。C++最强同时也是最弱的地方是你可以偏离你实际上需要的首选方法。还有一些其它被语言许可的危险影行为，取决于你是否知道你正在做什么。In the end, my "perfect" objective was to map exceptions to some other form of error propagation should a designer choose to do so. Not that it was always best to do so, but that it could be done. One of the simultaneous strengths/weaknesses of C++ is that you can deviate from the preferred path if you really need to. There are other dangerous behaviors the language tolerates, under the assumption you know what you are doing.

C++标准经常容忍甚至许可潜在的不安全行为，但不是在这个问题上。显然，认同程序员的判断力应该服从于一个更高层次的目的（Apparently, the desire to allow programmer discretion yields to a higher purpose）。Herb 在 C++精髓的第二个表现形式上发现了这个更高层次的目的：一个对象不是一个真正的对象（因此也是不可用的），除非它被完全构造（意味着它的所有要素也都被完全构造了）。

看一下这个例子：

```
struct X
   {
   A a;
   B b;
   C c;
```

```
   void f();
   };

try
   {
   X x;
   x.f();
   }
catch (...)
   {
   }
```

这里，A、B 和 C 是其它的类。假设 x.a 和 x.b 的构造完成了，而 x.c 的构造过程中抛了异常。如我们在前面几部分中看到的，语言规则规定执行这样的序列：

- x 的构造函数抛了异常
- x.b 的析构函数被调用
- x.a 的析构函数被调用
- 控制权交给异常处理函数

这个规则符合 C++的精髓。因为 x.c 没有完成构造，它从未成为一个对象。于是，x 也从未成为一个对象，因为它的一个内部成员（x.c）从没存在过。因为没有一个对象真的存在过，所以也没有哪个需要正式地析构。

现在假设 x 的构造函数不知怎么控制住了最初的异常。在这种情况下，执行序列将是：

- x.f()被调用
- x.c 的析构函数被调用
- x.b 的析构函数被调用
- x.a 的析构函数被调用
- x 的析构函数被调用
- 控制权跳过异常处理函数向下走

于是异常将会允许析构那些从没被完全构造的对象（x.c 和 x)。这将造成自相矛盾：一个死亡的对象是从来都没有产生过的。通过强迫构造函数抛异常，语言构造避免了这种矛盾。

16.2 C++的幽灵

前面表明一个对象当且仅当它的成员被完全构造时才真的存在。但真的一个对象存在等价于被完全构造？尤其 x.c 的构造失败“总是”如此恶劣到 x 必须在真的在被产生前就死亡？

在 C++语言有异常前，x 的定义过程必定成功，并且 x.f()的调用将被执行。代替抛异常的方法，我们将调用一个状态检测函数：

```
X x;
if (x.is_OK())
   x.f();
```

或使用一个回传状态参数：

```
bool is_OK;
X x(is_OK);
if (is_OK)
```

```
x.f();
```

在那个时候，我们不知何故在如 x.c 这样的子对象的构造失败时没有强调：这样的对象从没真的存在过。那时的设计真的这么根本错误(而我们现在绝不允许的这样行为了)？ C++的精髓真的在那时是不同的？或者我们生活在梦中，没有想到过 x 真的没有成形、没有存在过？

公正地说，这个问题有点过份，因为 C++语言现在和过去相比已不是同样的语言。将老的（异常支持以前）的 C++当作现在的 C++如同将 C 当作 C++。虽然它们有相同的语法，但语意却是不相同的。看一下：

```
struct X
   {
   X()
      {
      p = new T; // assume 'new' fails
      }
   void f();
   };

X x;
x.f();
```

假设 new 语句没有成功分配一个 T 对象。异常支持之前的编译器（或禁止异常的现代编译器）下，new 返回 NULL,x 的构造函数和 x.f()被调用。但在异常允许后，new 抛异常,x 构造失败，x.f()没有被调用。同样的代码，非常不同的含意。

在过去，对象没有自毁的能力，它们必须构造，并且依赖我们来发现它的状态。它们不处理构造失败的子对象。并且，它们不调用标准运行库中抛异常的库函数。简而言之，过去的程序和现在的程序存在于不同的世界中。我们不能期望它们对同样的错误总有同样的反应。

