# 第I部分 构建稳定的系统

软件工程师普遍面临一个挑战：验证代码的时间与编写代码的时间相比，往往不相上下。追求卓越的软件质量是每个开发者的目标，而要想达到这一目标，在问题出现之前就将其消弭于无形之中就显得至关重要。此外，一个真正健壮的软件系统，即使遭遇预料之外的环境挑战，也能游刃有余，从容应对。

C++引入了“异常”机制，因而不仅支持高级错误处理，还能避免在代码中充斥过多的错误处理逻辑。第1章将展示如何正确使用异常来实现行为良好的软件，介绍一些设计原则来指导你编写“异常安全”的[[1]](#footnote-1)代码。第2章则展示在代码发布之前如何利用单元测试和调试技术来最大程度地提高代码质量。经验丰富的软件工程师会使用断言来表达并强制执行程序中的“不变量”[[2]](#footnote-2)。此外，第2章还要介绍一个用于支持单元测试的简单框架。

# 第1章 异常处理

为了增强代码的健壮性，最有力的手段之一就是增强错误恢复能力。

遗憾的是，忽略错误情况几乎成了一种默认的做法，大家似乎都在刻意否认错误的存在。毫无疑问，其中一个原因是检查多种错误所带来的繁琐和代码膨胀。例如，printf()函数会返回成功打印的字符数，但实际上鲜有人去检查这个值。人们不喜欢代码量的增加，更不喜欢由此而来的代码可读性的降低。

C语言在错误处理方面有一个显著问题——过于紧密的耦合性。函数的调用者必须将错误处理代码与被调用函数紧密相连，这样一来，就导致代码变得复杂、难以维护且容易引入新的错误。[[3]](#footnote-3)

C++语言的重要特性之一是异常处理，这是一种更好的思考和处理错误的方式。借助于异常处理，我们可以获得以下好处：

1. 编写错误处理代码变得不再那么枯燥乏味，也不会与“正常”的代码混在一起。我们先编写**希望**执行的代码，然后在单独的部分编写处理问题的代码。如果需要多次调用某个函数，那么只需要集中于一个地方处理来自该函数的错误。
2. 错误再也无法被忽视。如果一个函数需要向调用者发送错误信息，那么它可以“抛出”一个表示该错误的对象。如果调用者没有“捕获”并处理这个错误，它就会传递到上一层的动态作用域。以此类推，直到错误被捕获，或者程序因为没有相应的异常处理程序而终止。[[4]](#footnote-4)

本章要探讨C语言处理错误的方式，要讨论为什么这种方式对C语言来说不够理想，要解释为什么这种方式完全不适合C++语言。最后，本章还要介绍 try、throw和catch等C++异常处理关键字。

## 1.1 传统的错误处理

在本书共两卷的大多数示例中，我们都按照assert()的设计初衷来使用它：在开发期间进行调试，并通过#define NDEBUG在最终发行版中禁用调试代码。至于运行时的错误检查，则使用在本书第1卷第9章中开发，并在这一卷附录B中重复介绍的assure()函数和require()函数（均在require.h中定义）。这些函数提供一种方便的方式来表达这样的意思：“这里有问题，你可能希望用更复杂的代码来处理它，但在本示例中无需为此分心。”require.h中定义的函数对小型程序而言可能已经足够，但对于更复杂的程序，则可能需要编写更复杂的错误处理代码。

如果明确知道要做什么，那么错误处理就可以相当简单直接，因为当前已经拥有在那个上下文中处理错误所需的全部必要信息。在这种情况下，直接在出错的位置处理即可。

如果当前上下文没有提供处理错误所需要的足够充分的信息，那么问题就来了。在这种情况下，必须将错误信息传递给另一个有足够信息的上下文。C语言支持以三种方式处理这种情况。

1. 从函数返回错误信息。如果不能用返回值来表示错误，就设置一个全局错误状态标志（标准C使用errno和perror()来做这件事情）。如前所述，由于每次函数调用都必须进行冗长且混乱的错误检查，所以程序员很可能会忽略错误信息。此外，从出现异常情况的函数返回，这本身就可能有点不合理。
2. 使用鲜为人知的标准C库信号处理系统，这需要用到signal()函数（用于确定事件发生时的行为）和raise()函数（用于生成事件）。同样地，这种方法具有高耦合性，因为如果要使用会生成信号的库，那么库的用户就必须理解并安装适当的信号处理机制。而且，在大型项目中，不同库之间的信号值可能发生冲突。
3. 使用标准C库中的非局部跳转函数：setjmp()和longjmp()。使用setjmp()保存程序中已知的良好状态，如果出现问题，那么longjmp()将恢复那个状态。同样，存储状态的位置与发生错误的位置之间存在高耦合性。

在设计C++语言的错误处理方案时，还需要考虑一个重要的问题：C语言中的信号处理机制以及setjmp()和longjmp()技术不会调用析构函数，因此对象无法得到正确的清理。（事实上，如果longjmp()跳过原本应该调用析构函数的作用域结束位置，会导致程序行为“未定义”。）这会导致我们几乎不可能有效地从异常状况中恢复，因为总会留下一些未被清理的对象，这些对象之后再也无法访问。以下示例使用setjmp()和longjmp()对此进行了演示：

//: C01:Nonlocal.cpp

// setjmp() & longjmp().

#include <iostream>

#include <csetjmp>

using namespace std;

// 代表奥兹魔法世界

class Rainbow {

public:

Rainbow() { cout << "Rainbow()" << endl; }

~Rainbow() { cout << "~Rainbow()" << endl; }

};

jmp\_buf kansas; // 保存跳转回的点

// 模拟桃乐丝在奥兹国的经历

void oz() {

Rainbow rb; // 创建魔法世界

for(int i = 0; i < 3; i++)

cout << "没有哪个地方比家更好" << endl;

// 跳回故事的开头

longjmp(kansas, 47);

}

int main() {

// 保存当前执行上下文

if(setjmp(kansas) == 0) {

cout << "龙卷风、女巫、芒奇金人..." << endl;

oz(); // 进入魔法世界

} else {

// 从梦境中醒来

cout << "艾姆阿姨! "

<< "我做了个最奇怪的梦..."

<< endl;

}

} ///:~

setjmp()函数有些奇特，如果直接调用的话，它会将有关当前处理器状态的所有信息（例如指令指针和运行时栈指针的内容）都存储在jmp\_buf中并返回零。在这种情况下，它的行为和普通的函数无异。然而，如果使用同一个jmp\_buf来调用longjmp()，那么就好比再次从setjmp()返回——恢复为setjmp()刚刚执行完毕后的样子。但这一次，返回的是longjmp()的第二个参数。这样一来，就可以检测到当前实际从longjmp()返回的值。可以想象，若使用多个不同的jmp\_buf，则完全可以跳转到程序中多个不同的位置。局部goto（带标签）与非局部goto（使用setjmp()/longjmp()来实现）的区别在于，后者可以返回到运行时栈中事先确定的**任何**位置（即任何调用setjmp()的位置）。

在C++语言中，longjmp()的问题在于它不“尊重”对象，尤其是当它跳出作用域时不会自动调用析构函数。[[5]](#footnote-5)由于在C++语言中必须调用析构函数，所以这样显然是行不通的。事实上，C++标准已经规定，如果使用goto跳入某个作用域（相当于绕过构造函数调用），或者使用longjmp() 离开具有析构函数的一个对象的作用域，那么程序的行为就是未定义的。

## 1.2 抛出异常

如果在代码中遇到了异常情况（也就是说，当前上下文没有足够的信息来帮助自己决定下一步该怎么做），则可以通过创建包含该信息的一个“异常”对象，并将它从当前上下文中“抛出”，从而将有关错误的信息发送到一个更大的上下文中。这就是所谓的**抛出异常**，如下面的例子所示。

//: C01:MyError.cpp {RunByHand}

class MyError {

const char\* const data;

public:

MyError(const char\* const msg = nullptr) : data(msg) {}

};

void f() {

// 我们在这里“抛出”一个异常对象：

throw MyError("发生了不好的事情");

}

int main() {

// 稍后会讲到，需要在这里放一个“try 块”：

f();

} ///:~

在本例中，MyError是一个普通的类，它接收一个char\*作为构造函数的参数。虽然允许抛出任何类型的异常（甚至是内置类型），但我们一般都需要为抛出的异常创建特殊的类。

关键字throw会引发一系列神奇的操作。首先，它会创建所抛出对象的一个拷贝，并从包含throw表达式的那个函数中实际 “返回”该对象——即使函数原本设计的并不是返回该对象类型。我们可以简单地将异常处理想象成一种替代返回机制（但深入推敲的话会发现问题）。还可以通过抛出异常从一个普通的作用域中退出。总之，它会返回一个值，并且退出函数或作用域。

好了，与return语句的相似之处到此为止，因为返回的位置与普通函数调用返回的位置完全不一样。异常处理最终会在代码的一个恰当的部分（称为**异常处理程序[[6]](#footnote-6)**）结束，这部分代码可能远离抛出异常的位置。此外，异常发生时创建的任何局部对象都会被销毁。这种对局部对象的自动清理通常称为**栈展开[[7]](#footnote-7)**。

此外，完全可以根据需要抛出不同类型的对象。我们一般需要会为不同类别的错误抛出不同类型的对象，目的是将错误信息存储在对象中，并用**类名**来加以简单的提示，方便调用者弄清楚如何处理异常。

## 1.3 捕获异常

如前所述，C++异常处理的一个优点是可以在一个地方专注于当前试图解决的问题，并在另一个地方处理来自那段代码的错误。

### 1.3.1 try块

如果在一个函数内部抛出异常（或者被调用的函数抛出异常），那么函数会因为抛出的异常而退出执行。如果不想因为抛出异常而导致函数退出执行，那么可以在函数内部设置一个特殊的块，并在其中尝试解决当前实际的编程问题（它可能产生异常）。这个块称为**try块**，因为要在其中“尝试”各种函数调用。try 块是一个普通的作用域，用关键字try来定义，如下所示：

try {

// 可能产生异常的代码

}

如果仍然采用传统的返回码方式检查错误，则需要在每个函数调用前后添加额外的代码来设置和检查返回值。如果函数需要多次调用，这种做法无疑会大幅增加代码量，同时还会降低可读性。异常处理机制提供了一种更优雅的解决方案：将核心业务逻辑置于try块中，将错误处理逻辑放在catch块中。这种分离使得代码结构显得更清晰，也便于我们定位和处理异常。

### 1.3.2 异常处理程序

当然，抛出的异常最终必须到达某个地方。这个地方就是**异常处理程序**，需要为捕获的每种类型的异常都设置一个异常处理程序。然而，多态性也适用于异常。换言之，一个异常处理程序可以处理某种异常类型及其派生类。

异常处理程序紧跟在try块之后，并由关键字catch标记。

try {

// 可能产生异常的代码

} catch(type1 id1) {

// 处理type1类型的异常

} catch(type2 id2) {

// 处理type2类型的异常

} catch(type3 id3)

// 等等...

} catch(type*N* id*N*)

// 处理type*N*类型的异常

}

// 正常执行从这里恢复...

catch子句的语法类似于接收单个实参的函数。可以在处理程序内部使用标识符（例如id1、id2等），它们就像函数参数一样。当然，如果在处理程序中不需要用到标识符，那么也可以省略它。通常情况下，单是异常类型本身就提供了足够多的信息来帮助我们处理异常。

处理程序必须直接跟在try块之后。一旦抛出异常，异常处理机制就会寻找与异常类型匹配的第一个处理程序，并进入那个catch子句，认为异常已被处理。换言之，一旦找到符合条件的某个catch子句，就会停止对异常处理程序的搜索过程。只有匹配的catch子句才会被执行。完成异常处理后，控制权会从与那个try块关联的最后一个处理程序后恢复。

注意，在try块内部，多个不同的函数调用可能引发相同类型的异常。但是，同类型的异只需要一个异常处理程序。

为了说明try和catch的工作方式，下面是前面非局部跳转示例Nonlocal.cpp的一个变体，它用try块替换了setjmp()调用，并用throw语句替换了longjmp()调用：

//: C01:Nonlocal2.cpp

// 演示异常处理。

#include <iostream>

using namespace std;

// 代表奥兹魔法世界

class Rainbow {

public:

Rainbow() { cout << "Rainbow()" << endl; }

~Rainbow() { cout << "~Rainbow()" << endl; }

};

// 模拟桃乐丝在奥兹国的经历

void oz() {

Rainbow rb;

for(int i = 0; i < 3; i++)

cout << "没有哪个地方比家更好" << endl;

throw 47;

}

int main() {

try {

cout << "龙卷风、女巫、芒奇金人..." << endl;

oz(); // 进入魔法世界

} catch(int) {

cout << "艾姆阿姨! "

<< "我做了个最奇怪的梦..."

<< endl;

}

} ///:~

一旦执行oz()中的throw语句，程序控制就会回溯（backtrace），直至找到接收一个int参数的catch子句。然后，执行将从那个 catch 子句的主体继续。这个程序与Nonlocal.cpp最重要的区别在于，当throw语句导致执行离开oz()函数时，在函数中创建的rb对象的析构函数会被自动调用。

### 1.3.3 终止和恢复

异常处理理论有两种基本的模型：**终止**（termination）和**恢复**（resumption）。在终止模型中（这是C++语言所支持的），会假设错误相当严重以至于无法从异常发生的位置自动恢复执行。换言之，抛出异常的人认为这种情况无法挽回，他们不想返回到抛出异常的位置。

另一种错误处理模型称为恢复模型，最早是在20世纪60年代的PL/I语言中引入的。[[8]](#footnote-8)若使用“恢复”语义，则意味着异常处理程序期望做一些事情来纠正当前的情况，然后自动重试导致异常的代码，并假设这一次能够成功。如果想在C++语言中使用恢复模型，那么必须显式将执行转移回发生错误的代码位置，这通常是通过重复最初的函数调用来实现的。一种常见的做法是将try块放到一个while循环中，然后不断重新进入try块，直到结果令人满意。

历史上确实有一些操作系统支持恢复模型，但写这些操作系统的程序员最终往往还是采用了类似于终止模型的代码，并完全跳过了恢复步骤。恢复模型表面上似乎颇有吸引力，但实际上并没有那么好用。其中一个原因是异常的发生位置和处理异常的位置相距甚远。很容易就可能在一个遥远的处理程序那里终止。但如果从那里跳转回去，那么对于大型系统来说，在概念上显得过于复杂。在这种系统中，有许多位置都可能产生异常。

## 1.4 异常匹配

当抛出异常时，异常处理系统会按照源代码中出现的顺序查找距离最近的处理程序。一旦找到匹配的处理程序，就认为异常已得到处理，不再继续查找下去。

异常不一定非要和处理程序完全匹配。如果抛出一个派生类的对象或引用，那么它可以被用于处理其基类的处理程序捕获到。不过，如果设置处理程序来捕获一个对象而不是引用，那么异常对象在传递给处理程序时会被“切片”（sliced）——截断为基类类型。这没有太大损害，但会造成所有派生类型信息的丢失（它的一些特有的成员就没了）。出于这个原因，也是为了避免再次拷贝异常对象，最好总是按引用而不是按值来捕获异常。[[9]](#footnote-9)如果抛出的是一个指针，那么会执行标准的指针转换以匹配异常。但是，在匹配过程中，不会使用自动类型转换将一种异常类型转换为另一种。下面展示了一个例子。[[10]](#footnote-10)

//: C01:Autoexcp.cpp

// 即使提供了类型转换构造函数，也不会自动执行类型转换，

#include <iostream>

using namespace std;

class Except1 {};

class Except2 {

public:

// 转换构造函数

Except2(const Except1&) {}

};

void f() { throw Except1(); }

int main() {

try {

f();

} catch(Except2&) {

cout << "当前在Except2处理程序中" << endl;

} catch(Except1&) {

cout << "当前在Except1处理程序中" << endl;

}

} ///:~

你可能会以为，第一个处理程序能通过转换构造函数将Except1对象转换为Except2来匹配。但是，系统在异常处理期间不会执行这样的转换。所以，抛出的异常最终在Except1的处理程序中处理的。

下面的例子展示了如何用基类处理程序来捕获派生类异常。

//: C01:Basexcpt.cpp

// 异常层次结构

#include <iostream>

using namespace std;

class X {

public:

class Trouble {};

class Small : public Trouble {};

class Big : public Trouble {};

void f() { throw Big(); }

};

int main() {

X x;

try {

x.f();

} catch(X::Trouble&) {

cout << "捕获Trouble" << endl;

// 被前面的处理器隐藏：

} catch(X::Small&) {

cout << "捕获Small Trouble" << endl;

} catch(X::Big&) {

cout << "捕获Big Trouble" << endl;

}

} ///:~

在本例中，抛出的无论是Trouble对象，还是它的派生类型的对象（通过公共继承[[11]](#footnote-11)），都会被第一个异常处理程序捕获到。这意味着第二个和第三个处理程序永远得不到调用，因为第一个处理程序已捕获了Trouble类层次结构中的所有异常。因此，更合理的做法是首先捕获派生类型，最后才捕获基类型。换言之，先捕获较具体的类型，再捕获较不具体的。

注意，之前展示的例子都是按引用来捕获异常的，但这对于简单的类来说不是很必要，因为派生类中并没有额外的成员，而且处理程序也没有指定参数标识符（例如，catch(X::Trouble& **trouble**)）。在处理程序中，我们通常使用引用参数而不是值参数来避免丢失信息。

### 1.4.1 捕获任意异常

有的时候，我们需要创建一个处理程序来捕获任意类型的异常。可以在参数列表中使用省略号来做到这一点，如下所示：

catch(...) {

cout << "抛出了一个异常" << endl;

}

由于使用省略号会捕获所有异常，所以应该放到处理程序列表的最后，避免“吞噬”所有较具体的异常，“架空”它后面的所有异常处理程序。

省略号没有给我我们提供添加实参的机会，所以无从了解异常或者它的类型。它是一个“通配符”。我们一般用这种catch子句来清理一些资源，然后重新抛出异常。

### 1.4.2 重新抛出异常

通常，为了释放一些资源，比如网络连接或者堆（heap）内存，我们希望重新抛出异常。（详情请参见本章后面的“资源管理”一节）。发生异常时，我们不一定关心是什么错误导致了异常。相反，只是想关闭之前打开的网络连接。然后，我们想让更靠近用户的某个上下文（即调用链更高层的位置）来处理这个异常。在这种情况下，省略号catch子句正是我们所需要的。我们想捕捉任何异常，清理资源，然后重新抛出异常，以便在其他地方处理。在处理程序中，使用一个无参的throw即可重新抛出异常。

catch(...) {

cout << "捕获了一个异常" << endl;

// 在这里释放资源，然后重新抛出异常

throw;

}

注意，无论在哪个catch子句中执行throw，当前try块内的任何后续catch子句都会被忽略——throw导致异常传递给更高层（外层）的下一个异常处理程序。此外，异常对象的所有信息都会得到保留。因此，可以在更高层的上下文中捕获特定的异常类型，并提取对象中可能包含的任何信息。