16.3 这是你的最终答案吗？

我现在相信 C++标准的行为是正确的：构造函数抛异常将析构正在处理的对象及其包容对象。我不知道 C++标准委员会制订这个行为的精确原因，但我猜想是：

- 部分构造的对象将导致一些微妙的错误,因为它的使用者对其的构造程度的假设超过了实际。同样的类的不同对象将会有出乎意料的和不可预测的不同行为。
- 编译器需要额外的纪录。当一个部分构造的对象消失时，编译器要避免对它及它的部分构造的子对象调用析构函数。
- 对象被构造和对象存在的等价关系将被打破，破坏了 C++的精髓。

16.4 对对象的使用者的指导

异常是对象的接口的一部分。如果能够，事先准备好接口可能抛的异常集。如果一个接口没有提供异常规格申明，而且又不能从其它地方得知其异常行为，那么假设它可能在任何时候抛任意的异常。

换句话说，准备好捕获或至少要过滤所有可能的异常。不要让任何异常在没有被预料到的情况下进入或离开你的代码；即使你只是简单地传递或重新抛出异常，也必须是经过认真选择的。

16.5 构造函数抛异常

准备好所有子对象的构造函数可能抛的异常的异常集，并在你的构造函数中捕获它们。如：

```
struct A
   {
   A() throw(char, int);
   };

struct B
   {
   B() throw(int);
   };

struct C
   {
   C() throw(long);
   };

struct X
   {
   A a;
   B b;
   C c;
   X();
   };
```

子对象构造函数的异常集是{char,int,long}。它就是 X 的构造函数遭遇的可能异常。如果 X 的构造函数未经过滤就传递这些异常，它的异常规格申明将是

```
X() throw(char, int, long);
```

但使用 function try 块，构造函数可以将这些异常映射为其它类型：

```
X() throw(unsigned)
   try
      {
      // ... X::X body
      }
   catch (...)
      {
      // map caught sub-object exceptions to another type
      throw 1U; // type unsigned
```

```
    }
```

如同前面的部分所写，用户的构造函数不能阻止子对象的异常传播出去，但能控制传递出去的类型，通过将进入的异常映射为受控的传出类型（这儿是 unsigned）。

16.6 构造函数不抛异常

如果没有子对象的构造函数抛异常，其异常集是空，表明包容对象的构造函数不会遇到异常。唯一能确定你的构造函数不抛异常的办法是只包容不抛异常的子对象。

如果必须包容一个可能抛异常的子对象，但仍然不想从你自己的构造函数中抛出异常，考虑使用被叫做 Handle Class 或 Pimpl 的方法（“Pimpl”个双关语：pImpl 或“pointer to implementation”）。长久以来被用作减短编译时间的技巧，它也提高异常安全性。

回到前面的例子：

```
class X
  {
public:
  X();
  // ...other X members
private:
  A a;
  B b;
  C c;
  };
```

根据这种方法，必须将 X 分割为两个独立的部分。第一部分是被 X 的用户引用的“公有”头文件：

```
struct X_implementation;

class X
  {
public:
  X() throw();
  // ...other X members
private:
  struct X_implementation *implementation;
  };
```

而第二部分是私有实现

```
struct X_implementation
  {
  A a;
  B b;
  C c;
  };

X::X() throw()
```

```
{
try
   {
   implementation = new X_implementation;
   }
catch (...)
   {
   // ... Exception handled, but not implicitly rethrown.
   }
}

// ...other X members
```

X 的构造函数捕获了构造*implementation 过程（也就是构造 a、b 和 c 的过程）中的所有异常。更进一层，如果数据成员变了，X 的用户不需要重新编译，因为 X 的头文件没有变化。

（反面问题：如果 X::X 捕获了一个异常，*implementation 及至少子对象 a/b/c 中的一个没有完全构造。但是，包容类 X 的对象作为一个有效实体延续了生命期。这个 X 的部分构造的对象的存在违背 C++精髓吗？）

许多 C++的指导手册讨论这个方法，所以我不在这儿详述了。一个极其详细的讨论出现在 Herb Sutter 的著作《Exceptional C++》的 Items26－30 上。

16.7 对对象提供者的指导

不要将异常体系等同于一种错误处理体系，认为它和返回错误码或设置全局变量处在同一层次上。异常根本性地改变了它周围的代码的结构和意义。它们临时地改变了程序的运行期语意，跳过了一些通常都运行的代码，并激活其它从没被运行的代码。它们强迫你的程序回应和处理可导致程序死亡的错误状态。

因此，异常的特性和简单的错误处理大不相同。如果你不希望这些特性，或不理解这些特性，或不想将这些特性写入文档，那么不要抛异常，使用其它的错误处理体系。

如果决定抛异常，必须明白全部的因果关系。明白你的决定对使用你的代码的人有巨大的潜在影响。你的异常是你的接口的一部分；你必须在文档中写入你的接口将抛什么异常，什么时候抛，以及为什么抛。并将这文档在异常规格申明出注释出来。