### 1.4.3 未捕获的异常

如本章开头所述，由于异常无法被忽略，并且错误处理逻辑与当前问题分离，所以异常处理被认为优于传统的“返回错误码”技术。如果当前try块后面没有任何异常处理程序能与一个异常匹配，那么该异常会上移到更高层次的上下文。这个上下文是一个函数或者try块，它包围着没有捕获到异常的那个try 块。注意，具体会在外层的哪个try块中捕获，这并非总是让人一目了然的，因为它位于调用链中较高的某一层。该过程持续进行（不断进行“栈展开”），直到某一层的处理程序匹配了该异常。此时，异常被视为“已捕获”，并且不再继续查找异常处理程序。

#### 1.4.3.1 terminate()函数

如果抛出的异常没有被任何一层的处理程序捕获到，那么最终会自动调用一个特殊的库函数 terminate()（在<exception>头文件中声明）。默认情况下，terminate()会调用标准C库函数abort()，这会导致程序的“突然”退出。在Unix系统中，abort()还会引起一次核心转储（core dump）。一旦调用了abort()，就不会调用正常的程序终止函数，这意味着所有全局和静态对象的析构函数都不会执行。在“栈展开”过程中，如果某个局部对象的析构函数抛出了异常，那么也会执行terminate()函数（这会中断进行中的异常）。当全局或静态对象的构造函数或析构函数抛出异常时，也会发生这种情况。注意，通常不应允许析构函数抛出异常。

#### 1.4.3.2 set\_terminate()函数

可以使用标准函数set\_terminate()来设置一个自定义的terminate()函数，该函数返回一个指针，它指向想要替代的那个terminate()函数（首次调用时，它指向默认的、由库提供的那个版本），以便将来可以恢复它。自定义的terminate()必须无参，并且返回void，但可以随意命名。此外，不能在自定义terminate()函数（本例命名为terminator()）中返回，也不能抛出异常，而是必须执行某种程序终止逻辑。一旦调用terminate()，不管是不是自定义的，问题都无解了[[12]](#footnote-12)，程序只能终止。区别在于，自定义的terminate()函数可以让我们更“得体”地终止程序。

下面的例子展示了set\_terminate()的使用。本例保存了返回值。所以，如有必要，可以使用terminate()函数来隔离发生“未捕获的异常”的那个代码段。

//: C01:Terminator.cpp

// 演示如何使用set\_terminate()，同时演示“未捕获的异常”。

#include <exception>

#include <iostream>

using namespace std;

// 这是一个自定义的terminate()

// 函数被命名为一部著名电影的名称：terminator（终结者）

// 并使用了其中的一句名言：I'll be back!

void terminator() {

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

exit(0);

}

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

class Botch {

public:

class Fruit {};

void f() {

cout << "Botch::f()" << endl;

throw Fruit();

}

~Botch() { throw 'c'; }

};

int main() {

try {

Botch b;

b.f();

} catch(...) {

cout << "当前在catch(...)处理程序中" << endl;

}

} ///:~

乍一看，old\_terminate的定义有些令人困惑：它不仅创建了一个函数指针，还把该指针初始化为set\_terminate()的返回值。尽管你可能习惯于在函数指针声明后面直接看到一个分号，在它在这里不过是一个变量而已，而所有变量都可以在定义时初始化。[[13]](#footnote-13)

Botch类不仅在f()中抛出了异常，还在其析构函数中抛出了异常。这会导致对terminate()的调用。执行main()时便可以体会到这一点。尽管异常处理程序被声明为catch(...)，似乎能捕获所有异常，不会导致对terminate()的调用，但terminate()还是被调用了。这是因为每当处理一个异常时，都会先清理栈上的对象，这自然会导致对Botch的析构函数的调用，而这引发了第二个异常，迫使调用terminate()。因此，千万不要在析构函数中抛出异常，也不要因为某种原因导致异常的抛出，这标志着拙劣的设计或糟糕的编码。[[14]](#footnote-14)

## 1.5 资源清理

异常处理机制一个神奇的地方在于，我们可以中断正常的程序流程，并跳转到适当的异常处理程序。然而，如果在抛出异常后没有正确地清理资源，这样做就没有意义了。C++的异常处理机制保证了每当离开一个作用域，该作用域内所有已完成构造的对象都会调用它们的析构函数。

下面的例子证明了未完成构造的对象不会调用其析构函数。除此之外，还演示了在创建对象数组的过程中抛出异常后会发生什么。

//: C01:Cleanup.cpp

// 抛出异常后，只有已构造完毕的对象才会被清理

#include <iostream>

using namespace std;

class Trace {

static int counter; // 全局int计数器

int objid;

public:

Trace() {

objid = counter++;

cout << "正在构造Trace对象#" << objid << endl;

if (objid == 3) throw 3;

}

~Trace() {

cout << "正在析构Trace对象#" << objid << endl;

}

};

int Trace::counter = 0;

int main() {

try {

Trace n1;

// 创建Trace对象数组的过程中抛出了异常：

Trace array[5];

Trace n2; // 永远执行不到这里

} catch (int i) {

cout << "捕获" << i << endl;

}

} ///:~

Trace类跟踪对象的创建过程，以方便我们跟踪程序的执行过程。它使用一个静态数据成员counter对创建的对象进行计数，并使用objid跟踪特定对象的编号。

主程序首先创建一个对象n1（编号#0），然后尝试创建包含5个Trace对象的一个数组。但是，第四个对象（编号#3）尚未完成构造，便抛出了一个异常。至于后面的n2对象，则永远不会创建。以下是程序的输出结果：

正在构造Trace对象#0

正在构造Trace对象#1

正在构造Trace对象#2

正在构造Trace对象#3

正在析构Trace对象#2

正在析构Trace对象#1

正在析构Trace对象#0

捕获3

注意，程序成功创建了三个数组元素，但在构造第四个元素期间抛出异常。由于main()中的第四次对象（即array[2]对象）构造操作“半途而废”了，所以只有array[1]和array[0]对象的析构函数会被调用。最后，对象n1会被销毁，而由于对象n2从未创建过，所以不需要销毁。

### 1.5.1 资源管理

在编写可能引发异常的代码时，尤为关键的一点是始终自问：“如果发生异常，资源能否得到正确的清理？”大多数时候，我们的代码都相对安全，但在构造函数中存在一个特殊的问题：如果在构造函数完成之前抛出异常，那么不会调用与该对象相关的析构函数。因此，在编写构造函数时必须格外小心。

这里的难点在于如何在构造函数中安全地分配资源。如果构造函数中发生了异常，那么就没有机会调用析构函数来释放这些资源。这个问题最常见于使用“裸”指针的情形中。[[15]](#footnote-15)

//: C01:Rawp.cpp

// 演示裸指针

#include <iostream>

#include <cstddef>

using namespace std;

// 猫类

class Cat {

public:

Cat() { cout << "Cat()" << endl; }

~Cat() { cout << "~Cat()" << endl; }

};

// 狗类

class Dog {

public:

void\* operator new(size\_t sz) {

cout << "分配一只狗" << endl;

throw 47;

}

void operator delete(void\* p) {

cout << "取消分配（释放）一只狗" << endl;

::operator delete(p);

}

};

// 这个类使用了资源

class UseResources {

public:

Cat\* bp;

Dog\* op;

UseResources(int count = 1) {

cout << "UseResources()" << endl;

bp = new Cat[count];

op = new Dog;

}

~UseResources() {

cout << "~UseResources()" << endl;

delete [] bp; // 释放猫对象数组

delete op; // 释放狗对象

}

};

int main() {

try {

UseResources ur(3); // 试图创建三只猫和一只狗

} catch(int) {

cout << "已进入异常处理程序" << endl;

}

}

上述代码的输出如下：

UseResources()

Cat()

Cat()

Cat()

分配一只狗

已进入异常处理程序

程序正常进入了UseResources 类的构造函数，Cat类的构造函数正常创建了三个数组对象。但是，在 Dog::operator new()构造函数内部，我们人为地抛出了一个异常（以模拟内存不足错误）。随后，就突兀地进入了异常处理程序，UseResources类的析构函数未获得调用。由于UseResources类的构造函数没有顺利完成，所以这是预料之内的结果。但是，这同时意味着在堆上成功创建的那些Cat对象永远不会被销毁。

### 1.5.2 让一切成为对象

为了避免出现像这样的资源泄漏，必须采取以下两种方式之一来防止这种“原始”的资源分配：

* 在构造函数内部捕获异常，以便及时释放资源。
* 在对象的构造函数中分配资源，并在对象的析构函数中释放资源。

若采用第二种方式，那么每个分配操作都会变得“原子化”，因为它成了本地对象生命周期的一部分。[[16]](#footnote-16)如果分配失败，那么其他分配了资源的对象都会在“栈展开”期间被正确清理。这种技术称为**资源获取即初始化**（Resource Acquisition Is Initialization，RAII），因为它将资源的获取和释放绑定到对象的生命周期。我们可以使用模板来修改上一个示例来轻松实现RAII。

//: C01:Wrapped.cpp

// 安全的、原子化的指针

#include <iostream>

#include <cstddef>

using namespace std;

// 这里从简了，你可以考虑提供其他参数

template <class T, int sz = 1> class PWrap {

T\* ptr;

public:

class RangeError {}; // 异常类

PWrap() {

ptr = new T[sz];

cout << "PWrap类的构造函数" << endl;

}

~PWrap() {

delete[] ptr;

cout << "PWrap类的析构函数" << endl;

}

T& operator[](int i) throw(RangeError) {

if (i >= 0 && i < sz) return ptr[i];

throw RangeError();

}

};

class Cat {

public:

Cat() { cout << "Cat()" << endl; }

~Cat() { cout << "~Cat()" << endl; }

void g() {}

};

class Dog {

public:

void\* operator new[](size\_t) {

cout << "分配一只狗" << endl;

throw 47;

}

void operator delete[](void\* p) {

cout << "取消分配（释放）一只狗" << endl;

::operator delete[](p);

}

};

class UseResources {

PWrap<Cat, 3> cats;

PWrap<Dog> dog;

public:

UseResources() { cout << "UseResources()" << endl; }

~UseResources() { cout << "~UseResources()" << endl; }

void f() { cats[1].g(); }

};

int main() {

try {

UseResources ur;

} catch (int) {

cout << "已进入异常处理程序" << endl;

} catch (...) {

cout << "已进入catch(...)" << endl;

}

} ///:~

和上一个版本的区别在于，这个版本使用模板类来包装指针并将其转换为对象。这些对象的构造函数会在调用UseResources构造函数的主体**之前**得到调用。另外，在抛出异常之前已完成的任何构造函数都会在“栈展开”期间调用其关联的析构函数。

PWrap模板类展示了一种比迄今为止所见到的更为典型的异常使用方式。它创建了一个名为RangeError的嵌套异常类。该异常类设计在操作符函数operator[]中使用，当该函数的实参超出了范围（即越界）时，就会抛出该异常。由于operator[]返回的是一个引用，所以不可能返回零（C++语言不存在空引用）。[[17]](#footnote-17)这是一种真正的“异常”情况——我们不知道在当前上下文中应该如何处理，也无法返回一个不可能的值。在本例中，RangeError[[18]](#footnote-18)非常简单，它假设仅凭其类名就反映所有必要的信息。但如果需要的话，完全可以添加一个成员来容纳索引值，使调用者更清楚地了解具体是哪个索引“越界”了。

上述代码的输出如下：

Cat()

Cat()

Cat()

PWrap类的构造函数

分配一只狗

~Cat()

~Cat()

~Cat()

PWrap类的析构函数

已进入异常处理程序

程序为Dog对象分配存储空间时再一次抛出了异常，但这一次Cat对象数组得到了正确的清理，没有造成内存泄漏。

### 1.5.3 auto\_ptr

由于动态内存是典型 C++程序所使用的最常见的资源，所以标准库提供了一个RAII（资源获取即初始化）包装器来管理指向堆内存的指针，以支持内存的自动释放。auto\_ptr类模板在<memory>头文件中定义，它的一个构造函数接收指向其泛型类型的指针（可以在自己的代码中使用任意类型）。auto\_ptr类模板还重载了指针操作符\*和->，以便将这些操作转发给auto\_ptr对象包装的原始指针。这样一来，就可以像使用裸指针那样使用auto\_ptr对象。[[19]](#footnote-19)下面的例子展示了它是如何工作的。[[20]](#footnote-20)

//: C01:Auto\_ptr.cpp

// 演示auto\_ptr的RAII特性

#include <memory>

#include <iostream>

#include <cstddef>

using namespace std;

class TraceHeap {

int i;

public:

static void\* operator new(size\_t siz) {

void\* p = ::operator new(siz);

cout << "在堆的地址" << p << "处分配了一个TraceHeap对象" << endl;

return p;

}

static void operator delete(void\* p) {

cout << "释放了地址" << p << "处的TraceHeap对象" << endl;

::operator delete(p);

}

TraceHeap(int i) : i(i) {}

int getVal() const { return i; }

};

int main() {

auto\_ptr<TraceHeap> pMyObject(new TraceHeap(5));

cout << pMyObject->getVal() << endl; // 输出5

} ///:~

TraceHeap类重载了operator new和operator delete，以便明确显示这些操作符函数内部执行了什么操作。注意，和其他任何类模板一样，必须在模板参数中指定想要使用的类型。不过，这里不要写成TraceHeap\*，因为auto\_ptr知道它存储的是指向你的类型的指针。main()中的第二行验证了auto\_ptr的operator->()函数会间接应用于原始的底层指针。最重要的是，即使这里没有显式地释放（即delete）原始指针，pMyObject的析构函数也会在栈展开期间释放原始指针，如以下输出所示：

在堆的地址012F6C30处分配了一个TraceHeap对象

5

释放了地址012F6C30处的TraceHeap对象

对于指针数据成员来说，auto\_ptr类模板非常“好用”。由于按值包含的类对象总是会被析构[[21]](#footnote-21)，因此它的auto\_ptr成员总是会在当前所在的对象被析构时释放自己包装的原始指针。[[22]](#footnote-22)

### 1.5.4 函数级try块

由于构造函数经常会抛出异常，所以可以考虑在对象的成员或基类子对象[[23]](#footnote-23)初始化期间处理异常。为此，可以将子对象的初始化操作放在一个函数级的try块中。与常规语法有所不同，构造函数初始化器的try块现在变成了构造函数的主体，而关联的catch块位于构造函数的主体之后，如下面的例子所示：

//: C01:InitExcept.cpp {-bor}

// 处理来自子对象的异常

#include <iostream>

using namespace std;

class Base {

int i;

public:

class BaseExcept {};

Base(int i) : i(i) { throw BaseExcept(); }

};

class Derived : public Base {

public:

class DerivedExcept {

const char\* msg;

public:

DerivedExcept(const char\* msg) : msg(msg) {}

const char\* what() const { return msg; }

};

Derived(int j) try : Base(j) {

// 构造函数的主体

cout << "这部分不会打印" << endl;

} catch (BaseExcept&) {

throw DerivedExcept("Base子对象抛出了异常");

}

};

int main() {

try {

Derived d(3);

} catch (Derived::DerivedExcept& d) {

cout << d.what() << endl; // 输出："Base子对象抛出了异常"

}

} ///:~

注意，在 Derived 构造函数的初始化列表位于 try 关键字之后、构造函数主体之前。如果真的发生了异常，那么包含的对象不会被构造。因此，回到创建它的代码是没有意义的。因此，唯一合理的做法是在函数级的catch子句中抛出异常。

尽管不是很实用，但C++语言也允许**任何**函数使用函数级try块，如下面的例子所示：

//: C01:FunctionTryBlock.cpp {-bor}

// 函数级的try块

// {RunByHand}（不要通过makefile自动运行）

#include <iostream>

using namespace std;

int main() try {

throw "main";

} catch (const char\* msg) {

cout << msg << endl; // 输出："main"

return 1;

} ///:~

在本例中，catch块的返回方式与函数主体的正常返回方式无异。使用这种类型的函数级try块与正常使用try-catch来包围函数主体没有太大的区别。

## 1.6 标准异常

我们可以直接使用由标准C++库定义的异常类。通常情况下，从一个标准异常类开始，要比尝试自己定义一个更容易、更快。如果标准类不能完全满足需求，则完全可以从它派生出自己的异常类。

所有标准异常类最终都继承自头文件<exception>中定义的exception类。两个主要的派生类是logic\_error和runtime\_error，它们可以在<stdexcept>中找到（<stdexcept>本身包含了 <exception>）。其中，logic\_error （逻辑错误）类代表编程逻辑中存在的错误，比如传递了无效的参数。runtime\_error（运行时错误）则是由诸如硬件故障或内存耗尽等不可预见的因素引起的。runtime\_error和logic\_error都提供了一个接受std::string参数的构造函数。这样就可以将消息存储在异常对象中，并在以后通过exception::what()提取。下面展示了一个例子。

//: C01:StdExcept.cpp

// 从std::runtime\_error派生出一个异常类

#include <stdexcept>

#include <iostream>

using namespace std;

class MyError : public runtime\_error {

public:

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

};

int main() {

try {

throw MyError("我的消息");

} catch(MyError& x) {

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

}

} ///:~

虽然runtime\_error提供了一个构造函数，允许将消息插入其std::exception子对象，但std::exception并没有提供这样一个接受std::string参数的构造函数。因此，一般都要从runtime\_error或logic\_error（或它们的派生类）派生出自己的异常类，而不是从std::exception派生。[[24]](#footnote-24)

表1.1总结了主要的异常类。

表1.1 主要的异常类

|  |  |
| --- | --- |
| **类名** | **描述** |
| exception | 由C++标准库抛出的所有异常的基类。可以调用what()来检索初始化异常对象时提供的可选字符串 |
| logic\_error | 继承自exception。报告程序逻辑错误，这些错误理论上可以通过检查代码检测出来 |
| runtime\_error | 继承自exception。报告运行时错误，这些错误理论上只有在程序执行时才能检测到 |

在头文件<iostream>中的定义的异常类ios::failure也继承自exception，但它没有进一步的子类[[25]](#footnote-25)。

可以直接使用表1.2和表1.3中列出的类，也可以将它们作为基类来派生出新的异常类。

表1.2 派生自logic\_error的异常类

|  |  |
| --- | --- |
| **类名** | **描述** |
| domain\_error | 有的函数具有“前置条件”，例如sqrt()的参数不能为负。违反这种条件就会抛出该异常 |
| invalid\_argument | 表明抛出该异常的函数接收到了无效的参数 |
| length\_error | 试图生成一个长度大于或等于npos的对象。npos是当前上下文中的大小类型（通常为std::size\_t）允许表达的最大值[[26]](#footnote-26) |
| out\_of\_range | 参数超出范围 |
| bad\_cast | 在运行时类型识别（参见第 8 章）中试图执行一个无效的dynamic\_cast表达式 |
| bad\_typeid | 在表达式typeid(\*p)中使用了空指针 p（这也是第 8 章讲解的一个运行时类型识别特性） |

表1.3 派生自runtime\_error的异常类类

|  |  |
| --- | --- |
| **类名** | **描述** |
| range\_error | 运行时违反了“后置条件” |
| overflow\_error | 运行时算术溢出 |
| bad\_alloc | 运行时分配存储空间失败 |

## 1.7 异常规范

虽然函数不一定非要明确说明可能抛出的异常，但这样做被认为是一种良好的编程习惯。如果不这样做，用户就无法提前预知可能发生的异常，从而无法编写相应的异常处理代码。即使用户拥有源代码，逐行查找throw语句也并不高效。而且，以库的形式提供的函数通常不包含源代码。良好的文档可以缓解这一问题，但现实情况是，并非所有软件项目都提供了完善的文档。C++语言提供了**异常规范**这一语法机制，允许函数明确声明可能抛出的异常类型，以便调用者有针对性地处理这些异常。异常规范是可选的，位于函数声明的参数列表之后。