16.8 构造函数抛异常

如果你的构造函数抛异常，或你（直接地或间接地）包容的某个子对象抛异常，包容你的对象的用户对象也将抛异常并因此构造失败。这就是重用你的代码的用户的代价。要确保这个代价值得。

你没有被强迫要在构造函数里抛异常，老的方法仍然有效的。当你的构造函数遇到错误时，你必须判断这些错误是致命的还是稍有影响。抛出一个构造异常传递了一个强烈的信息：这个对象被破坏且无法修补。返回一个构造状态码表明一个不同信息：这个对象被破坏但还具有功能。

不抛异常只是因为它是一个时髦的方法：在一个对象真的不能或不该生存时，推迟其自毁。

16.9 过职

别让你的接口过职。如果知道你的接口的精确异常集，将它在异常规格申明中列举出来。否则，不提供异常规格申明。没有异常规格申明比撒谎的异常规格申明好，因为它不会欺骗用户。

这条规则的可能例外是：模板异常。如前三部分所写，模板的编写者通常不知道可能抛出的异常。如果你的模板不提供异常规格申明，用户将降低安全感和信心。如果你的模板有异常规格申明你必须：

- 要么使用前面看过的异常安全的技巧来确保异常规格申明是精确的
- 要么在文档中写下你的模板只接受有确定特性的参数类型，并警告其它类型将导致失控（with the caveat that other types may induce interface-contract violations beyond your control）。

16.10 必要 vs 充分

不要人为增加你的类的复杂度，只是为了适应所有可能的需求。不是所有对象都会被重用的。如 pet Becker 写给我的：

现在的程序员花了太多的时间来应付可能发生的事情，而他们本应该简单地拒绝的。如果有一个抛异常的好理由的话，大胆地抛异常，并写入文档，不要创造一些精巧的方法来避免抛这些异常。增加的复杂度可能导致维护上的恶梦，超过了错误使用受限版本时遇到的痛苦。

Pete 的说法对析构函数也同样有用。看一下这条原则（从 Part14 引用过来的）：

不要在析构函数中抛异常。

一般来说，符合这条原则比违背它好。但，有时不是这样的：

- 如果你准备让其他人包容你的对象，或至少不禁止别人包容你的对象，那么别在析构函数中抛异常。
- 如果你真的有理由抛异常，并且知道它违背了安全策略，那么大胆地抛异常，在文档中写入原因。

就如同在设计的时候必须考虑异常处理，也必须考虑重用。在析构函数上申明 throw() 是成为一个好的子对象的必要条件，但远不充分。你必须前瞻性地考虑你的代码将遇到什么上下文，它将容忍什么、将反抗什么。如果增加了设计的复杂度，确保这些复杂度是策略的一部分，而不是脆弱的“以防万一”的保险单。

16.11 感谢

（略）

除了一些零星的东西，我已经完成了异常安全的主题！实际上我也几乎完成了异常的专题。下次时间暂停，在三月中将讨论很久前承诺的 C++异常和 Visual C++ SEH 的混合使用。

回到目录

17. C++异常和 Visual C++ SEH 的混合使用

我在 Part2 介绍了 Structured Exception Handling（简称 SEH)。在那时我就说过，SEH 是 window 及其平台上的编译器专有的。它不是定义在 ISO C++标准中的，使用它的程序将不能跨编译器移植。因为我注重于标准兼容和可移植性，所以我对将 windows 专有的 SEH 映射为 ISO 标准 C++的 exception handing（简称 EH）很感兴趣。

同时，我不是 SEH 的专家。对它的了解绝大部分来自于本专栏前面的研究。当我考虑混合使用 SEH 与 EH 时，我猜想解决方法应该是困难的和不是显而易见的。这是它花了我两个星期的原因：我预料到需要额外的时间来研究和试验。

很高兴，我完全错了。我不知道的是 Visual C++运行期库直接支持了绝大部分我所想要的东西。不用创造新的方法了，我可以展示你 Visual C++已经支持了的东西，以及改造为所需要的东西的方法。基于这个目的，我将研究同一个例子的四个不同版本。

17.1 Version 1：定义一个转换函数

捆绑 SEH 和 EH 的方法分两步：

- 一个用户自定义的转换函数来捕获 SEH 的异常并将它映射为 C++的异常。
- 一个 Visual C++运行期库函数来安装这个转换函数

用户自定义的转换函数必要有如下形式：

```
void my_translator(unsigned code, EXCEPTION_POINTERS *info);
```

转换函数接受一个 SEH 异常（通过给定的异常 code 和 info 来定义的)。然后抛出一个 C++异常，以此将传入的 SEH 异常映射为向外传的 C++异常。这个 C++异常将出现在原来的 SEH 异常发生点上并向外传播。

这个机制非常象 std::set_terminate()和 std::set_unexpected()。要安装转换函数，要调用 Visual C++库函数_set_se_translator()。这个函数申明在头文件 eh.h 中：

```
typedef void (*_se_translator_function)(unsigned, EXCEPTION_POINTERS *);

_se_translator_function _set_se_translator(_se_translator_function);
```

它接受一个指向新转换函数的指针，返回上次安装的指针。一旦安装了一个转换函数，前一次的就丢失了；任何时候只有一个转换函数有效。(在多线程程序中，每个线程有一个独立的转换函数。)

如果还没安装过转换函数，第一次调用_set_se_translator()返回值可能是(也可能不是）NULL。也就是说，不能不分青红皂白就通过其返回的指针调用函数。很有趣的，如果返回值是 NULL，而你又通过此 NULL 调用函数，将产生一个 SEH 异常，并且进入你刚刚安装的转换函数。

一个简单的例子：

```
#include <iostream>
using namespace std;

int main()
   {
   try
      {
      *(int *) 0 = 0; // generate Structured Exception
```

```
    }
  catch (unsigned exception)
    {
    cout << "caught C++ exception " << hex << exception << endl;
    }
  return 0;
  }
```

运行它的话，这个控制台程序将导致如此一个 windows messagebox：

它是由于一个未被捕获的 SEH 异常传递到程序外面造成的。

现在，增加一个异常转换函数，并将 Visual C++运行库设为使用这个转换函数：

```
#include <iostream>
using namespace std;

#include "windows.h"

static void my_translator(unsigned code, EXCEPTION_POINTERS *)
  {
  throw code;
  }

int main()
  {
  _set_se_translator(my_translator);
  try
    {
    *(int *) 0 = 0; // generate Structured Exception
    }
  catch (unsigned exception)
    {
    cout << "caught C++ exception " << hex << exception << endl;
    }
  return 0;
  }
```

再运行程序。现在将看到：

```
caught C++ exception c0000005
```

my_translator()截获了 SEH 异常，并转换为 C++异常，其类型为 unsigned，内容为 SEH 异常码（本例中为 C0000005h，它是一个非法读取错误）。因为这个 C++异常出现在原来的 SEH 异常发生点，也就说在 try 块中，所以被 try 块的异常处理函数捕获了。

17.2 Version 2：定义一个转换对象

上面的例子非常简单，将每个 SEH 异常转换为一个 unsigned 值。实际上，你可能需要一个比较复杂的异常对象：

```
#include <iostream>
using namespace std;

//#include "windows.h"

#include "structured_exception.h"

/*static void my_translator(unsigned code, EXCEPTION_POINTERS *)
   {
   throw code;
   }*/

int main()
   {
//_set_se_translator(my_translator);
   structured_exception::install();
   try
      {
      *(int *) 0 = 0; // generate Structured Exception
      }
   catch (structured_exception const &exception)
      {
      cout << "caught C++ exception " << hex << exception.what()
           << " thrown from " << exception.where() << endl;
      }
   return 0;
   }
```

这个例子抛出了一个用户自定义类型（structured_exception）的C++异常。为了让这个例子更具实际意义，也更方便阅读，我将 structured_exception 的申明放到了头文件 structured_exception.h 中：

```
#if !defined INC_structured_exception_
    #define  INC_structured_exception_

#include "windows.h"

class structured_exception
   {
public:
   structured_exception(EXCEPTION_POINTERS const &) throw();
   static void install() throw();
   unsigned what() const throw();
   void const *where() const throw();
private:
```

```
   void const *address_;
   unsigned code_;
   };

#endif // !defined INC_structured_exception_
```

其实现文件为：

```
#include "structured_exception.h"

#include "eh.h"

//
//  ::
//
static void my_translator(unsigned, EXCEPTION_POINTERS *info)
   {
   throw structured_exception(*info);
   }

//
//  structured_exception::
//
structured_exception::structured_exception
      (EXCEPTION_POINTERS const &info) throw()
   {
   EXCEPTION_RECORD const &exception = *(info.ExceptionRecord);
   address_ = exception.ExceptionAddress;
   code_ = exception.ExceptionCode;
   }

void structured_exception::install() throw()
   {
   _set_se_translator(my_translator);
   }

unsigned structured_exception::what() const throw()
   {
   return code_;
   }

void const *structured_exception::where() const throw()
   {
   return address_;
   }
```