异常规范要求先写throw关键字，后跟一对圆括号，并在括号内列出函数可能抛出的所有异常类型。下面这个函数声明使用了异常规范： [[27]](#footnote-27)

void f() throw(toobig, toosmall, divzero);

注意，以下传统的函数声明：

void f();

意味着函数可能抛出任何类型的异常。但如果像下面这样声明：

void f() throw();

则表示该函数不会抛出任何异常。[[28]](#footnote-28)因此，最好确保调用链中的任何函数都不会造成任何异常的向上传播！[[29]](#footnote-29)

为了遵循良好的编码规范、提供良好的文档以及为调用者提供方便，在编写可能抛出异常的函数时，请务必考虑使用异常规范。（本章稍后会讨论该指导原则的各种变体。）

unexpected()函数

如果异常规范声称可能抛出一组特定的异常，但却抛出了不在该规范中的异常，那么会有什么后果呢？若抛出的异常未在异常规范中列出，那么会调用特殊的unexpected()函数。在这种情况下，unexpected()会默认调用本章前面描述的terminate()函数。

**set\_unexpected()函数**

与terminate()相似，unexpected()机制也允许指定自己的函数来响应未预期的异常。为此，我们需要使用set\_unexpected()函数。与set\_terminate()相似，它接受一个无参且返回void的函数的地址。另外，由于它返回了unexpected()指针先前的值，所以可以保存该值并在以后恢复。注意，为了使用set\_unexpected()，我们需要包含头文件<exception>。下面展示了一个简单的例子。[[30]](#footnote-30)

//: C01:Unexpected.cpp

// 演示异常规范与unexpected()

//{-msc} (无法正常终止)

#include <exception>

#include <iostream>

using namespace std;

class Up {};

class Fit {};

void g();

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

switch(i) {

case 1: throw Up();

case 2: throw Fit();

}

g();

}

// void g() {} // 版本 1

void g() { throw 47; } // 版本 2

void my\_unexpected() {

cout << "抛出未预期的异常" << endl;

exit(0);

}

int main() {

set\_unexpected(my\_unexpected); // (忽略返回值)

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

try {

f(i);

} catch(Up) {

cout << "Up被捕获" << endl;

} catch(Fit) {

cout << "Fit被捕获" << endl;

}

}

} ///:~

Up类和Fit类纯粹作为异常抛出。异常类通常都很小，但当然也可以在其中包含额外的信息，以便处理程序查询。

函数f()的异常规范承诺只抛出Up类和Fit类的异常。从函数定义来看，这似乎是合理的。g()的第一个版本（被f()调用）不抛出任何异常，所以确实合理。但是，如果有人修改g()使其抛出不同类型的异常（例如本例中的第二个版本，它抛出了一个int），那么f()的异常规范就被违反了。

my\_unexpected()函数既没有参数也没有返回值，它遵循了自定义unexpected()函数的正确形式。它只是显示一条消息，以证明它确实被调用了，然后就退出程序（这里使用exit(0)是为了让本书示例代码的make过程不被中断）。在你自己的unexpected()函数中，不应该有return语句。

在main()中，try块位于一个for循环内，其目的是测试所有可能性。采取这种方式，我们可以模拟一下“努力恢复程序的正常运行”。也就是说，将try块嵌套在一个for、while、do或if语句中，导致在发生任何异常时都尝试修复问题；然后继续try。

只有Up和Fit异常才会被捕获，因为f()的程序员在规范中承诺只抛出这些异常。g()的第二个版本导致my\_unexpected()被调用，因为f()抛出了一个int，然而，这是不符合“规范”的。

本例在调用set\_unexpected()时忽略了返回值，但完全可以把返回值保存在某个函数指针中，并在以后恢复。这具体可以参考本章早些时候的set\_terminate()示例。

一个典型的unexpected处理程序会记录错误，并调用exit()来终止程序。然而，它也可以抛出另一个异常（或重新抛出相同的异常），甚至可以调用abort()[[31]](#footnote-31)。如果在unexpected处理程序中抛出的异常类型是最初违反规范的函数所允许的，那么将从该函数的调用位置开始，继续搜索异常处理程序（这一行为是unexpected()独有的）。

但如果从unexpected处理程序抛出的异常不符合原始函数的规范，则会发生下面两种情况之一。

1. 如果std::bad\_exception（在<exception>中定义）在函数的异常规范中，那么从unexpected处理程序抛出的异常将被替换为一个std::bad\_exception对象，并且从该函数处继续搜索异常处理程序，如同往常一样。
2. 如果原始函数的规范不包括std::bad\_exception，则调用terminate()。

以下程序演示了这种行为：[[32]](#footnote-32)

//: C01:BadException.cpp {-bor}

#include <exception> // 为了使用std::bad\_exception

#include <iostream>

#include <cstdio>

using namespace std;

// 异常类:

class A {};

class B {};

// terminate()处理程序

void my\_thandler() {

cout << "terminate被调用" << endl;

exit(0);

}

// unexpected()处理程序

void my\_uhandler1() { throw A(); }

void my\_uhandler2() { throw; }

// 如果将这个throw语句嵌入到f或g中，

// 那么编译器会检测到违反异常规范，并报告错误。

// 因此，我们把它放在自己的函数中。

void t() { throw B(); }

// 在C++11中弃用，从C++17起禁止

void f() throw(A) { t(); }

void g() throw(A, bad\_exception) { t(); }

int main() {

set\_terminate(my\_thandler);

set\_unexpected(my\_uhandler1);

try {

f();

} catch(A&) {

cout << "从f捕获了一个A" << endl;

}

set\_unexpected(my\_uhandler2);

try {

g();

} catch(bad\_exception&) {

cout << "从g捕获了一个bad\_exception" << endl;

}

try {

f();

} catch(...) {

cout << "这段话永远不会显示" << endl;

}

} ///:~

my\_uhandler1()处理程序抛出一个可接受的异常（A），因此从第一个catch处恢复执行，该catch成功捕获异常。虽然my\_uhandler2()处理程序抛出的异常（B）无效，但由于g规定了bad\_exception，所以B异常被替换为bad\_exception对象，因此第二个catch也成功捕获异常。由于f的规范中不包括bad\_exception，因此调用自定义的terminate处理程序my\_thandler()。

以下是输出结果：

从f捕获了一个A

从g捕获了一个bad\_exception

terminate被调用

### 1.7.1 更好的异常规范

显然，当前（C++98）的异常规范不够安全。事实上，void f();**应该**意味着这个函数不会抛出任何异常。如果希望程序员能抛出任何类型的异常，那么似乎*应该*像下面这样声明：

void f() throw(...); // C++不支持这样写

这无疑是一种进步，因为函数声明变得更明确了。遗憾的是，不能总是通过查看函数的代码来判断是否会抛出异常——例如，可能因为运行时的内存分配而引发异常。更糟的是，有些函数是在引入异常处理机制之前写的，它们可能因为调用的函数（后者可能链接到新的、会抛出异常的版本）而无意中抛出异常。因此，有必要让void f();意味着“也许我会抛出异常，也许不会”这种不明确的情况，以避免阻碍代码的演进。[[33]](#footnote-33)如果想指定f不抛出任何异常，请使用空列表，如下所示：[[34]](#footnote-34)

void f() throw();

### 1.7.2 异常规范与继承

类中的每个公共函数都相当于与用户建立了一个契约：传递某些参数（实参）[[35]](#footnote-35)，它会执行某些操作并且/或者返回一个结果。在派生类中，同样的契约也必须成立；否则，派生类与基类之间预期的“属于”（is-a）关系[[36]](#footnote-36)将被破坏。由于异常规范逻辑上是函数声明的一部分，所以它们在整个继承层次结构中也必须保持一致。例如，如果基类中的成员函数声明它只会抛出类型为A的异常，那么在派生类中重写（override）该函数时，就不得向规范列表中添加其他任何异常类型，因为这会破坏遵循基类接口的任何程序。不过，可以在重写版本中指定更少的异常，或根本不指定任何异常，而这不需要用户做任何不同的事情[[37]](#footnote-37)。除此之外，还可以在派生函数的异常规范中使用任何属于（继承自）A的类型来代替A，如下面的例子所示： [[38]](#footnote-38)

//: C01:Covariance.cpp {-xo}

// 应当导致编译错误{-mwcc}{-msc}

#include <iostream>

using namespace std;

class Base {

public:

class BaseException {};

class DerivedException : public BaseException {};

virtual void f() throw(DerivedException) {

throw DerivedException();

}

virtual void g() throw(BaseException) {

throw BaseException();

}

};

class Derived : public Base {

public:

// 保持异常规范与基类一致

void f() throw(DerivedException) {

throw BaseException();

}

virtual void g() throw(DerivedException) {

throw DerivedException();

}

}; ///:~

编译器应报告Derived::f()这个重写版本存在错误（或至少显示一个警告），因为它以一种违反Base::f()规范的方式更改了异常规范。Derived::g()的异常规范则没有问题，因为DerivedException是BaseException的子类；换言之，前者“is-a”后者（反之则不然）。可以将Base/Derived和BaseException/DerivedException视为平行的类层次结构。当前若处在Derived中，就可以在异常规范和返回值中使用DerivedException来替换对BaseException的引用。这种行为称为**协变**（covariance）[[39]](#footnote-39)，因为这两组类分别沿它们各自的层次结构一起向下变化。本书第1卷曾提醒过：参数类型不是**协变量**（covariant）；也就是说，不允许更改重写的虚函数的签名。[[40]](#footnote-40)

### 1.7.3 何时不使用异常规范

浏览标准C++库中的函数声明，我们会发现居然没有任何一个地方在用异常规范！虽然这看起来可能有些奇怪，但却有一个很好的理由：库主要由模板构成，而我们永远不知道一个泛型类型或函数会做些什么。例如，假定当前要开发一个泛型栈模板，并尝试为自己的pop函数附加一个异常规范，如下所示：

T pop() throw(logic\_error);

由于预见到的唯一错误是栈下溢（underflow）[[41]](#footnote-41)，所以可能认为指定logic\_error或其他适当的异常类型就“安全”了。但是，类型T的拷贝构造函数也可能抛出异常。这样就会调用unexpected()，而程序将会终止。如果没有把握，就不要随便保证！如果不知道可能会发生什么异常，就不要使用异常规范。这就是为什么作为标准C++库主体的模板类没有使用异常规范的原因。[[42]](#footnote-42)它们在**文档**中说明了自己已知的异常，并将剩下的留给模板的“用户”来处理。总之，异常规范主要用于非模板类。

## 1.8 异常安全性

第7章将深入探讨标准C++库中的容器，其中包括栈（stack）。届时，你会注意到其pop成员函数的声明如下：

void pop();

你可能觉得奇怪，pop()不返回一个值。相反，它直接移除栈顶元素。要获取栈顶的值，需要在调用pop()之前调用top()。这种行为背后有一个重要的原因，即**异常安全性**（exception safety），它是库设计中的一项重要考虑因素。存在多种不同的异常安全级别，但重点在于，顾名思义，为了确保异常安全性，我们需要在处理异常时使用正确的语义。

假设要使用动态数组来实现一个栈（我们将这个数组称为data，并使用一个辅助的整型计数器变量count），并且尝试编写返回一个值的pop()，如下所示：

template<class T>

T stack<T>::pop() {

if (count == 0)

throw logic\_error("栈下溢");

else

return data[--count];

}

最后一行在返回值时会调用拷贝构造函数，而如果此时抛出异常会发生什么？由于异常，出栈的元素不会返回，但count已提前递减。换言之，栈顶元素永远丢失了！问题在于，该函数试图同时做两件事：（1）返回一个值，（2）改变栈的状态[[43]](#footnote-43)。最好将这两个动作分离为两个独立的成员函数，而这正是标准stack类所做的。换言之，我们应遵循“内聚”设计实践，即每个函数只做好一件事。[[44]](#footnote-44)异常安全的代码会使对象保持一致的状态，并且不会造成资源的泄露。

编写自定义赋值操作符时也需要小心。在本书第1卷的第12章中，我们提到operator=应遵循以下模式。

1. 确保不是自己给自己赋值。如果是，那么跳到步骤6（这一步纯粹是为了优化）。
2. 为指针数据成员分配新内存。
3. 将数据从旧内存拷贝到新内存。
4. 释放（delete）旧内存。
5. 将新的堆指针赋给指针数据成员，以更新对象的状态。
6. 返回\*this。

这里的重点在于，在为所有新内容安全分配内存，并完成初始化之前，不要改变对象的状态。一个不错的技巧是将步骤2和步骤3转移到一个单独的函数中，它通常命名为clone（克隆）。下面的例子演示了如何为具有两个指针成员（theString和theInts）的类做这件事情。[[45]](#footnote-45)

//: C01:SafeAssign.cpp

// 一个异常安全的 operator=

#include <iostream>

#include <new> // 为了使用std::bad\_alloc

#include <cstring>

#include <cstddef>

using namespace std;

// 该类包含两个指针成员

class HasPointers {

// 一个负责容纳数据的句柄类

struct MyData {

const char\* theString;

const int\* theInts;

size\_t numInts;

MyData(const char\* pString, const int\* pInts, size\_t nInts)

: theString(pString), theInts(pInts), numInts(nInts) {}

} \*theData; // 句柄

// 克隆和资源清理函数：

static MyData\* clone(const char\* otherString, const int\* otherInts, size\_t nInts) {

char\* newChars = new char[strlen(otherString) + 1];

int\* newInts;

try {

newInts = new int[nInts];

} catch (bad\_alloc&) {

delete[] newChars;

throw;

}

try {

// 本例使用语言的内置类型，因此不会抛出异常。

// 但是，若使用类类型，则可能抛出异常。

// 所以，这里用try块来演示。（这也是本例的重点！）

strcpy(newChars, otherString);

for (size\_t i = 0; i < nInts; ++i)

newInts[i] = otherInts[i];

} catch (...) {

delete[] newInts;

delete[] newChars;

throw;

}

return new MyData(newChars, newInts, nInts);

}

// 第二个克隆函数

static MyData\* clone(const MyData\* otherData) {

return clone(otherData->theString, otherData->theInts,

otherData->numInts);

}

static void cleanup(const MyData\* theData) {

delete[] theData->theString;

delete[] theData->theInts;

delete theData;

}

public:

HasPointers(const char\* someString, const int\* someInts, size\_t numInts) {

theData = clone(someString, someInts, numInts);

}

HasPointers(const HasPointers& source) {

theData = clone(source.theData);

}

HasPointers& operator=(const HasPointers& rhs) {

if (this != &rhs) {

MyData\* newData = clone(rhs.theData->theString,

rhs.theData->theInts, rhs.theData->numInts);

cleanup(theData);

theData = newData;

}

return \*this;

}

~HasPointers() { cleanup(theData); }

friend ostream& operator<<(ostream& os, const HasPointers& obj) {

os << obj.theData->theString << ": ";

for (size\_t i = 0; i < obj.theData->numInts; ++i)

os << obj.theData->theInts[i] << ' ';

return os;

}

};

int main() {

int someNums[] = { 1, 2, 3, 4 };

size\_t someCount = sizeof someNums / sizeof someNums[0];

int someMoreNums[] = { 5, 6, 7 };

size\_t someMoreCount = sizeof someMoreNums / sizeof someMoreNums[0];

HasPointers h1("你好", someNums, someCount);

HasPointers h2("再见", someMoreNums, someMoreCount);

cout << h1 << endl; // 你好: 1 2 3 4

h1 = h2;

cout << h1 << endl; // 再见: 5 6 7

} ///:~

为了方便起见，HasPointers使用MyData类作为两个指针的句柄。每当需要分配更多内存时，无论是构造还是赋值，最终都会调用第一个clone函数来完成任务。如果第一次调用new操作符失败，那么会自动抛出bad\_alloc异常。如果在针对theInts的第二次内存分配时抛出异常，那么必须清理已为theString分配的内存——因此，我们在第一个try块中捕获bad\_alloc异常。第二个try块对本例来说并不是必需的，因为它只是拷贝int和指针（因此不会发生异常）。但若要拷贝对象，那么它们的赋值操作符就可能引发异常，此时就需要清理已分配的所有内存。注意，这两个异常处理程序都重新抛出了异常，因为这里只是执行资源的管理工作；用户仍然需要知道有错误发生。所以，我们让异常沿着动态链向上传播。不默默“吞噬”异常的软件库被称为是异常中立（exception neutral）的。我们应竭尽全力编写既异常安全又异常中立的库。[[46]](#footnote-46)

仔细检查前面的代码，会发现事实上没有任何delete操作会抛出异常。上述代码要想正常运行，就必须依赖于这一事实。我们知道，对一个对象调用delete时，会调用该对象的析构函数。实际上，若非假设析构函数不会抛出异常，那么就几乎不可能设计出“异常安全”的代码。因此，千万不要在析构函数中抛出异常！（在本章结束前，我们会再次强调这一点。）[[47]](#footnote-47)

## 1.9 使用异常进行编程

大多数程序员（尤其是C程序员）原先使用的语言并不存在“异常”这一概念，因此需要一段时间来适应。接下来的小节将介绍一些使用异常进行编程的指导原则。

### 1.9.1 什么时候应避免使用异常

异常不是万金油，所以过度使用可能会带来麻烦。下面几个小节描述了不应使用异常的一些情况。为了判断何时应该使用异常，最佳的建议是“只有当一个函数未能满足其规范时才抛出异常”。

不适合异步事件

标准C语言中的signal()系统和其他任何类似的系统处理的都是异步事件：即发生在程序流程外部的事件，是程序本身无法预见的。由于异常及其处理程序位于同一个调用栈上，因此不能使用C++异常来处理异步事件。换言之，异常依赖于程序的运行时栈上的动态函数调用链（它们具有“动态作用域”），[[48]](#footnote-48)而异步事件必须由完全独立于正常程序流程的代码来处理（通常是中断服务例程或事件循环）。不要在中断处理程序中抛出异常。

这并不是说异步事件不能与异常**关联**。但是，中断处理程序应尽可能快速完成工作并返回。为了处理这种情况，通常的做法是在中断处理程序中设置一个标志（flag），并在主线程[[49]](#footnote-49)代码中同步[[50]](#footnote-50)检查它。

不适合良性错误情况

一个错误如果有足够的信息来处理，就不要把它当作异常。应该在当前上下文中处理它，而不是把它当作异常抛给更大的上下文。

另外，C++异常并不适合处理诸如“除以零”错误这样的机器级事件。[[51]](#footnote-51)我们一般都要假定有其他机制（例如，操作系统或硬件）在处理这种事件。这样一来，C++异常可以保持相对高效，并且将它们的应用限制在程序级的异常情况。

不适合控制流

异常看起来有点像一种备选的返回机制，又有点像一个switch语句，因此你可能会觉得能用异常来替代这些常规的语言机制。但这是一个糟糕的想法，部分原因是异常处理机制比正常的程序执行要低效得多。记住，异常是罕见事件，所以正常的程序不应该为此付出代价。此外，除非异常反映的是真正的错误情况，否则会让使用你的类或函数的用户感到困惑。

不要强迫使用异常

有些程序非常简单（例如，一些小型工具程序）。它们可能只需要接收输入并执行一些处理。在这些程序中，可能会遇到分配内存失败、打开文件失败等“错误情况”。在这些情况下，显示一条消息并退出程序，让系统去收拾残局（清理资源），是完全可以接受的做法。没有必要机械地捕捉所有异常并恢复所有资源。简而言之，如果不需要异常，就没必要强迫自己使用它们。

当旧代码遇到新异常

另一种情况是修改一个原本没有使用异常的旧程序。在修改时，你可能会引入一个使用了异常的新库，并且想知道是否需要修改程序中的所有代码。假设你已经有了一个可接受的错误处理方案，那么最直接的做法是在使用新库的最大代码块周围（这可能是main()中的所有代码）加上一个 try 块，后面跟着一个catch(...)和基本的错误消息。当然，可以通过增加更具体的异常处理程序来进一步完善，但无论如何，需要新增的代码量都可以是最少的。更好的做法是将产生异常的代码隔离在一个try块中，并编写处理程序将异常转换为现有的错误处理方案。

为其他人创建库时，必须认真考虑异常处理，特别是当你不知道他们最终会如何响应关键错误情况时（可以回顾一下之前关于异常安全性和为什么标准C++库没有使用异常规范的讨论）。[[52]](#footnote-52)

### 1.9.2 异常的典型应用

我们应该使用异常来做以下事情。

* 修复问题并重试引发异常的函数。
* 做一些临时处理[[53]](#footnote-53)，继续执行而不重试函数。
* 在当前上下文中做自己能做的一切，并将**相同**的异常重新抛给更高的上下文。
* 在当前上下文中做自己能做的一切，并将**不同**的异常重新抛给更高的上下文。
* 终止程序。
* 将使用了传统错误处理方案的函数（特别是C库函数）包装成一个新函数，使之引发异常。
* 简化代码。传统的错误处理方案可能会使代码变得复杂，使用起来令人痛苦和烦恼。使用异常机制后，错误处理可以变得更加简单和有效。
* 让自己的库和程序变得更加安全。这既是一种短期投资（方便调试），也是一种长期投资（确保应用程序的健壮性）。

什么时候使用异常规范

异常规范类似于函数原型：它告诉使用者应编写异常处理代码来处理特定的异常。另外，它告诉编译器该函数可能会抛出哪些异常，以便它检测运行时的违规行为。[[54]](#footnote-54)

若只是检查代码，那么并不能总是预见到特定函数将会引发哪些异常。有的时候，它所调用的函数会产生意想不到的异常。有的时候，一个原本不抛出异常的旧函数被替换成了一个新的、会抛出异常的函数，并因此发生一个unexpected()调用。任何时候只要使用异常规范或者调用使用了异常规范的函数，就应考虑创建自己的unexpected()函数来记录消息。然后，要么抛出异常，要么终止程序。

如前所述，应该避免在模板类中使用异常规范，因为无法预知作为模板参数的类型可能会抛出哪种类型的异常。

从标准异常开始

在创建自己的异常之前，请先查阅标准C++库中的异常。如果标准异常就能满足需求，那么用户通常会更容易理解和处理这些异常。

如果需要定义一个新的异常类型，而该类型不在标准库中提供，那么最好从标准库中的现有异常类继承它。这样做的优点是，你的用户可以在他们的代码中使用what()函数来获取异常的描述信息，而无需关心你的异常类的具体实现。

嵌套自己的异常类

为自定义类创建异常时，最好将异常类嵌套在该类内部，或者包含在该类的命名空间内。这样可以明确地提醒读者：该异常只适用于该自定义类。此外，这还有助于防止对全局命名空间的污染。

即使是从C++标准异常派生的异常类也是可以嵌套的。

使用异常层次结构

利用异常层次结构，可以对类或库中可能遇到的关键错误类型进行有效的分类。这为用户提供了有用的信息，帮助他们组织代码，并使之有机会选择忽略所有具体类型的异常而仅仅捕获基类类型。此外，以后如果通过从同一个基类继承来添加新异常，那么不必重写现有的所有代码——基类处理程序会捕获这些新的异常。

标准C++异常是异常层次结构的一个很好的范例，应尽可能基于它来构建自己的异常。

多重继承

如第9章所述，只有在需要将对象指针向上转型（upcast）为两个不同的基类时（也就是说，需要这两个基类的多态行为），才需要使用多重继承（Multiple Inheritance，MI）。结果证明，异常层次结构是使用多重继承的好地方，因为可以在任何一个基类处理程序中处理多重继承而来的异常。

按引用捕获，而不是按值捕获

如1.4节“异常匹配”所述，应该按引用捕获异常，原因有二：

* 将异常对象传递给处理程序时，避免创建不必要的拷贝；
* 将派生异常作为基类对象捕获时，避免发生对象切片（截断为基类类型）。

虽然也可以抛出和捕获指针，但这样做会增加耦合性——抛出者和捕获者必须就异常对象如何分配和清理达成一致。这会成为一个问题，因为异常本身可能是由于堆耗尽而发生的。相反，如果抛出的是异常对象，那么异常处理系统会自动管理异常对象的生命周期，自动清理所有的存储。

在构造函数中抛出异常

由于构造函数没有返回值，所以在引入异常机制之前，我们在构造过程中只有两种方式报告错误：

* 设置一个非局部（nonlocal）标志，并希望使用者检查它；[[55]](#footnote-55)
* 返回一个未完全构造的对象，并希望使用者检查它。

这个问题非常严重，因为C程序员期望对象创建总是成功的。这一期望是合理的，因为C语言中的类型相对“原始”。然而，在C++程序中，如果构造失败后继续执行，肯定会导致灾难性的后果。因此，构造函数是抛出异常最重要的地方之一——我们现在有一个安全有效的方式来处理构造错误。但是，也必须注意对象内部的指针以及在构造函数内部抛出异常时的清理方式。

不要在析构函数中抛出异常

由于析构函数会在抛出其他异常的过程中被调用，所以永远都不要在析构函数中抛出异常，也不要因为在析构函数中执行的某个操作而导致另一个异常的抛出。如果发生这种情况，在到达现有异常的catch子句之前，可能会抛出一个新的异常，而这会导致调用terminate()。

如果在析构函数内部调用了某个可能抛出异常的函数，那么应该将这个调用放到析构函数内的一个try块中，而且析构函数必须自行处理所有异常。不允许任何异常从析构函数中“逃逸”。

避免裸指针

参见本章早些时候的Wrapped.cpp示例（1.5.2节）。如果在构造函数中为裸指针分配了资源，并在构造过程中抛出异常，那么这些资源就可能无法得到释放。指针是没有析构函数的，所以在构造函数中抛出异常时，这些资源不会被释放。若要创建指向堆内存的指针，请使用auto\_ptr或其他类型的智能指针。[[56]](#footnote-56)

## 1.10 开销

抛出异常会产生相当大的运行时开销（但这是一种**有益**的开销，因为对象会被自动清理）。因此，不应将异常作为正常控制流的一部分来使用，无论你觉得这样做有多么诱人或者多么“聪明”。异常应该很少发生，因此只有在发生异常时才应产生开销，而不能让正常的代码执行产生这种无谓的开销。异常处理的一个重要设计目标是：只要不发生异常，执行速度就不会受到影响。也就是说，只要不抛出异常，代码的运行速度应该与以前没有异常处理机制时一样快。不过，这具体要取决于所使用的特定的编译器实现。（参见本节稍后对“零成本模型”的描述。）

可以将throw表达式视为对一个特殊的系统函数的调用，该函数将异常对象作为参数，并回溯执行链。为此，编译器需要在栈上放置额外的信息，以帮助执行栈展开（stack unwinding）。为了理解这一点，需要先了解**运行时栈**（runtime stack）的相关知识。

每当一个函数被调用时，有关该函数的信息都会被压入运行时栈的一个**活动记录实例**（activation record instance，ARI）中，也称为**栈帧**（stack frame）。一个典型的栈帧包含以下信息：调用函数的地址（以便执行可以返回到那里）[[57]](#footnote-57)，指向函数的静态父函数的 ARI 的指针（父函数的作用域在词法上包含被调用函数，因此可以访问该作用域内的全局变量），以及指向调用它的函数（即它的动态父函数）的指针[[58]](#footnote-58)。从逻辑上重复跟随动态父链接所形成的一个路径即为本章提到的**动态链**或**调用链**。当抛出异常时，执行可以通过这个链来回溯，这也是使各个组件能够在运行时就错误信息进行沟通的机制。

为了在异常处理中实现“栈展开”，每个栈帧都需要存储与每个函数的异常相关的额外信息。这些信息描述了需要调用哪些析构函数（以便清理局部对象），指出了当前函数是否有try块，并列出了每个关联的catch子句可以处理的异常。这些额外的信息自然需要占用空间，因此支持异常处理的程序可能会比不支持的程序略大。[[59]](#footnote-59)程序如果启用了异常处理，那么编译后的大小也会更大一些，因为编译器必须生成如何在运行时生成扩展栈帧的逻辑。

为了说明这一点，我们分别启用和禁用对异常处理的支持，在Borland C++ Builder 和Microsoft Visual C++中尝试编译以下程序：[[60]](#footnote-60)

//: C01:HasDestructor.cpp {O}

class HasDestructor {

public:

~HasDestructor() {}

};

void g(); // 目前只知道可能抛出g

void f() {

HasDestructor h;

g();

} ///:~

在启用异常处理的情况下，编译器必须在f()的 ARI 中保持~HasDestructor()相关信息的可用性（目的是在g()抛出异常时可以正确销毁h对象）。表1.4总结了编译结果，以编译后的（.obj）文件的大小表示。[[61]](#footnote-61)

表1.4 不同编译器启用和禁用异常支持后的编译结果

|  |  |  |
| --- | --- | --- |
| **编译器\模式** | **启用异常支持** | **禁用异常支持** |
| Borland | 616 字节 | 234 字节 |
| Microsoft | 1162 字节 | 680 字节 |

不必太认真对待两种模式之间的百分比差异。记住，异常通常应该只是程序的一小部分，因此空间开销往往较小（通常在 5% 到 15% 之间）。

异常会带来一些额外的工作，因为会减慢执行速度，但聪明的编译器实现可以避免这种情况。由于关于异常处理代码和局部对象偏移信息可以在编译时一次性计算出来，因此这些信息可以保存在与每个函数关联的单一位置，而不必动态保存在每个ARI（栈帧）中。这实际相当于从每个ARI中移除了异常开销，避免花费额外的时间使它们入栈。这种方法被称为异常处理的**零成本**模型[[62]](#footnote-62)，而前面提到的优化存储空间被称为**影子栈**（shadow stack）。

## 1.11 小结

我们编写每一个程序都必须关注错误恢复，特别是在C++中创建供其他人使用的程序组件时。要想创建一个健壮的系统，每个组件都必须健壮。

C++异常处理的目标是使用比以前[[63]](#footnote-63)更少的代码来创建大型、可靠的程序，并且更好地确保应用程序不会产生任何未处理的错误。使用异常，程序几乎没有性能损失，对现有代码的影响也很小。

很容易就能掌握异常的基本应用。因此，请尽早开始在自己的程序中使用它们。异常是那些能立即为项目带来显著好处的语言特性之一。

## 1.12 练习

*本书配套资源提供了这些练习的答案，请从译者主页（bookzhou.com）或者扫码从清华大学出版社的网盘下载。*

1. 编写三个函数，其中一个返回错误值来指示发生了错误情况，一个设置errno，还有一个使用signal()。编写调用这些函数并响应错误的代码。然后，再编写第四个函数来抛出异常，调用这个函数并捕获异常。描述这四种方法之间的差异，并解释为什么异常处理是一种改进。

2. 创建一个在成员函数中抛出异常的类。在类中创建一个嵌套类作为异常对象。它接收一个代表描述字符串的const char\*参数。创建抛出该异常的一个成员函数（在函数的异常规范中声明[[64]](#footnote-64)）。编写调用了该函数的一个try块，并创建关联的catch块来处理该异常（直接显示描述字符串即可）。

3. 重写第1卷第13章的Stash类，使其在operator[]中抛出out\_of\_range（越界）异常。

4. 编写一个通用的main()函数，捕获所有异常并将其报告为错误。

5. 创建一个类来重载operator new。[[65]](#footnote-65)该操作符应分配10个对象，并且在第11个对象时“耗尽内存”并抛出异常。再添加一个用于回收这部分内存的静态成员函数。然后，在main()函数中添加try块和调用了内存回收例程的catch子句，将这些都放到一个while循环中，以演示从异常中恢复并继续执行的过程。

6. 创建一个会抛出异常的析构函数，然后编写代码来证明这不是个好的想法。证明若在到达现有异常的处理程序之前抛出新的异常，那么terminate()会被调用。

7. 证明所有抛出的异常对象都会被正确销毁。

8. 证明如果在堆上创建一个异常对象并抛出指向该对象的指针，那么它将不会被清理。

9. 编写一个函数，其异常规范持抛出四种异常类型：一个char、一个int、一个bool和你自己的异常类。在main()中捕获每一个并验证捕获。你的异常类应继承自标准异常。以一种系统能够恢复并再次尝试执行的方式来编写这个函数。

10. 修改上一个练习的答案，使函数抛出一个double（这违反了异常规范）。使用自己的unexpected()处理程序捕获该异常，显示一条消息并优雅地退出程序（即不调用abort()）。

11. 编写一个车库（Garage）类，其中一辆车（Car）的发动机（Motor）出了问题。在Garage类的构造函数中，使用函数级的try块在初始化Car对象时捕获从这辆车的Motor类中抛出的异常。在Garage构造函数的处理程序中抛出一个不同的异常，然后在main()中捕获这个新的异常。

# 第2章 防御性编程

编写“完美软件”或许是开发者可望不可及的一个目标，但习惯性地运用一些防御性编程技术，可以大幅提升代码的质量。

对于典型的生产软件，其复杂性确保了测试人员总是有工作可做。但是，我衷心希望开发者“不忘初心”：尽力开发无缺陷的软件！面向对象设计技术大幅降低了大型项目的难度，但开发者最终还是需要写循环和函数。这些“小规模编程”的细节会成为更大组件的基石。如果循环存在“相差一”（off by one）错误，或者函数只能在“大部分”时间内计算出正确的值，那么无论整体的方法论有多么先进，我们最终都会遭遇“可耻的失败”。本章将讨论一些“最佳实践”，无论项目的规模如何，它们都能帮助开发者创建健壮的代码。[[66]](#footnote-66)

我们写的代码表达了解决问题的一种尝试（这是代码最起码的作用）。作为开发者，应该清楚地理解在设计一个循环时的想法。在程序中的特定位置，应该能大胆地陈述某个条件成立。（如果不能，那么表明实际上还没有解决问题。）这样的陈述称为**不变量**（invariant）[[67]](#footnote-67)，因为在这个位置它必然为真；否则，要么是设计有问题，要么是代码没有准确反映出设计。

考虑一个“猜大小”（Hi-Lo）游戏程序。玩家在心里想一个1到100的数字，计算机猜测这个数字具体是哪个。玩家告诉计算机猜测的数字是太大、太小还是正确。作为猜测者（计算机），最佳策略是执行**二叉查找**。首先，选择目标数字所在范围的中点（因此，第一次猜，自然应该猜50）。根据玩家的回答，猜测者就知道目标数字应该在哪一半的列表中。然后，重复这个过程，每次迭代都将搜索范围减半。那么，如何写一个循环来正确地驱动这个重复过程呢？以下代码过于“简陋”：

bool guessed = false; // 是否猜中的标志

while(!guessed) { // 在没有猜中的情况下

...

}

这是因为恶意用户可能会说假话，让程序猜上一整天。然而，每次猜测时，程序做了什么简单的假设？换言之，**根据设计**，在每次循环迭代时，什么条件是必然成立的？

简单的假设是，在最开始的时候，这个秘密数字在当前活跃的未猜测数字范围内：[1, 100]。假定使用变量low和high来标记范围的两个端点。每次循环迭代，都需要确保如果数字在循环开始时处于[low, high]范围内，那么在当前循环迭代结束时，该数字仍然在计算出来的新范围内。

我们的目标是以代码形式表达循环不变量（因为也称为“循环不变式”），以便在运行时检测到违反假设的情况。遗憾的是，由于计算机不知道秘密数字，所以无法直接在代码中表达这个条件，但至少可以添加一个注释。

while(!guessed) {

// INVARIANT（不变量）: 数字在范围[low, high]内

...

}

当用户说猜测的一个数太大或太小时，如果实际并非如此，那么会发生什么？这是一种“欺骗”，会将秘密数字排除在新的子范围之外。由于一个谎言会滋生更多的谎言，所以最终范围会缩小到零（因为范围每次都缩小一半，而秘密数字不在其中）。可以用下面的程序表达这种情况。

//: C02:HiLo.cpp {RunByHand}

// 过个“猜大小”游戏利用了循环不变量

#include <cstdlib>

#include <iostream>

#include <string>

using namespace std;

int main() {

cout << "请想一个1到100之间的数字" << endl

<< "我会猜这个数；告诉我猜的数是太大(H)，还是太小(L)" << endl;

int low = 1, high = 100; // 初始范围

bool guessed = false; // 是否猜中的标志

while(!guessed) {

// INVARIANT(不变量): 数字在范围[low, high]内

if(low > high) { // 违反不变量

cout << "你作弊！我不玩了！" << endl;

return EXIT\_FAILURE;

}

int guess = (low + high) / 2;

cout << "我猜的数是" << guess << "。";

cout << "(H)太大, (L)太小, 或者 (E)猜对了? ";

string response;

cin >> response;

switch(toupper(response[0])) {

case 'H':

case 'h':

high = guess - 1;

break;

case 'L':

case 'l':

low = guess + 1;

break;

case 'E':

case 'e':

guessed = true;

break;

default:

cout << "无效回答" << endl;

continue;

}

}

cout << "我猜到了！" << endl;

return EXIT\_SUCCESS;

} ///:~

一旦low > high，就表明违反了不变量。如果用户总是诚实地回答，那么程序肯定会在有限次数内猜出那个秘密数字。

我们还采用了C语言的一个标准技术，即通过main函数的返回值向调用上下文（调用者）报告程序的执行状态。C++允许使用return 0;表示程序执行成功，这种写法具有可移植性，在所有平台上都能正常编译。但是，并没有一个具有可移植性的、表示失败的返回值。因此，我们使用<cstdlib>中为此目的而声明的宏EXIT\_FAILURE。为了保持代码风格的一致性，以后只要使用了EXIT\_FAILURE，我们就会同时使用EXIT\_SUCCESS，尽管后者总是被定义为零。

## 2.1 断言

Hi-Lo程序中的条件依赖于用户输入，因此事实上无法防止故意违反不变量的情况。然而，不变量通常只依赖于我们写的代码，所以如果正确实现了设计，它们就会持续生效。在这种情况下，创建一个**断言**（assertion）显得更清楚。断言是一种肯定性的陈述，它揭示了我们的设计意图和决策。

假设要实现一个整数向量（可以根据需要扩展的数组）。在这种情况下，将元素添加到向量中的函数首先必须验证容纳了元素的基础数组是否还有空白位置[[68]](#footnote-68)。如果没有，那么首先需要请求更多的堆空间，然后将现有元素复制到新的空间，最后才能添加新元素（同时释放旧数组）。下面展示了这样的一个函数。

void MyVector::push\_back(int x) {

if(nextSlot == capacity)

grow();

assert(nextSlot < capacity);

data[nextSlot++] = x;

}

在本例中，data是一个具有capacity个存储位置（slot）的动态整数数组，其中有nextSlot个位置已被占用。grow()函数的作用是对data进行扩容，使得新的capacity值严格大于nextSlot。[[69]](#footnote-69)MyVector 的行为要想正确，就必须依赖于这一设计决策，而且如果其他支持代码都正确无误的话，那么它永远不会失败。我们通过头文件<cassert>中定义的assert()宏来断言这一条件。

标准 C 库中的assert()宏简单明了，并且可移植。[[70]](#footnote-70)如果作为参数提供的条件表达式求值为非零值，那么程序将继续执行，不会终止。相反，如果条件不满足，那么会向标准错误通道（通常是控制台）打印一条消息，其中显示了表达式的文本以及源文件名和行号，最后终止程序。这种处理方式是否过于极端？事实上，当基本的设计假设失败时，让程序带着“暗伤”继续执行会造成更严重的后果。此时，程序必须立即终止，以避免更大的问题。

如果一切顺利，在发布最终产品之前，我们将完成对代码的全面测试，确保所有断言都保持完整。（稍后会更多地讨论测试。）然而，根据应用程序的具体情况，部署后如果还要在运行时测试所有断言，那么可能会对性能产生显著的影响。在这种情况下，可以通过定义NDEBUG宏并重新生成（rebuild）应用程序来自动移除断言代码，从而优化性能。

为了理解这具体是如何工作的，请注意assert()的典型实现看起来像下面这样：

#ifdef NDEBUG

#define assert(cond) ((void)0)

#else

void assertImpl(const char\*, const char\*, long);

#define assert(cond) \

((cond) ? (void)0 : assertImpl(???))