这些函数的意义是：

- my_translator()是异常转换函数。我把它从 main 文件中移到这儿。于是，main 文件不再需要包含 windows.h 了。
- install()将运行器库的全局转换函数设置为 my_translator()。
- structured_exception 的构造函数接收并解析 SEH 异常的信息。
- what()返回 SEH 异常的异常码。
- where()返回 SEH 异常发生的地点。注意，where()的返回类型是 void const *，虽然 C++标准不同意将代码地址转换为 void 指针。我只是重复了 Micorsoft 的用法，因为 Visual C++运行库将地址存在了 SEH 异常的 EXCEPTION_RECORD 的一个 void *成员中了。

编译并链接这三个文件。运行结果是：

```
caught C++ exception c0000005 thrown from 0040181D
```

（其中的代码地址值在你的系统上可能有所不同。）

17.3 Version 3：模仿 C++标准运行库

在 my_translator()中，所有的 SEH 异常映射为同样的 structured_exception 类型。这使得异常容易被捕获，因为它们匹配于我们的唯一的异常处理函数：

```
catch (structured_exception const &exception)
```

虽然捕获了异常，但我们没有办法事先知道异常的类型。唯一能做的是运行期查询，调用这个异常的 what()成员：

```
catch (structured_exception const &exception)
   {
   switch (exception.what())
      {
      case EXCEPTION_ACCESS_VIOLATION:
         // ...
      case EXCEPTION_INT_DIVIDE_BY_ZERO:
         // ...
      case EXCEPTION_STACK_OVERFLOW:
         // ...
      // ...
   }
```

这样的查询需要 windows.h 中的信息，以知道最初的 SEH 异常码的含意。这样的需求违背了 structured_exception 的抽象原则。此外，switch 语句也经常违背了多态的原则。从用户代码的角度看，你通常应该用继承和模板来实现它。

C++标准运行库在这方面提供了一些指导。如我在 Part3 中勾画的，头文件<stdexcept>定义了一个异常类层次，std::exception 是根结点。这个根类定义了虚成员 what()，它返回一个编译器自定义的 NTBS（C++标准中是“以 NULL 结束的字符串”）。每个继承类指定自己的 what()的返回值。虽然 C++标准没有规定这些值的内容，但我相信标准委员会打算用这个字符串来描述异常的类型或含意的。

根据这种精神，standard_exception 的申明是：

```
#if !defined INC_structured_exception_
    #define  INC_structured_exception_
```

```
#include "eh.h"
#include "windows.h"

class structured_exception
    {
public:
   structured_exception(EXCEPTION_POINTERS const &) throw();
   static void install() throw();
   virtual char const *what() const throw();
   void const *where() const throw();
private:
   void const *address_;
   //unsigned code_;
   };

class access_violation : public structured_exception
   {
public:
   access_violation(EXCEPTION_POINTERS const &) throw();
   virtual char const *what() const throw();
   };

class divide_by_zero : public structured_exception
   {
public:
   divide_by_zero(EXCEPTION_POINTERS const &) throw();
   virtual char const *what() const throw();
   };

#endif // !defined INC_structured_exception_
```

实现是：

```
#include <exception>
using namespace std;

#include "structured_exception.h"

#include "windows.h"

//
//  ::
//
static void my_translator(unsigned code, EXCEPTION_POINTERS *info)
   {
```

```
   switch (code)
      {
      case EXCEPTION_ACCESS_VIOLATION:
         throw access_violation(*info);
         break;
      case EXCEPTION_INT_DIVIDE_BY_ZERO:
      case EXCEPTION_FLT_DIVIDE_BY_ZERO:
         throw divide_by_zero(*info);
         break;
      default:
         throw structured_exception(*info);
         break;
      }
   }

//
//  structured_exception::
//
structured_exception::structured_exception
       (EXCEPTION_POINTERS const &info) throw()
   {
   EXCEPTION_RECORD const &exception = *(info.ExceptionRecord);
   address_ = exception.ExceptionAddress;
   //code_ = exception.ExceptionCode;
   }

void structured_exception::install() throw()
   {
   _set_se_translator(my_translator);
   }

char const *structured_exception::what() const throw()
   {
   return "unspecified Structured Exception";
   }

void const *structured_exception::where() const throw()
   {
   return address_;
   }

//
//  access_violation::
//
```

```
access_violation::access_violation
      (EXCEPTION_POINTERS const &info) throw()
      : structured_exception(info)
   {
   }

char const *access_violation::what() const throw()
   {
   return "access violation";
   }

//
//  divide_by_zero::
//
divide_by_zero::divide_by_zero
      (EXCEPTION_POINTERS const &info) throw()
      : structured_exception(info)
   {
   }

char const *divide_by_zero::what() const throw()
   {
   return "divide by zero";
   }
```