#endif

如果当前定义了NDEBUG宏，那么代码将退化为表达式(void) 0，所以在编译流中，唯一剩下的就是本质上为空的一个语句，这是因为我们在调用每个assert()时都会附加分号。[[71]](#footnote-71)相反，如果当前没有定义NDEBUG，那么assert(cond)会展开为一个条件语句，当cond为零时，会调用一个具体依赖于编译器的函数（我们命名为assertImpl()），并向函数传递一个字符串实参来表示cond文本，还要传递断言所在的文件名和行号。（本例使用“???”作为占位符，但无论代表cond的字符串，还是文件名和宏在文件中出现的行号，都是在运行时计算的。但是，这些值具体如何获得与当前的讨论无关。）如果想在程序的不同位置启用或禁用断言，那么不仅需要#define或#undef NDEBUG，还需要重新包含<cassert>。预处理器一旦遇到宏就会对其进行求值，并会使用包含<cassert>时适用的任何NDEBUG状态。如果想在整个程序中仅定义一次NDEBUG，那么最常见的方式是把它作为编译器选项。这可以在IDE（例如，Visual Studio）中设置，也可以在命令行上指定，如下所示：

mycc -DNDEBUG myfile

大多数编译器都使用-D选项来定义宏，例如，在使用Microsoft C++编译器时，可以使用cl.exe /DNDEBUG来定义NDEBUG宏。这种方式的优点在于，断言可以作为一种内嵌文档，保留在源代码中，同时在发布版本中可以被轻松地禁用，从而避免引入额外的运行时开销。因此，断言应该只用于检查那些不会影响程序逻辑的条件（即不变量），例如检查函数参数的有效性、数据结构的一致性等。

进行发布生成（release build）时，使用NDEBUG是否是一个好主意仍然存在争议。计算机科学领域最有影响力的科学家之一东尼·霍尔（Tony Hoare）[[72]](#footnote-72)指出，关闭像断言这样的运行时检查，类似于一位帆船爱好者在陆地上训练时穿着救生衣，但在出海时却将其丢弃。[[73]](#footnote-73)如果生产环境中断言失败，那么面临的问题比性能下降严重得多，因此要谨慎选择。

并非所有条件都适合通过断言来强制。如果发生用户错误（例如，用户打错了字）和运行时资源分配失败，那么应该通过抛出异常来指示（参见第1章）。在初步编写代码时，开发者可能为了方便而使用断言来处理大多数错误情况，并打算在以后替换为更健壮的异常处理。但是，请谨慎对待这种做法，因为很可能会忘记在后续阶段做出所有必要的更改。记住，断言旨在验证只会因为程序员的逻辑错误才会失败的设计决策。理想情况是在开发过程中解决所有违反断言的问题。不要对那些不在自己完全控制下的条件使用断言（例如，需要依赖用户输入的时候）。特别是，不应该使用断言来验证传递给函数的实参的有效性；相反，此时应抛出logic\_error。

伯特兰·迈耶（Bertrand Meyer ）在他的**契约设计**（Design by Contract）方法论中将断言正式化为确保程序正确性的工具。[[74]](#footnote-74)每个函数与客户端之间都有一个隐含的契约，即给定一定的前置条件（precondition），可以保证一定的后置条件（postcondition）。换句话说，前置条件是为了使用函数而必须满足的要求，例如提供特定范围内的实参。而后置条件是函数通过返回值或副作用[[75]](#footnote-75)而交付的结果。[[76]](#footnote-76)

若客户端程序未能提供有效的输入，就必须告诉契约被违反了。但是，这不是退出程序的最佳时机（尽管这样做完全正当。毕竟，它“违约”了嘛）。这种情况适合抛出一个异常。这就是为什么标准C++库会抛出从logic\_error派生的各种异常的原因，例如out\_of\_range（越界）。[[77]](#footnote-77)然而，如果有一个只有自己才会调用的函数，比如自定义类中的某个私有函数，那么就适合使用assert()宏了，因为当前的情况完全由自己掌控，而且肯定希望在交付前完成代码的调试。

若后置条件失败，表明出现了程序错误。对于**任何位置的任何不变量**，包括函数末尾的后置条件测试，使用断言都是恰当的。这尤其适用于维护对象状态的类成员函数。例如，在之前的MyVector示例中，所有公共成员函数的一个合理的不变量是：

assert(0 <= nextSlot && nextSlot <= capacity);

或者，如果nextSlot是一个无符号整数，那么可以简单地写成：

assert(nextSlot <= capacity);

这种不变量被称为**类不变量[[78]](#footnote-78)**，并且可以通过断言来合理地予以强制。子类在其基类中扮演着分包商（subcontractor）的角色，因为它们必须遵守基类与其客户端之间的原始契约。因此，派生类中的前置条件不能超出基契约中的要求，而后置条件则必须至少交付同样的结果。[[79]](#footnote-79)有一句简单的话可以帮助我们记忆：“要求不能多，承诺不能少”（Require no more; promise no less）。这句话最早是由 Marshall Cline 和 Greg Lomow 在他们的*C++ FAQs*（C++经典问答）一书中提出的（Addison-Wesley，1994）。由于派生类的前置条件可以比基类更宽松，因此我们称其为**逆变**。相反，派生类的后置条件必须与基类的后置条件相同或更强[[80]](#footnote-80)，因此我们称其为**协变**。这也可以解释我们在第1章中提到的异常规范的协变（或协变性，参见1.7.2节）。

然而，对返回给客户端的结果进行验证，这无非就是**测试**。因此，在这种情况下使用后置条件断言是多余的。诚然，这是一种很好的文档记录方式。但是，不止一个开发者被误导，错误地将后置条件断言当作单元测试的替代品。[[81]](#footnote-81)

## 2.2 简单的单元测试框架

编写软件是为了满足特定的需求。[[82]](#footnote-82)但是，创建真正的“需求”非常困难，用户所谓的“需求”可能每天都在改变。在每周的项目会议上，可能发现自己刚花了一周时间做的工作并不是用户真正想要的。

如果没有看到一个不断演进的、能实际工作的系统，那么人们无法阐述自己真正的软件需求。最好的做法是规定一点，设计一点，编码一点，再测试一点。然后，在对工作成果（交付物）进行评估之后，再来一遍。以这种方式进行迭代开发，是面向对象方法的一大进步之一。但是，它需要能编写灵活代码的敏捷程序员。而改变是最难的！[[83]](#footnote-83)

程序员对代码的精益求精是推动软件演进的另一重要动力。我们常常希望通过改进代码设计来提升代码的可维护性、可扩展性。然而，现实中，许多项目都面临着技术债务的困扰：耦合度高、难以维护的代码就像一团乱麻，让维护人员苦不堪言。管理层出于对系统稳定性的考虑，往往会对大规模的代码重构持谨慎态度。但“不改变，就无法改进”的观点同样成立。如果一直固守陈旧的代码，最终可能导致系统难以适应新的需求，不得不推翻重来。

幸好，**重构**（refactoring）这种在不改变外部行为的前提下优化代码内部结构的方法越来越受到重视。重构能有效提升代码的可读性和可维护性，并为后续的扩展提供基础。[[84]](#footnote-84)常见的重构手段包括函数提取、函数合并、将成员函数替换成对象、对成员函数或类进行参数化、多态性替换条件语句等。通过这些手段，我们可以逐步改善代码质量，让代码更好地适应不断变化的需求。

无论是用户需求的驱动还是程序员的主动优化，软件系统都面临着持续变化的挑战。任何改动都可能对既有功能产生影响，甚至引入新的问题。因此，构建能够适应变化、不断改进的代码体系显得尤为重要。

**极限编程**（XP）[[85]](#footnote-85)是众多支持敏捷开发的实践之一。它提倡通过频繁的迭代和增量开发来快速响应变化的需求。而要实现这一目标，一个易于使用的自动化单元测试框架至关重要。需要注意的是，虽然单元测试能有效提高代码质量，但专业的测试人员在软件质量保障体系中仍然扮演着不可或缺的角色。在这里，我们主要探讨的是如何帮助开发者更好地理解需求，并通过编写单元测试来验证代码的正确性。

开发者之所以编写**单元测试**（unit test），主要是为了有信心说出下面这两句对任何开发者来说都非常重要的话：

1. 我理解需求；
2. 我的代码满足了这些需求（据我所知）。

编写单元测试的目的是提前明确编码目标，最终提升代码的质量。通过提前设计测试用例，开发者可以更清晰地把握功能需求，从而编写出更符合预期、更健壮的代码。这种“测试先行”的做法不仅有助于集中注意力，而且往往能提高开发效率，正如极限编程所倡导的：

测试+编程比单纯编程更快。[[86]](#footnote-86)

“测试先行”还可以帮助开发者更早地发现并修复潜在的缺陷（尤其对边界条件的处理[[87]](#footnote-87)），可以使代码更加健壮。

一旦代码通过了所有测试，就知道假如整个软件系统不起作用，那么问题很可能并不出在自己的代码上。“我的所有测试都通过了”就是一个强有力的证据。[[88]](#footnote-88)

### 2.2.1 自动化测试

那么，一个单元测试具体应该是什么样的呢？开发者往往会使用一些正常的输入，并以“手动”（换言之，靠自己目视）方式验证程序是否能产生预期输出。这种方法虽然简单，但存在两个潜在风险。

首先，程序并不总是能收到“正常”的输入。我们都知道应该对输入边界进行测试，但在开发初期，往往会忽略这一点。一个更有效的方法是：在编写代码之前，先编写测试用例。 此时，我们以“测试者”的视角思考，设想各种可能导致程序出错的输入，然后编写测试用例来验证程序的行为。随后，再以“开发者”的视角，编写代码来通过这些测试。这种“测试先行”的开发方式，能帮助我们写出更加健壮和可靠的代码。

第二个风险是，手动检查输出既费时又容易出错。人类能做到的几乎任何事情，计算机都能做到且更加高效。与此同时，还可以避免人为因素而导致的错误。因此，最好的方式是将测试用例表述为一组**布尔表达式**，由测试程序自动执行并判断结果是否为真。

例如，假设需要构建一个具有以下属性的Date（日期）类：

* 日期可以通过提供一个字符串（YYYYMMDD）、三个整数（Y，M，D）或者什么都不提供（默认今天的日期）来初始化。
* 日期对象可以返回其年、月、日或形式为“YYYYMMDD”的字符串。
* 支持各种比较，并且可以计算两个日期之间的持续时间（以年、月、日为单位）。
* 要比较的日期要允许跨越任意数量的世纪（例如，1600—2200）。

这个类可以存储代表年、月、日的三个整数（只需确保年份至少16位大小，以满足最后一个要求）。基于上述条件，可以为Date类写一个如下所示的接口：

//: C02:Date1.h

// 因为是对Date.h的初次尝试，所以命名为Date1.h

#ifndef DATE1\_H

#define DATE1\_H

#include <string>

class Date {

public:

// 用一个结构体来保存经过的时间

struct Duration {

int years; // 年

int months; // 月

int days; // 日

Duration(int y, int m, int d)

: years(y), months(m), days(d) {} // 构造函数

};

Date(); // 默认构造函数

Date(int year, int month, int day); // 带参数的构造函数

Date(const std::string&); // 从字符串构造

// 获取年、月、日

int getYear() const;

int getMonth() const;

int getDay() const;

// 将日期转换为字符串

std::string toString() const;

// 这些友元函数重载了比较操作符，以便比较日期

friend bool operator<(const Date&, const Date&);

friend bool operator>(const Date&, const Date&);

friend bool operator<=(const Date&, const Date&);

friend bool operator>=(const Date&, const Date&);

friend bool operator==(const Date&, const Date&);

friend bool operator!=(const Date&, const Date&);

// 计算两个日期之间的间隔

friend Date::Duration duration(const Date&, const Date&);

};

#endif // DATE1\_H ///:~

在具体实现这个类之前，可以先通过编写测试程序的开头来巩固自己对需求的理解，如下所示：[[89]](#footnote-89)

//: C02:SimpleDateTest.cpp

//{L} Date

#include <iostream>

#include "Date.h" // 来自附录B

using namespace std;

// 测试框架

int nPass = 0, nFail = 0;

void test(bool t) { if(t) nPass++; else nFail++; }

int main() {

Date mybday(1951, 10, 1);

test(mybday.getYear() == 1951);

test(mybday.getMonth() == 10);

test(mybday.getDay() == 1);

cout << "通过: " << nPass << ",失败: "

<< nFail << endl;

}

/\* 预期的输出：

通过: 3, 失败: 0

\*/ ///:~

在这个简单的示例中，test()函数维护全局变量nPass和nFail（即通过和失败的次数）。唯一需要手动检查的就是最后的计数。如果测试失败，一个更复杂的test()函数应显示适当的信息。本章后面描述的框架就提供了这样的一个测试函数。

写好测试用例后，Date类实现起来就更有把握了。通过“测试驱动开发”，我们可以更早地发现并修复问题，提高代码的质量。由于先写测试，所以更有可能考虑到各种边界条件，例如日期的有效性、闰年等，从而减少代码中的错误，而且更有可能第一次就写出正确的代码。无论如何，我们可以如此逐步迭代，直至满足所有需求。如此进行练习，我们可能会写出下面这个新版本的Date类测试。[[90]](#footnote-90)

//: C02:SimpleDateTest2.cpp

//{L} Date

#include <iostream>

#include "Date.h"

using namespace std;

// 测试框架

int nPass = 0, nFail = 0;

void test(bool t) { if(t) ++nPass; else ++nFail; }

int main() {

// 创建三个Date对象：一个表示作者的生日，一个表示今天，一个表示生日的前一天

Date mybday(1951, 10, 1);

Date today;

Date myevebday("19510930");

// 测试重载的比较操作符

test(mybday < today);

test(mybday <= today);

test(mybday != today);

test(mybday == mybday);

test(mybday >= mybday);

test(mybday <= mybday);

test(myevebday < mybday);

test(mybday > myevebday);

test(mybday >= myevebday);

test(mybday != myevebday);

// 测试getYear()、getMonth()、getDay()和toString()成员函数

test(mybday.getYear() == 1951);

test(mybday.getMonth() == 10);

test(mybday.getDay() == 1);

test(myevebday.getYear() == 1951);

test(myevebday.getMonth() == 9);

test(myevebday.getDay() == 30);

test(mybday.toString() == "19511001");

test(myevebday.toString() == "19510930");

// 测试duration()函数（一个友元），计算两个日期之间的差

Date d2(2003, 7, 4);

Date::Duration dur = duration(mybday, d2);

test(dur.years == 51);

test(dur.months == 9);

test(dur.days == 3);

// 报告结果:

cout << "通过: " << nPass << ",失败: "

<< nFail << endl;

} ///:~

此测试还可以进一步完善。例如，当前尚未测试长的持续时间是否得到了正确处理。由于篇幅所限，所以这里不再赘述。但是，想必你已经大致理解了。注意，Date类的完整实现请参见附录B中的Date.h和Date.cpp文件。[[91]](#footnote-91)

### 2.2.2 TestSuit框架

可以从网上下载一些自动化的C++单元测试工具，例如CppUnit。[[92]](#footnote-92)我们的目的不仅是呈现一个易于使用的测试机制，还要求易于理解其内部原理，甚至必要时可以做修改。因此，本着“做最简单可行的事”（Do The Simplest Thing That Could Possibly Work，DTSTCPW）[[93]](#footnote-93)的精神，我们开发了**TestSuite框架**，它本质上是一个名为TestSuite的命名空间，其中包含两个关键的类：Test和Suite。

Test类是抽象基类，我们从它派生出一个测试类。它跟踪“通过”和“失败”的数量，并显示对任何失败的测试条件进行描述的文本。只需在派生类中重写run()成员函数，后者进而为我们定义的每个布尔测试条件调用test\_()宏。

如以下程序所示，为了使用该框架为Date类定义一个测试，可以从Test派生出一个名为DateTest的测试类。

//: C02:DateTest.h

#ifndef DATETEST\_H

#define DATETEST\_H

#include "Date.h"

#include "../TestSuite/Test.h"

class DateTest : public TestSuite::Test {

Date mybday;

Date today;

Date myevebday;

public:

DateTest(): mybday(1951, 10, 1), myevebday("19510930") {}

void run() {

testOps();

testFunctions();

testDuration();

}

void testOps() {

test\_(mybday < today);

test\_(mybday <= today);

test\_(mybday != today);

test\_(mybday == mybday);

test\_(mybday >= mybday);

test\_(mybday <= mybday);

test\_(myevebday < mybday);

test\_(mybday > myevebday);

test\_(mybday >= myevebday);

test\_(mybday != myevebday);

}

void testFunctions() {

test\_(mybday.getYear() == 1951);

test\_(mybday.getMonth() == 10);

test\_(mybday.getDay() == 1);

test\_(myevebday.getYear() == 1951);

test\_(myevebday.getMonth() == 9);

test\_(myevebday.getDay() == 30);

test\_(mybday.toString() == "19511001");

test\_(myevebday.toString() == "19510930");

}

void testDuration() {

Date d2(2003, 7, 4);

Date::Duration dur = duration(mybday, d2);

test\_(dur.years == 51);

test\_(dur.months == 9);

test\_(dur.days == 3);

}

};

#endif // DATETEST\_H ///:~

要想运行测试，只需实例化一个DateTest对象，并调用它的run()成员函数。[[94]](#footnote-94)

//: C02:DateTest.cpp

// 自动化测试(通过一个框架)

//{L} Date ../TestSuite/Test

#include <iostream>

#include "DateTest.h"

using namespace std;

int main() {

DateTest test;

test.run();

return test.report();

}

/\* 输出:

测试"class DateTest":

通过: 21 失败: 0

\*/ ///:~

Test::report()函数显示代码中描述的输出并返回失败数量，因此适合用作main()的返回值。Test类利用 RTTI[[95]](#footnote-95)来实时获取类名（例如，DateTest）来进行报告。除此之外，还可以利用一个名为setStream()的成员函数将测试结果发送到文件，而不是发送到默认的标准输出（通常是控制台）。本章稍后会展示Test类的具体实现。

test\_()宏可以提取失败的布尔条件的文本及其文件名和行号。[[96]](#footnote-96)如果想体验当发生失败时会发生什么，可以在代码中故意引入一个错误，例如反转前面示例代码中DateTest::testOps()中第一个test\_()调用的条件（也就是说，将mybday < today改成mybday > today）。输出会明确指出哪个测试出错了，以及具体发生在什么位置。

class DateTest失败:(mybday > today) , F:\...\ThinkingInCpp\第2章\DateTest.h (行19)

测试"class DateTest":

通过: 20 失败: 1

除了test\_()，这个测试框架还包括succeed\_()和fail\_()，它们适用于布尔测试无法处理的情况。如果要测试的类可能抛出异常，就适合使用这些函数。在测试过程中，请创建一组会导致发生异常的输入数据。如果没有抛出异常，就表明代码存在错误，应显式调用fail\_()来显示一条消息并更新“失败”计数。如果按预期抛出了异常，则调用succeed\_()更新“通过”计数。

例如，假设修改了两个非默认Date构造函数的规范，使其在输入参数不代表有效日期时抛出DateError异常（这是嵌套在Date内，并继承自std::logic\_error的一个异常类型）。

Date(const string& s) throw(DateError);

Date(int year, int month, int day) throw(DateError);

现在，DateTest::run()成员函数可以调用以下函数来测试异常处理：

void testExceptions() {

try {

Date d(0,0,0); // 无效日期

fail\_("Date获取int的构造函数未检测到无效日期");

} catch(Date::DateError&) {

succeed\_();

}

try {

Date d(""); // 无效日期

fail\_("Date获取字符串的构造函数未检测到无效日期");

} catch(Date::DateError&) {

succeed\_();

}

}

在这两种情况下，如果没有抛出异常，就认为编码有误。注意，必须向fail\_()手动传递一条消息，因为这里没有对一个布尔表达式进行求值。

### 2.2.3 测试套件

真实的项目通常包含许多类，因此需要一种方法对测试进行分组，这样只需一个命令即可测试整个项目。[[97]](#footnote-97)Suite类将测试组合成一个功能单元。可以通过addTest()成员函数向Suite添加自己的Test对象，或者使用addSuite()来包含一个现有的套件（suite）。为了说明这一点，下例将第3章使用了Test类的程序全部集中到一个单独的套件中。注意，该文件位于本书中文版配套资源的“第3章”子目录中。[[98]](#footnote-98)

//: C03:StringSuite.cpp

//{L} ../TestSuite/Test ../TestSuite/Suite

//{L} TrimTest

// 这个测试套件用于对第3章的代码进行测试

#include <iostream>

#include "../TestSuite/Suite.h"

#include "StringStorage.h"

#include "Sieve.h"

#include "Find.h"

#include "Rparse.h"

#include "TrimTest.h"

#include "CompStr.h"

using namespace std;

using namespace TestSuite;

int main() {

Suite suite("字符串测试");

suite.addTest(new StringStorageTest);

suite.addTest(new SieveTest);

suite.addTest(new FindTest);

suite.addTest(new RparseTest);

suite.addTest(new TrimTest);

suite.addTest(new CompStrTest);

suite.run();

long nFail = suite.report();

suite.free();

return nFail;

}

/\* 输出：

s1 = 62345

s2 = 12345

Suite "字符串测试"

==================

测试"class StringStorageTest":

通过: 2 失败: 0

测试"class SieveTest":

通过: 50 失败: 0

测试"class FindTest":

通过: 9 失败: 0

测试"class RparseTest":

通过: 8 失败: 0

测试"class TrimTest":

通过: 11 失败: 0

测试"class CompStrTest":

通过: 8 失败: 0

==================

\*/ ///:~

其中，有5个测试（的实现）均完整地包含在头（.h）文件中，只有TrimTest例外，因其包含了必须在实现（.cpp）文件中定义的静态数据。[[99]](#footnote-99)前两行输出来自对StringStorage的测试（请参见StringStorage.h）。必须将套件的名称作为构造函数的实参传入。Suite::run()成员函数会依次对其包含的每一个测试调用Test::run()。Suite::report()的行为与此相似，但可以将每个测试报告都输出到与套件报告不同的一个目标流。如果传递给addSuite()的测试已经关联了一个流指针，那么它会保留该指针。否则，会从Suite对象中获取流（类似于Test类，Suite构造函数的第二个参数也是可选的，默认为std::cout）。Suite的析构函数不会自动释放（delete）对象包含的Test指针，因其不需要在堆上分配；这些指针的释放由Suite::free()负责。

### 2.2.4 测试框架的源代码

测试框架代码位于TestSuite子目录（这是我们的一个“库”），与“第1章”、“第2章”等子目录同级。要使用它，需要在头文件中包含TestSuite子目录的搜索路径，链接目标文件（.o或.obj文件），并将TestSuite子目录包含在库搜索路径中。[[100]](#footnote-100)头文件Test.h的源代码如下所示：

// TestSuite:Test.h

#ifndef TEST\_H

#define TEST\_H

#include <string>

#include <iostream>

#include <cassert>

using std::string;

using std::ostream;

using std::cout;

// fail\_()使用下划线后缀来避免与ios::fail()冲突。

// 为了保持一致，test\_()和succeed\_()也添加了下划线后缀。

#define test\_(cond) \

do\_test(cond, #cond, \_\_FILE\_\_, \_\_LINE\_\_)

#define fail\_(str) \

do\_fail(str, \_\_FILE\_\_, \_\_LINE\_\_)

namespace TestSuite {

class Test {

// 指向输出流的指针，用于输出测试结果（默认为标准输出cout）

ostream\* osptr;

// 记录测试通过的次数

long nPass;

// 记录测试失败的次数

long nFail;

// Disallowed: 下面两个构造函数和赋值操作符被禁止使用

Test(const Test&);

Test& operator=(const Test&);

protected:

// 用于执行测试并报告结果的辅助函数

void do\_test(bool cond, const string& lbl, const char\* fname, long lineno);

// 用于报告测试失败的辅助函数

void do\_fail(const string& lbl, const char\* fname, long lineno);

public:

// 构造函数，可以指定输出流，默认为标准输出cout

Test(ostream\* osptr = &cout) {

this->osptr = osptr;

nPass = nFail = 0;

}

virtual ~Test() {} // 虚构析构函数，用于释放资源

// 纯虚函数，需由子类实现，用于执行具体的测试逻辑

virtual void run() = 0;

// 获取测试通过的次数

long getNumPassed() const { return nPass; }

// 获取测试失败的次数

long getNumFailed() const { return nFail; }

// 获取输出流指针

const ostream\* getStream() const { return osptr; }

// 设置输出流指针

void setStream(ostream\* osptr) { this->osptr = osptr; }

// 记录一次测试通过

void succeed\_() { ++nPass; }

// 生成测试报告

long report() const;

// 虚函数，用于重置测试结果计数器

virtual void reset() { nPass = nFail = 0; }

};

} // namespace TestSuite

#endif // TEST\_H///:~

注意，Test类包含三个虚函数：

* 一个虚析构函数
* 虚reset()函数
* 纯虚函数run()

如本书第1卷所述，除非基类包含虚析构函数，否则不能通过基类指针释放（delete）为派生类动态分配的堆对象。[[101]](#footnote-101)任何打算作为基类的类都应该包含虚析构函数（该类通常还至少应该有一个别的虚函数）。Test::reset()的默认实现是将“通过”和“失败”计数器重置为零。但是，完全可以重写（override）这个函数，并在自己的派生测试对象中重置数据的状态；只需确保在重写版本中显式调用Test::reset()以重置计数器。Test::run()成员函数是纯虚的；换言之，必须在派生类中重写它。

test\_()和fail\_()宏可以包含预处理器提供的文件名和行号信息。我们最初未在名称末尾添加下划线，但是fail\_宏与ios::fail()冲突了，会发生编译错误。

下面展示了Test类其余函数的实现：

//: TestSuite:Test.cpp {O}

#include "Test.h"

#include <iostream>

#include <typeinfo>

using namespace std;

using namespace TestSuite;

void Test::do\_test(bool cond, const std::string& lbl,

const char\* fname, long lineno) {

if (!cond)

do\_fail(lbl, fname, lineno);

else

succeed\_();

}

void Test::do\_fail(const std::string& lbl,

const char\* fname, long lineno) {

++nFail;

if (osptr) {

\*osptr << typeid(\*this).name()

<< " 失败:(" << lbl << ") , " << fname

<< " (行" << lineno << ")" << endl;

}

}

long Test::report() const {

if (osptr) {

\*osptr << "测试\"" << typeid(\*this).name()

<< "\":\n通过: " << nPass

<< "\t 失败: " << nFail

<< endl;

}

return nFail;

} ///:~

Test类跟踪成功和失败的测试次数，以及希望Test::report()向哪个流显示结果。test\_()和fail\_()宏从预处理器中提取当前文件名和行号信息，并将文件名传递给do\_test()，将行号传递给do\_fail()，这些函数显示消息并更新相应的计数器。注意，对“测试”对象进行拷贝和赋值是没有道理的，因此这里将它们的原型设为私有，并省略了它们各自的函数体。

Suite类的头文件如下所示：

//: TestSuite:Suite.h

#ifndef SUITE\_H

#define SUITE\_H

#include <vector>

#include <stdexcept>

#include "../TestSuite/Test.h"

using std::vector;

using std::logic\_error;

namespace TestSuite {

class TestSuiteError : public logic\_error {

public:

TestSuiteError(const string& s = "")

: logic\_error(s) {}

};

class Suite {

private:

string name;

ostream\* osptr;

vector<Test\*> tests;

void reset();

// 禁止的操作

Suite(const Suite&);

Suite& operator=(const Suite&);

public:

Suite(const string& name, ostream\* osptr = &cout)

: name(name) { this->osptr = osptr; }

string getName() const { return name; }

long getNumPassed() const;

long getNumFailed() const;

const ostream\* getStream() const { return osptr; }

void setStream(ostream\* osptr) { this->osptr = osptr; }

void addTest(Test\* t) throw(TestSuiteError);

void addSuite(const Suite&);

void run(); // 反复调用 Test::run()

long report() const;

void free(); // 释放“测试”对象

};

} // namespace TestSuite

#endif // SUITE\_H

Suite类用一个vector来保存指向其各个Test对象的指针。请注意addTest()成员函数的异常规范。向套件添加一个测试时，Suite::addTest()会验证传递的指针非空；如果是空指针，它会抛出一个TestSuiteError异常。由于这确保了不可能向套件添加空指针，因此addSuite()在其每个测试中都断言这一条件，遍历测试vector的其他函数也如此处理（参见如下所示的实现）。和Test类一样，拷贝和赋值操作也被禁止。

//: TestSuite:Suite.cpp {O}

#include "Suite.h"

#include <iostream>

#include <cassert>

#include <cstddef>

using namespace std;

using namespace TestSuite;

void Suite::addTest(Test\* t) throw(TestSuiteError) {

// 验证测试有效并且有一个输出流

if (t == 0)

throw TestSuiteError("Null test in Suite::addTest");

else if (osptr && !t->getStream())

t->setStream(osptr);

tests.push\_back(t);

t->reset();

}

void Suite::addSuite(const Suite& s) {

for (size\_t i = 0; i < s.tests.size(); ++i) {

assert(tests[i]);

addTest(s.tests[i]);

}

}

void Suite::free() {

for (size\_t i = 0; i < tests.size(); ++i) {

delete tests[i];

tests[i] = 0;

}

}

void Suite::run() {

reset();

for (size\_t i = 0; i < tests.size(); ++i) {

assert(tests[i]);

tests[i]->run();

}

}

long Suite::report() const {

if (osptr) {

long totFail = 0;

\*osptr << "Suite \"" << name << "\"\n=======";

size\_t i;

for (i = 0; i < name.size(); ++i)

\*osptr << '=';

\*osptr << "=" << endl;

for (i = 0; i < tests.size(); ++i) {

assert(tests[i]);

totFail += tests[i]->report();

}

\*osptr << "=======";

for (i = 0; i < name.size(); ++i)

\*osptr << '=';

\*osptr << "=" << endl;

return totFail;

} else

return getNumFailed();

}

// 获取通过次数

long Suite::getNumPassed() const {

long totPass = 0;

for (size\_t i = 0; i < tests.size(); ++i) {

assert(tests[i]);

totPass += tests[i]->getNumPassed();

}

return totPass;

}

// 获取失败次数

long Suite::getNumFailed() const {

long totFail = 0;

for (size\_t i = 0; i < tests.size(); ++i) {

assert(tests[i]);

totFail += tests[i]->getNumFailed();

}

return totFail;

}

// 重置

void Suite::reset() {

for (size\_t i = 0; i < tests.size(); ++i) {

assert(tests[i]);

tests[i]->reset();

}

} ///:~

本书以后会在适当的时候继续使用TestSuite框架。

## 2.3 调试技术

最好的调试习惯就是使用如本章开头所述的断言。这有助于在真正出现问题之前发现逻辑错误。本节解释了其他一些可能在调试过程中有帮助的技巧和技术。

### 2.3.1 跟踪宏

有的时候，我们需要在代码执行时打印每个语句的源代码，将其输出到cout或跟踪文件（trace file）中。以下是一个实现此功能的预处理器宏：

#define TRACE(ARG) cout << #ARG << endl; ARG

然后，可以使用这个宏来包围想要跟踪的语句。但这可能会引入新问题。例如，如果用TRACE()宏来包围以下语句：

for (int i = 0; i < 100; i++)

cout << i << endl;

那么将得到：

TRACE(for(int i = 0; i < 100; i++))

TRACE(cout << i << endl;)

这将展开为：

cout << "for(int i = 0; i < 100; i++)" << endl;

for(int i = 0; i < 100; i++)

cout << i << endl;

cout << "cout << i << endl;" << endl;

cout << i << endl;

由于输出内容过于繁琐，所以自然并非我们所愿。因此，必须谨慎使用这种技术。

以下是TRACE()宏的一个变体：

#define D(a) cout << #a "=[" << a << "]" << endl;

如果想显示一个表达式，那么将其放在对D()的一个调用中即可。这样就会显示该表达式，然后显示其求值结果（前提是结果类型重载了<<操作符函数）。例如，可以像这样写：D(a + b)。注意，随时都可以使用这个宏来检查中间值。

这两个宏代表了调试器的两个基本功能：跟踪代码执行和显示值。一个好的调试器自然是优秀的生产力工具，但有的时候，我们会发现调试器不可用，或者用起来不方便。相反，不管在什么情况下，这些宏总是有效的。

### 2.3.2 跟踪文件

免责声明：本节和下一节的代码并非严格遵循C++标准。特别是，我们通过宏重新定义了cout和new。如果不小心，可能会导致意外结果。以下示例在我们使用的所有编译器上都能正常工作，并能提供有用的信息。这是本书唯一偏离标准兼容编码实践的地方。请谨慎使用，并注意潜在的风险！注意，为了使代码生效，必须使用“using声明”，这样做是为了省略cout的命名空间前缀；也就是说，不能使用std::cout。

以下代码可以轻松创建一个跟踪文件，并将所有原本输出到cout的内容写入该文件。只需添加#define TRACEON指令并包含头文件（当然，也可以考虑在自己的文件中直接写那两行关键的代码[[102]](#footnote-102)）。

//: C03:Trace.h

// 创建一个跟踪文件

#ifndef TRACE\_H

#define TRACE\_H

#include <fstream>

#ifdef TRACEON

std::ofstream TRACEFILE\_\_("TRACE.OUT");

#define cout TRACEFILE\_\_

#endif

#endif // TRACE\_H ///:~

下例对上述文件进行了一次简单的测试。[[103]](#footnote-103)

//: C03:Tracetst.cpp {-bor}

#include <iostream>

#include <fstream>

#include "../require.h"

using namespace std;

#define TRACEON

#include "Trace.h"

int main() {

ifstream f("Tracetst.cpp");

assure(f, "Tracetst.cpp");

cout << f.rdbuf

(); // 将指定文件的内容转储到TRACE.OUT文件中

} ///:~

由于Trace.h将cout文本转换为其他内容，因此程序中的所有cout语句现在都将信息发送到跟踪文件。这是一种将输出捕获到文件中的简便方法，以防操作系统有的时候不方便进行输出重定向。

### 2.3.3 发现内存泄漏

以下简单的调试技术已在本书第 1 卷中进行了说明：

* 为了检查数组边界，请为所有数组都使用第 1 卷展示的 C16:Array3.cpp 中的 Array 模板。准备好发布时，可以关闭检查以提高效率。（虽然这个技术还不能处理使用数组指针的情况。[[104]](#footnote-104)）
* 检查基类中的非虚析构函数。

#### 跟踪new/delete和malloc/free

内存分配错误在C++程序中很常见，而且难以排查。常见的错误包括：错误地为不在自由存储区（free store）上的内存调用delete（即试图释放非动态分配的内存）[[105]](#footnote-105)、重复释放已释放的内存以及最容易忽视的内存泄漏（忘记delete指针）。本节将介绍一个能有效监测并预防这些问题的系统。

以下免责声明是对上一节的补充：由于重载了new，因此可能存在平台兼容性问题。此外，该技术仅适用于那些没有显式调用operator new()函数的程序。本书尽量只展示完全符合C++标准的代码，但这是一个特例，主要是出于以下原因：

1. 虽然在技术上不符合标准，但它在许多编译器上都有效。[[106]](#footnote-106)
2. 我们在此过程中阐述了一些有用的编程思想。

要使用这个内存检查系统，只需包含头文件MemCheck.h，将MemCheck.obj文件链接到自己的应用程序以拦截所有 new 和 delete 操作，并调用宏MEM\_ON()（本节稍后就会解释）来启动内存跟踪。然后，包含了所有分配和释放操作的记录会打印到标准输出（stdout）。

该系统通过 operator new 的放置（placement）语法，在每次调用 new 时都存储与调用它们的文件和代码行有关的信息。[[107]](#footnote-107)尽管放置语法通常用于将对象放置在指定内存位置，但还可以用它来创建带有多个实参的operator new()函数。例如，下例使用这个技术在每次调用new时存储\_\_FILE\_\_和\_\_LINE\_\_宏的结果。

//: C02:MemCheck.h

#ifndef MEMCHECK\_H

#define MEMCHECK\_H

#include <cstddef> // 为了使用size\_t

// 重载new操作符（标量和数组版本）

void\* operator new(std::size\_t, const char\*, long);

void\* operator new[](std::size\_t, const char\*, long);

#define new new (\_\_FILE\_\_, \_\_LINE\_\_)

extern bool traceFlag;

#define TRACE\_ON() traceFlag = true

#define TRACE\_OFF() traceFlag = false

extern bool activeFlag;