注意：

- 那些本来在用户的异常处理函数中的 switch 语句，现在移到了 my_translator()中。不再是将所有 SEH 异常映射为单个值(如 version 1 中)或单个类型的对象(version 2)，现在的 my_translator()将它们映射为多个类型的对象（取决于运行时的实际环境)。
- structured_exception 成为了一个基类。我没有让它成为纯虚类，这是跟从了 C++标准运行库的引导（std::exception 是个实体类)。
- 我没有定义任何析构函数，因为编译器隐含提供的的析构函数对这些简单类足够了。如果我定义了析构函数，它们将需要定义为 virtual。
- what()现在返回了一个用户友好的文本，取代了原来的 SEH 异常码。
- 因为我不再测试和显示这些代码，我去掉了数据成员 code_。这使得 structured_exception 对象的大小减小了。(别太高兴：节省的空间又被新增的 vptr 指针抵销了，因为有了虚函数。)
- 因为模板方式更好，你应该放弃这种继承模式的。我将它留给你作为习题。

试一下新的方案，将 main 文件改为：

```
#include <iostream>
using namespace std;

#include "structured_exception.h"
```

```
int main()
   {
   structured_exception::install();
   //
   //  discriminate exception by dynamic type
   //
   try
      {
      *(int *) 0 = 0; // generate Structured Exception
      }
   catch (structured_exception const &exception)
      {
      cout << "caught " << exception.what() << endl;
      }
   //
   //  discriminate exception by static type
   //
   try
      {
      static volatile int i = 0;
      i = 1 / i; // generate Structured Exception
      }
   catch (access_violation const &)
      {
      cout << "caught access violation" << endl;
      }
   catch (divide_by_zero const &)
      {
      cout << "caught divide by zero" << endl;
      }
   catch (structured_exception const &)
      {
      cout << "caught unspecified Structured Exception" << endl;
      }
   return 0;
   }
```

再次运行，结果是：

```
caught access violation
caught divide by zero
```

17.4 Version 4：匹配于C++标准运行库

我们所有的standard_exception继承类都提供公有的成员

```
virtual char const *what() const;
```
来识别异常的动态类型。我不是随便选取的函数名：所有的 C++标准运行库中的 std::exception 继承类为同样的目的提供了同样的公有成员。并且，what()是每个继承类的唯一的多态函数。

你可能已经注意到：

```
#include <exception>

class structured_exception : public std::exception
   {
public:
   structured_exception(EXCEPTION_POINTERS const &info) throw();
   static void install() throw();
   virtual char const *what() const throw();
   void const *where() const throw();
private:
   void const *address_;
   };
```

因为 structured_exception 现在也是一个 std:exception，我们可以用一个异常处理函数来同时捕获这个异常族：

```
catch (std::exception const &exception)
```

并且用同样的多态函数来获取异常的类型：

```
catch (std::exception const &exception)
   {
   cout << "caught " << exception.what();
   }
```

用这样的方案，SEH 异常能够表现得与标准 C++的固有行为一致。同时，我们仍然能够特殊对待 structured_exceptions 并访问它的特殊成员：

```
catch (structured_exception const &exception)
   {
   cout << "caught Structured Exception from " << exception.where();
   }
```

当然，如果你想放弃没有出现在 std::exception 继承体系中的类成员，如 where()，你完全可以不使用基类 structured_exception，而是直接从 std::exception 继承出 access_violation 等类。例如：一个 divide-by-zero 异常表示了一个程序值域控制错误，也就是说是个逻辑错误。你所以想直接从 std::logic_error 甚至是 std::out_of_range 派生 devide_by_zero 类。

我建议你看一下 C++标准 subclause 19.1 （“Exception classes”）以更好地理解 C++标准运行库的异常继承体系，以及如何更好地将你的自定义异常熔入此继承体系。

17.5 总结束

（略）

回到目录