#define MEM\_ON() activeFlag = true

#define MEM\_OFF() activeFlag = false

#endif // MEMCHECK\_H ///:~

注意，任何源文件如果想跟踪自由存储区活动，那么都应该包含此文件，但请把它放到**最后**（其他#include指令之后）。 标准库中的大多数头文件都是模板，并且由于大多数编译器采用的都是模板编译的**包含模型**（意味着所有源代码都在头文件中），因此当编译器处理MemCheck.h时，会将其中的宏替换应用于后续包含的所有头文件中。如果MemCheck.h被过早包含，那么标准库中的new操作符也会被替换，可能导致编译错误或不可预期的行为。毕竟，我们只想跟踪自己的内存错误，而不想跟踪库的内存错误。

以下文件包含内存跟踪系统的具体实现。注意，所有操作均使用C标准I/O而不是C++ iostreams[[108]](#footnote-108)。这应该不会有任何区别，因为我们没有干扰iostream对自由存储区的使用。[[109]](#footnote-109)但是，当我们尝试改为使用iostream而不是<cstdio>时，一些编译器会出现问题。相反，这个<cstdio>版本在所有编译器上都能顺利通过。

//: C02:MemCheck.cpp {O}

#include <cstdio>

#include <cstdlib>

#include <cassert>

#include <cstddef>

using namespace std;

#undef new

// 通过MemCheck.h中的宏设置全局标志

bool traceFlag = true;

bool activeFlag = false;

namespace {

// 内存映射记录项（memory map entry）的类型，一个结构体

struct Info {

void\* ptr;

const char\* file;

long line;

};

// 内存映射数据

const size\_t MAXPTRS = 10000u;

Info memMap[MAXPTRS];

size\_t nptrs = 0;

// 在映射中搜索地址

int findPtr(void\* p) {

for (size\_t i = 0; i < nptrs; ++i) {

if (memMap[i].ptr == p) {

return i;

}

}

return -1;

}

void delPtr(void\* p) {

int pos = findPtr(p);

assert(pos >= 0);

// 从映射中移除指针

for (size\_t i = pos; i < nptrs - 1; ++i) {

memMap[i] = memMap[i + 1];

}

--nptrs;

}

// 这个dummy（假）类型唯一的作用就是提供一个静态析构函数

struct Sentinel {

~Sentinel() {

if (nptrs > 0) {

printf("在以下位置发生内存泄漏:\n");

for (size\_t i = 0; i < nptrs; ++i) {

printf("\t%p (文件: %s, 行号: %ld)\n",

memMap[i].ptr, memMap[i].file, memMap[i].line);

}

} else {

printf("未发生用户内存泄漏!\n");

}

}

};

// 实例化一个静态的dummy（假）对象

Sentinel s;

} // 结束匿名命名空间

// 重载标量版本的new

void\* operator new(size\_t siz, const char\* file, long line) {

void\* p = malloc(siz);

if (activeFlag) {

if (nptrs == MAXPTRS) {

printf("内存映射太小(请增大MAXPTRS)\n");

exit(1);

}

memMap[nptrs].ptr = p;

memMap[nptrs].file = file;

memMap[nptrs].line = line;

++nptrs;

}

if (traceFlag) {

printf("在地址 %p 处分配了 %u 字节", p, siz);

printf("(文件: %s, 行号: %ld)\n", file, line);

}

return p;

}

// 重载数组版本的new

void\* operator new[](size\_t siz, const char\* file, long line) {

return operator new(siz, file, line);

}

// 重载标量版本的delete

void operator delete(void\* p) {

if (findPtr(p) >= 0) {

free(p);

assert(nptrs > 0);

delPtr(p);

if (traceFlag) {

printf("已释放地址%p处的内存\n", p);

}

} else if (!p && activeFlag) {

printf("试图delete未知指针: %p\n", p);

}

}

// 重载数组版本的delete。为了简化编程，把所有操作都转发给了标量版本

void operator delete[](void\* p) {

operator delete(p);

}

///:~

注意，traceFlag和activeFlag被设置为全局布尔标志，因此可以在代码中通过宏TRACE\_ON()、TRACE\_OFF()、MEM\_ON()和MEM\_OFF()进行修改。我们通常将main()中的所有代码都包围在MEM\_ON()和MEM\_OFF()之间，以确保对程序的内存分配和释放进行完整的跟踪。跟踪会回显operator new() 和operator delete()的替代函数的活动。这个功能是默认开启的，但也可以通过TRACE\_OFF()关闭。无论在情况下，始终都会打印最终结果（请参见本章稍后的测试）。

MemCheck机制为了跟踪内存，会将operator new()分配的所有地址都存储在由Info结构体构成的一个数组中。结构体中还容纳了调用new时的文件名和行号。为了防止与全局命名空间中的任何名称发生冲突，我们将尽可能多的信息都保存在一个匿名命名空间中。Sentinel是一个特殊的类（称为“哨兵”类或者“假”类），它唯一的用处就是提供当程序关闭时调用的一个静态对象析构函数。[[110]](#footnote-110)该析构函数检查memMap中的所有记录，以判断是否有任何指针等待delete（但凡发现一个，就表明存在内存泄漏）。[[111]](#footnote-111)

我们的operator new()使用malloc()来分配内存，然后将指针及其关联的文件信息添加到memMap数组中。operator delete()函数通过调用free()并递减nptrs来撤消（undo）所有这些操作，但它首先会检查要释放的指针是否在映射（也就是那个memMap数组）中。如果不是，则说明要么正在尝试delete不在自由存储区上的地址，要么正在尝试delete已经释放（内存）并且已从映射中移除的地址。

注意，activeFlag变量非常重要，因为在系统关闭期间，我们不想处理任何（内存）释放操作。[[112]](#footnote-112)通过在代码末尾调用MEM\_OFF()，activeFlag将被设置为false，后续的delete调用将被忽略。（在实际的程序中这样做自然是不对的，但当前的目标是查找泄漏，而不是调试库。）最后，为了简化编程，我们将所有数组版本的new和delete操作都转发给了它们的标量版本。

以下程序使用MemCheck机制进行了一次简单的测试：[[113]](#footnote-113)

//: C02:MemTest.cpp

//{L} MemCheck

// 测试MemCheck机制

#include <iostream>

#include <vector>

#include <cstring>

#include "MemCheck.h" // 必须最后包含！

using namespace std;

class Foo {

char\* s;

public:

Foo(const char\* s) {

this->s = new char[strlen(s) + 1];

strcpy(this->s, s);

}

~Foo() { delete [] s; }

};

int main() {

// 可以试验取消下一行注释之后的结果

// TRACE\_OFF();

MEM\_ON();

cout << "你好" << endl;

int\* p = new int;

delete p;

int\* q = new int[3];

delete [] q;

int\* r = nullptr; // 故意创建一个空指针

delete r;

vector<int> v;

v.push\_back(1);

Foo s("再见");

MEM\_OFF();

} ///:~

这个示例验证了在同时存在流、标准容器和通过构造函数来分配内存的类的环境中，我们可以正常地使用 MemCheck。指针p和q的分配和释放没有任何问题，但r不是一个有效的堆指针，因此会提醒“试图delete未知指针”。上述程序的输出结果如下所示：

你好

在地址 00A00538 处分配了 4 字节(文件: MemTest.cpp, 行号: 24)

已释放地址00A00538处的内存

在地址 00A00538 处分配了 12 字节(文件: MemTest.cpp, 行号: 26)

已释放地址00A00538处的内存

试图delete未知指针: 00000000

在地址 00A00548 处分配了 5 字节(文件: MemTest.cpp, 行号: 15)

已释放地址00A00548处的内存

未发生用户内存泄漏!

由于调用了MEM\_OFF()，因此自定义的内存跟踪系统将不再记录之后vector或ostream对operator delete()的调用。然而，容器内部的重新分配操作[[114]](#footnote-114)仍有可能触发delete操作。

如果在程序开始时添加一个TRACE\_OFF()调用，则输出变成：

你好

试图delete未知指针: 00000000

未发生用户内存泄漏!

## 2.4 小结

许多软件工程的问题都可以通过在编码前仔细思考来避免。[[115]](#footnote-115)即使没有习惯性地使用assert()宏，在编写循环和函数时，我们或多或少都会在心里进行一些“断言”。但是，如果将这些“心里的断言”具化为代码中的assert()，那么不仅能更早地发现逻辑错误，还能让代码更易读。但要记住的是，只能将断言用于“不变量”，而不要用于运行时的错误处理。

经过充分测试的代码能让我们更有信心。如果你曾觉得测试是一件麻烦事，那么不妨试试自动化测试框架（例如本章介绍的这个）。将测试集成到日常开发中，不仅能提升代码质量，还能让你和用户都更安心。

## 2.5 练习

*本书配套资源提供了这些练习的答案，请从译者主页（bookzhou.com）或者扫码从清华大学出版社的网盘下载。*

1. 使用TestSuite框架编写一个测试程序，用于对标准vector类以下成员函数进行全面测试。用于测试的对象是一个整数向量[[116]](#footnote-116)：push\_back()（将元素追加到向量末尾）、front()（返回向量中的第一个元素）、back()（返回向量中的最后一个元素）、pop\_back()（移除最后一个元素而不返回它）、at()（返回指定索引位置的元素）以及size()（返回元素数量）。确保验证vector::at()在提供的索引越界时会抛出std::out\_of\_range异常。

2. 假设需要开发一个名为Rational的类来支持有理数（分数）。Rational对象中的分数应该始终以最简形式存储[[117]](#footnote-117)，分母为零则被视为错误。下面是该Rational类的示例接口：

// C02:Rational.h {-xo}

#ifndef RATIONAL\_H

#define RATIONAL\_H

#include <iosfwd>

class Rational {

public:

Rational(int numerator = 0, int denominator = 1);

Rational operator-() const;

friend Rational operator+(const Rational&, const Rational&);

friend Rational operator-(const Rational&, const Rational&);

friend Rational operator\*(const Rational&, const Rational&);

friend Rational operator/(const Rational&, const Rational&);

friend std::ostream& operator<<(std::ostream&, const Rational&);

friend std::istream& operator>>(std::istream&, Rational&);

Rational& operator+=(const Rational&);

Rational& operator-=(const Rational&);

Rational& operator\*=(const Rational&);

Rational& operator/=(const Rational&);

friend bool operator<(const Rational&, const Rational&);

friend bool operator>(const Rational&, const Rational&);

friend bool operator<=(const Rational&, const Rational&);

friend bool operator>=(const Rational&, const Rational&);

friend bool operator==(const Rational&, const Rational&);

friend bool operator!=(const Rational&, const Rational&);

};

#endif // RATIONAL\_H ///:~

请编写该类的完整规范，包括前置条件、后置条件和异常规范。

3. 使用TestSuite框架编写一个测试，对上一个练习中的所有规范进行测试，包括异常测试。

4. 实现Rational类，使其通过上一个练习的所有测试。仅对“不变量”[[118]](#footnote-118)使用断言。

5. 以下BuggedSearch.cpp程序包含一个二叉查找（binary search）函数，用于在范围[beg, end)中查找what。但是，算法中存在一些错误。请使用本章介绍的跟踪技术来调试查找函数。[[119]](#footnote-119)

// C02:BuggedSearch.cpp {-xo}

//{L} ../TestSuite/Test

#include <cstdlib>

#include <ctime>

#include <cassert>

#include <fstream>

#include "../TestSuite/Test.h"

using namespace std;

// 这个函数是唯一有bug的函数

int\* binarySearch(int\* beg, int\* end, int what) {

while (end - beg != 1) {

if (\*beg == what) return beg;

int mid = (end - beg) / 2;

if (what <= beg[mid]) end = beg + mid;

else beg = beg + mid;

}

return 0; // 返回0表示未找到目标值

}

class BinarySearchTest : public TestSuite::Test {

public:

enum { SZ = 10 }; // 把data数组的大小固定为10

int\* data;

int max; // 最大数字

int current; // 当前不包含的数字，在notContained()中使用

// 查找数组中不包含的下一个数字

int notContained() {

while (data[current] + 1 == data[current + 1]) {

++current;

}

if (current >= SZ) return max + 1;

int retValue = data[current++] + 1;

return retValue;

}

void setData() {

data = new int[SZ];

assert(!max); // 在调用setData()函数之前，max应该被初始化为0，遂有此断言

// 创建一个包含递增值的数组。

// 本例会随机决定增量。换言之，每次循环，都会随机决定max是递增1还是递增2。

for(int i = 0; i < SZ; rand() % 2 == 0 ? max += 1 : max += 2) {

data[i++] = max;

}

}

void testInBound() {

// 测试二叉查找函数binarySearch在数组边界内的正确性

// 1. 测试数组中的所有元素：

// \* 从数组末尾开始，逐个元素进行二叉查找

// \* 由于元素一定在数组中，因此查找结果应该为true

for (int i = SZ; --i >= 0;) {

test\_(binarySearch(data, data + SZ, data[i]));

}

// 2. 测试不在数组中的元素：

// \* 循环获取不在数组中的元素

// \* 对这些元素进行二叉查找，结果应该为false

for (int i = notContained(); i < max; i = notContained()) {

test\_(!binarySearch(data, data + SZ, i));

}

}

void testOutBounds() {

// 测试二叉查找函数binarySearch在数组边界外的正确性

// 1. 测试比数组最小值小的值：

// \* 从数组最小值的前一个值开始，递减到最小值减去100

// \* 这些值一定不在数组中，因此查找结果应该为false

for (int i = data[0]; --i > data[0] - 100;) {

test\_(!binarySearch(data, data + SZ, i));

}

// 2. 测试比数组最大值大的值：

// \* 从数组最大值的后一个值开始，递增到最大值加100

// \* 这些值一定不在数组中，因此查找结果应该为false

for (int i = data[SZ - 1]; ++i < data[SZ - 1] + 100;) {

test\_(!binarySearch(data, data + SZ, i));

}

}

public:

BinarySearchTest() {

max = current = 0;

}

void run() {

setData();

testInBound();

testOutBounds();

delete[] data;

}

};

int main() {

srand(time(0));

BinarySearchTest t;

t.run();

return t.report();

}

///:~

1. 译注：能以得体的方式处理异常，就可以说代码是“异常安全”（exception-safe）的。 [↑](#footnote-ref-1)
2. 译注：不变量（invariant）是在算法中某个特定位置始终成立的条件。使用不变量，可以在开始编码之前发现错误，从而减少调试和测试的工作量。 [↑](#footnote-ref-2)
3. 译注：在C语言中，通常要求在函数之后马上执行一个错误检查。这样的紧耦合会导致代码可读性和可维护性降低，尤其是函数调用序列很长时。 [↑](#footnote-ref-3)
4. 译注：如果一个异常在整个程序中都没有被捕获到，该异常就会成为“未处理异常”，会造成应用程序的终止。 [↑](#footnote-ref-4)
5. 所谓“不尊重”对象，是指longjmp()在执行非局部跳转时不遵守C++ 中关于对象生命周期管理的规定，特别是它不会自动调用析构函数来清理那些在跳转出去的作用域内创建的对象。运行某些示例时，你会发现一些C++编译器扩展了longjmp()，能够清理栈上的对象。但是，这种行为会导致程序丧失可移植性，因为不符合C++标准。 [↑](#footnote-ref-5)
6. 译注：这里只是按约定俗成的方式将exception handler翻译为“异常处理程序”，但请把它想象为专门用于处理异常的一个代码块。 [↑](#footnote-ref-6)
7. 译注：也可以将stack unwinding称为“栈辗转开解”。虽然一般将unwind翻译为“展开”，但这并不是一个很好的翻译。wind和unwind均源自生活。将线缠到线圈上称为wind；从线圈上松开则称为unwind。同样地，调用方法时压入栈帧，称为wind；方法执行完毕弹出栈帧，称为unwind。一级一级地向上清理各个栈帧中实例化的对象，就是“栈展开”。 [↑](#footnote-ref-7)
8. BASIC语言长期支持一种有限形式的恢复式异常处理，这是通过其ON ERROR机制来实现的。. [↑](#footnote-ref-8)
9. 在异常处理程序中，最好使用const引用来指定异常对象，因为很少需要修改或重新抛出异常。然而，这并非一项硬性的规定。 [↑](#footnote-ref-9)
10. 译注：在这个例子中，注意异常处理程序捕获的是引用而非对象，这样可以保留派生类信息，避免类型截断，使得我们可以在异常处理程序中访问派生类对象的所有成员。除此之外，还避免了不必要的对象拷贝操作。 [↑](#footnote-ref-10)
11. 只有明确（无歧义）且可访问的基类才能捕获派生类型的异常。 这个规则有助于最小化验证异常所需的运行时开销。记住，异常是在运行时检查的，而不是在编译时。因此，在异常处理期间，许多在编译时可用的信息都是不可用的。 [↑](#footnote-ref-11)
12. 译注：换言之，问题修复不了，程序是肯定无法继续运行的。 [↑](#footnote-ref-12)
13. 译注：作者的意思是，这里在声明old\_terminate函数指针的同时，还用set\_terminate()函数的返回值进行了初始化。如果只是希望声明一个函数指针，但不初始化它，那么可以使用void (\*myFunctionPointer)();这样的形式。这正是作者所说的“在函数指针声明后面直接看到一个分号”的意思。 [↑](#footnote-ref-13)
14. 译注：从C++11开始，建议使用智能指针来管理动态分配的内存。智能指针遵循RAII（Resource Acquisition Is Initialization，资源获取即初始化）的原则，即资源的获取和释放与对象的生命周期绑定，从而避免了手动管理资源的需要。RAII的详情请参见1.5.2节。 [↑](#footnote-ref-14)
15. 译注：祼指针是相对于智能指针而言的。祼指针要求程序员手动管理分配的内存。就本例而言，它创建的所有指针都是祼指针。指针是没有析构函数的，所以若在构造函数中抛出异常，那么这些资源不会被释放。这就是“祼”指针的来历，它相当于在“裸奔”。 [↑](#footnote-ref-15)
16. 译注：原子化确保只要在对象生命周期内，就必然会完成资源的分配和释放。 [↑](#footnote-ref-16)
17. 和C#语言不同，C++语言不支持“空引用”。C#语言虽然支持空引用，但要求开发者执行“空检查”，以避免运行时错误。两种语言在这方面的选择可谓“见仁见智”，各有各的优缺点。支持“空引用”，会为开发者带来更大的灵活性。不支持“空引用”，引用则总是绑定到一个具体的对象，这简化了编译器的优化，能获得更好的性能。另外要注意的是，空引用不等于空指针。C++语言是允许空指针的，在C++11前是用值0或NULL宏来表示，从C++11开始则可以用nullptr来表示。。 [↑](#footnote-ref-17)
18. 注意，C++标准库专门提供了一个异常类（名为std::out\_of\_range）来处理这种情况。 [↑](#footnote-ref-18)
19. 译注：auto\_ptr是C++早期版本提供的一种智能指针，用于管理动态分配的内存。它试图通过RAII机制来实现自动内存释放。然而，由于其设计缺陷，在C++11中已被弃用。C++11及以后的版本推荐使用unique\_ptr和shared\_ptr来代替auto\_ptr。unique\_ptr表示独占所有权，不能被复制，指针销毁，指向的对象也会被释放；shared\_ptr 表示共享所有权，可以被多个指针共享，最后一个指针销毁，指向的对象才会被释放。 [↑](#footnote-ref-19)
20. 译注：由于auto\_ptr已弃用，本书中文版配套资源提供了该程序的C++20版本，文件名：Auto\_ptr\_CPP20.cpp。它使用unique\_ptr进行了重写。 [↑](#footnote-ref-20)
21. 译注：“按值包含”意味着该成员变量直接存储了类对象的副本。 [↑](#footnote-ref-21)
22. 有关auto\_ptr的详细信息，请参考赫伯·萨特（Herb Sutter）1999年10月在*C/C++Users Journal*期刊上发表的一篇文章，标题为“Using auto\_ptr Effectively”，第63～67页，目前可通过该网址访问这篇文章：http://www.gotw.ca/publications/using\_auto\_ptr\_effectively.htm [↑](#footnote-ref-22)
23. 译注：基类子对象是一个不常见的说法。实例化派生类对象时，派生类对象包含了基类的一个“子对象”。因此，会先调用基类的构造函数来初始化“基类子对象”，再初始化派生类新增的成员。 [↑](#footnote-ref-23)
24. 译注：微软实现的std::exception有一些非标准的构造函数，能直接获取std::string参数。 [↑](#footnote-ref-24)
25. 译注：也就是说，它没有像logic\_error和runtime\_error那样进一步细分为更多的子类来表示不同类型的I/O错误。 [↑](#footnote-ref-25)
26. 译注：例如，sizeof操作符返回的就是size\_t类型的一个值，即一个位宽不小于16位的无符号整型（32位系统上是32位，64位系统上是64位）。但要注意的是，sizeof 用于计算对象或类型的字节大小，是静态的；而npos 表示当前容器（例如std::string或std::vector）中最大的有效索引值加1，是动态的。npos 的类型通常是 size\_t。npos的意思是“不存在的位置”或者“空位置”。 [↑](#footnote-ref-26)
27. 译注：动态异常规范（即throw(*optional\_type\_list*)）已在 C++11 中弃用，并在 C++17 中删除。详见稍后的例子。 [↑](#footnote-ref-27)
28. 译注：在C++11及以后， noexcept关键字逐渐取代了throw()，因为它更简洁、更安全。 [↑](#footnote-ref-28)
29. 译注：都说了这个函数不抛出任何异常，那么它最好就是真的不抛出任何异常！ [↑](#footnote-ref-29)
30. 译注：这个例子使用的动态异常规范（即throw(*optional\_type\_list*)）已在 C++11 中弃用，并在 C++17 中删除。因此，在支持C++20的现代C++编译器中，这个例子不会调用由set\_unexpected()指定的用于处理“未处理异常”的函数。本书中文版配套资源提供了该程序的C++20版本，文件名：Unexpected\_CPP20.cpp。它更好地演示了set\_unexpected()函数。 [↑](#footnote-ref-30)
31. 译注：有别于exit()，abort()会立即终止程序，而不做任何资源清理工作。不过，abort()通常会向操作系统发送一个 SIGABRT 信号，这可能导致生成核心转储（core dump），以便将来进行调试。 [↑](#footnote-ref-31)
32. 译注：如前所述，现代C++已摒弃了动态异常规范（即throw(*optional\_type\_list*) ）。因此，这个示例代码并不能在现代C++编译器中生成后文所描述的结果。现代 C++ 提倡使用更简单的异常处理方式。通过使用 noexcept 关键字，避免滥用std::bad\_exception以及简化代码逻辑，可以写出更清晰、更健壮的 C++ 代码。 [↑](#footnote-ref-32)
33. 译注：作者的梦想成真了！在从C++11开始的现代C++中，如果想允许一个函数抛出任何类型的异常，那么直接写void f();就好，不必做任何特殊的声明。 编译器会默认允许函数抛出任何异常。 [↑](#footnote-ref-33)
34. 译注：如前所述，现在若想表示函数不抛出任何异常，那么应该使用noexcept关键字，例如void f() noexcept;。 [↑](#footnote-ref-34)
35. 译注：形参（parameter）是声明时使用的参数占位符，而实参（argument）是调用时实际传递的值。本书一般情况下不区分两者，都说成是“参数”。读者可根据上下文自行区分。但是，在必要的时候，例如两者同时出现的时候，还是会进行区分。 [↑](#footnote-ref-35)
36. 译注：例如，ChatGPT属于（is-a）一种大语言模型（LLM）。它有自己的特点，但它首先是一种LLM，继承了LLM的全部特点（例如，基于Transformer架构）。其他属于LLM的还有Qwen等。 [↑](#footnote-ref-36)
37. 译注：即使派生类抛出的异常更少或没有异常，用户在使用该派生类时也不需要修改自己的代码。他们仍然可以像使用基类一样使用派生类。 [↑](#footnote-ref-37)
38. 译注：在C++98/C++03中，异常规范是强制性的，并且存在严格的协变规则。如果基类中的虚函数有一个特定的异常规范，则派生类中的重写函数必须具有相同或更宽泛的异常规范。从C++11开始，旧的异常规范已经被noexcept关键字取代。到了C++17，异常规范语法throw(*optional\_type\_list*)甚至完全移除了。总之，在现代C++语言中，完全不需要担心传统意义上的异常规范协变问题，只需要关注函数是否可能抛出异常，并使用noexcept关键字来指定这一点即可。 [↑](#footnote-ref-38)
39. 译注：关于协变、逆变和不变，看看这里的总结就好：协变（covariance）：若类型A为协变量，则需要使用类型A的地方可以使用A的某个子类类型。逆变（contravariance）：若类型A为逆变量，则需要使用类型A的地方可以使用A的某个基类类型。不变（invariance）：若类型A为不变量，则需要使用类型A的地方只能使用A类型。 [↑](#footnote-ref-39)
40. 译注：虚函数是实现多态性的基础。如果子类可以任意修改函数签名，那么动态绑定机制就无法正常工作，多态性也就无从谈起。 [↑](#footnote-ref-40)
41. 译注：试图从空栈中pop一个元素，即会发生栈的“下溢”。从C++11开始，可以用nullptr检查空指针。 [↑](#footnote-ref-41)
42. 译注：也是自C++11后摒弃“异常规范”的原因。 [↑](#footnote-ref-42)
43. 译注：专业的说法是产生了“副作用”。 [↑](#footnote-ref-43)
44. 译注：每个模块都只应执行一个明确定义的任务；也就是说，它应该高度**内聚**（cohesive）。这个词有点拗口，但和其他许多面向对象的原则一样，**内聚性**（cohesion）可以用人类世界的术语来描述。一个内聚性低的人会有太多的头绪（too many irons in the fire）。这样的人倾向于在他们需要完成的所有事情中陷入困境，而且基本上所有事情都做不好。事实上，他们完全可以通过将一些责任委托给其他人来变得更“内聚”。——摘自《C++数据结构与算法(第8版)》，清华大学出版社2024年出版。 [↑](#footnote-ref-44)
45. 译注：本书中文版的配套资源提供了该程序的C++20版本，文件名：SafeAssign\_CPP20.cpp。它使用智能指针等现代C++语言特性进行了重写。 [↑](#footnote-ref-45)
46. 如果想更深入地了解异常安全问题，可以参考赫伯·萨特（Herb Sutter）的权威参考书《Exceptional C++（中文版）》，电子工业出版社，2012年出版。 [↑](#footnote-ref-46)
47. 库函数uncaught\_exception()在栈展开的过程中返回true，因此从技术上说，可以通过测试uncaught\_exception()是否为false，从而让一个异常从析构函数中逃逸（即向上传播）。但我们从未见过一个实例可以证明这是一种好的设计，所以这里也只是稍微提一下而已。 [↑](#footnote-ref-47)
48. 译注：异常在程序的运行时栈上依赖于动态的函数调用链。当一个异常被抛出时，它会沿着调用栈往上寻找合适的处理程序，直到找到一个匹配的catch块为止。这个过程是动态的，因为它依赖于抛出异常时的实际调用栈状态。 [↑](#footnote-ref-48)
49. 译注：例如，UI线程。 [↑](#footnote-ref-49)
50. 译注：注意区分同步（synchronous）和异步（asynchronous）。同步意味着一个操作开始后必须等待它完成；异步则意味着不用等它完成，可立即返回做其他事情。不要把编程中的“同步”理解为“同时”。 [↑](#footnote-ref-50)
51. 有的编译器确实会在这些情况下抛出异常，但它们通常都提供了一个编译器选项来禁用这种（非同寻常）的行为。 [↑](#footnote-ref-51)
52. 译注：开发供他人使用的库时，需要特别注意异常处理的设计，因为无法预测最终用户会如何处理库函数引发的关键错误。 [↑](#footnote-ref-52)
53. 译注：意味着对异常进行一些临时处理，让系统能够继续运行，但并没有真正解决问题。可能是记录日志、设置默认值、跳过某些操作等。适用于那些无法立即修复，或者修复成本过高的情况。 [↑](#footnote-ref-53)
54. 译注：如前所述，在现代C++中，异常规范以及unexpected()函数的使用频率已经显著降低。特别是，异常规范已在C++11中废弃。虽然unexpected()函数虽然仍存在，但其应用场景也变得越来越少。之所以这些旧式的异常处理机制逐渐被淘汰，主要是因为现代C++提供了更强大、灵活且安全的异常处理方式。例如，智能指针的广泛应用有效地避免了资源泄漏问题，RAII（资源获取即初始化）技术使得资源管理更加安全可靠。 [↑](#footnote-ref-54)
55. 译注：通常将该标志变量命名为error 或 valid，若构造失败，就将其设为false。 [↑](#footnote-ref-55)
56. 译注：如前所述，auto\_ptr从C++11开始已经弃用了。现代C++要求使用unique\_ptr和shared\_ptr等智能指针。 [↑](#footnote-ref-56)
57. 译注：“调用函数”（calling function）是指发出调用的函数。这里说的其实是调用函数中发出调用的那个指令的位置。可以理解为被调用函数的“返回地址”。 [↑](#footnote-ref-57)
58. 译注：该指针指向的其实是动态父函数的“栈帧”，即调用函数的栈帧的起始位置。 [↑](#footnote-ref-58)
59. 这具体要取决于如果不使用异常，你需要插入多少返回码检查代码。 [↑](#footnote-ref-59)
60. Borland默认启用异常；要禁用异常，请使用-x-编译器选项。Microsoft默认禁用异常支持；要启用它，请使用-GX选项。注意，在使用这两个编译器时，都用-c选项执行“仅编译”。 [↑](#footnote-ref-60)
61. 译注：这是个很有意思的例子。虽然Borland C++早已停止开发（最后一个版本停留在5.5），但在现代C++中，我们仍然可以用GCC和Microsoft C++编译器来重现这个例子。要注意的是，-GX选项已在当前的Microsoft C++中禁用，应改为使用-EHsc选项来启用异常支持。GCC则默认启用异常支持，要禁用它，请使用-fno-exceptions选项。同样，在使用这两个编译器时，都用-c选项执行“仅编译”。 [↑](#footnote-ref-61)
62. 感谢斯科特·梅耶斯（Scott Meyers）和约瑟·拉乔伊（Josee Lajoie）对零成本模型的见解。要更深入地了解异常的工作原理，可以参考约瑟写的这篇好文章：*Exception Handling: Behind the Scenes*（异常处理：幕后），该文章收录于*C++ Gems*一书（SIGS，1996年）。 [↑](#footnote-ref-62)
63. 译注：C++98以前。 [↑](#footnote-ref-63)
64. 译注：例如，void func() throw (MyException) { throw MyException("示例异常");}。当然，现代C++不会用这个法子。 [↑](#footnote-ref-64)
65. 译注：注意，这里是重载（overload）而不是重写（override）。如果重载了new，就应该在同一作用域内重载delete，以避免出现不容易发现的动态内存管理错误。重载new和delete通常是出于性能的考虑来精确控制内存的分配和解除分配。例如，可以用这个技术预分配一个内存池（使用placement new操作符），以后就在该内存池中创建新对象，从而减少运行时内存分配的开销。——摘自《学习C++20（中文版）》，清华大学出版社2023年出版。 [↑](#footnote-ref-65)
66. 译注：注意区分健壮性(鲁棒性)和可靠性，两者对应的英文单词分别是 robustness 和 reliability。健壮性主要描述一个系统对于参数变化的不敏感性，而可靠性主要描述一个系统的正确性，也就是在提供一个固定的参数时，它应该产生稳定的、能预测的输出。例如一个程序，它的设计目标是获取一个参数并输出一个值。假如它能正确完成这个设计目标，就说它是可靠的。但在这个程序执行完毕后，假如没有正确释放内存，或者说系统没有自动帮它释放占用的资源，就认为这个程序（或者它的“运行时”）不具备健壮性或者鲁棒性。——摘自《深入CLR》，清华大学出版社2024年出版。 [↑](#footnote-ref-66)
67. 译注：也可以将不变量称为不变式。 [↑](#footnote-ref-67)
68. 译注：称为open slot。 [↑](#footnote-ref-68)
69. 译注：所谓“严格大于nextSlot”，其实就是指至少要比当前使用的slot数量多至少一个。因为这是一个“严格”的条件，所以适合立即“断言”这个条件的成立。 [↑](#footnote-ref-69)
70. 译注：作者反复强调了某些编程方式的“可移植性”。这指的是一种代码的特性，即该代码可以在不同的硬件平台、操作系统和编译器上编译运行，而无需或仅需少量修改。换句话说，就是代码的跨平台兼容性。 [↑](#footnote-ref-70)
71. 译注：注意，这里展示的只是定义。实际调用时（参见前一个例子），需要为assert()调用附加分号以构成语句。若定义了NDEBUG宏，那么assert()宏会展开为一个空语句。而因为添加了分号，所以编译器仍然会把它视为一个有效的语句，并继续编译。 [↑](#footnote-ref-71)
72. 东尼·霍尔是一位英国计算机科学家，他最著名的贡献就是发明了快速排序算法（Quicksort）。 [↑](#footnote-ref-72)
73. 如麦克·L·史考特（Michael L. Scott）在其2000年出版的*Programming Language Pragmatics*一书中所述（Morgan Kaufmann出版社）。 [↑](#footnote-ref-73)
74. 请参见他的*Programming Language Pragmatics*一书，Morgan–Kaufmann出版社于2000年出版。 [↑](#footnote-ref-74)
75. 译注：在计算机编程中，如果一个函数/方法或表达式除了生成一个值，还会造成状态的改变（例如，向文件中写入或者创建数据库连接等），就说它会造成副作用；或者说会执行一个副作用。 [↑](#footnote-ref-75)
76. 译注：要想深入了解前置条件和后置条件，以及更多关于安全C++编程的主题，强烈推荐阅读《C++数据结构与算法(第8版)》一书，清华大学出版社2024年出版。 [↑](#footnote-ref-76)
77. 虽然从概念上来说，这仍然是一种断言，但由于我们不想中断程序的执行，因此assert()宏并不合适。例如，在Java 1.4中，当断言失败时会抛出一个异常。（译注：事实上，在Java的最新版本中，当断言失败时，仍然会抛出一个AssertionError异常。） [↑](#footnote-ref-77)
78. 译注：也称为“类不变式”。 [↑](#footnote-ref-78)
79. 译注：换句话说，子类必须满足基类的所有前置条件，并且满足至少与基类相同的后置条件。 [↑](#footnote-ref-79)
80. 译注：“更强”的意思是子类除了满足基类的所有后置条件外，还可以满足一些额外的条件。 [↑](#footnote-ref-80)
81. 译注：后置条件断言是一种有用的工具，可以帮助验证函数的输出是否符合预期。然而，它不能完全替代单元测试。为了编写高质量的代码，应该结合单元测试和后置条件断言，以确保代码的正确性和可靠性。单元测试应该涵盖各种输入情况、边界条件和异常情况。 [↑](#footnote-ref-81)
82. 本节基于Chuck的“The Simplest Automated Unit Test Framework That Could Possibly Work”（最简单的自动化单元测试框架）一文，发表于*C/C++ Users Journal*，2000年9月。 [↑](#footnote-ref-82)
83. 译注：本书是在《敏捷宣言》于2001年发布后写作的，因此引入了敏捷软件开发的思想。推荐读者参考清华大学出版社出版的以下书籍，深入了解敏捷软件开发的最新进展：《非凡敏捷：整合敏捷转型框架原理、实践及五项修炼》、《系统分析与设计（第9版）》以及《高质量需求：聚焦于商业价值的20个核心实践》。 [↑](#footnote-ref-83)
84. 关于重构，强烈推荐阅读马丁·福勒（Martin Fowler ）所著的《重构：改善既有代码的设计（第2版）》。 也可以直接访问网站http://www.refactoring.com。重构是极限编程（XP）的一项重要实践。 [↑](#footnote-ref-84)
85. 有关极限编程的详细信息，请参阅肯特·贝（Kent Beck）所著的*Extreme Programming Explained: Embrace Change*（解析极限编程：拥抱变化）一书（Addison Wesley，1999 年出版）。包括极限编程（XP）在内这样的轻量级方法论已经联合起来，共同组建了**敏捷联盟**。详情请访问https://www.agilealliance.org/。 [↑](#footnote-ref-85)
86. 译注：这正是“测试驱动开发”（test-driven development，TDD）的精髓。“先写测试，后写代码”是TDD的核心思想。 [↑](#footnote-ref-86)
87. 译注：例如，在编写一个计算器程序时，可以先编写测试用例来验证计算器在处理0、负数、除以0等边界情况下的表现。 [↑](#footnote-ref-87)
88. 译注：换言之，很可能是构成软件系统的其他组件出了问题。 [↑](#footnote-ref-88)
89. 译注：将本书中文版配套资源中的Date.h、Date.cpp和SimpleDateTest.cpp放到同一个目录。然后，如果使用Microsoft C++编译器，那么执行以下编译命令：cl /EHsc SimpleDateTest.cpp Date.cpp。如果使用GCC，那么执行以下编译命令：g++ -o Date.cpp SimpleDateTest.cpp -Wno-deprecated。注意，最后一个选项是为了屏蔽关于C++11已弃用 “异常规范”（参见第1章）的警告。 [↑](#footnote-ref-89)
90. 译注：在命令行上执行cl /EHsc SimpleDateTest2.cpp Date.cpp或g++ -o SimpleDateTest2 date.cpp SimpleDateTest2.cpp -Wno-deprecated。 [↑](#footnote-ref-90)
91. 我们的Date类还实现了“国际化”，能够支持宽字符集。这将在第3章末尾介绍。 [↑](#footnote-ref-91)
92. 欲知详情，请访问http://sourceforge.net/projects/cppunit。 [↑](#footnote-ref-92)
93. 这是极限编程（XP）的关键原则之一。 [↑](#footnote-ref-93)
94. 在“第2章”目录中执行以下编译命令：  
    cl /EHsc DateTest.cpp Date.cpp ../TestSuite/Test.cpp [↑](#footnote-ref-94)
95. RTTI的全称是Runtime Type Identification，即“运行时类型识别”，将在第 9 章讨论。具体地说，我们会使用typeinfo类的name()成员函数。如果使用Microsoft Visual C++，那么需要指定编译选项/GR。如果不指定，那么可能在运行时报告访问违例。（译注：现已不需要这个选项。） [↑](#footnote-ref-95)
96. 注意，这里使用了字符串化技术（即stringizing，它允许使用#预处理操作符将宏参数转换为一个字符串字面值）以及预定义宏\_\_FILE\_\_和\_\_LINE\_\_。欲知详情，请参阅本章稍后的代码。 [↑](#footnote-ref-96)
97. 这种情况非常适合使用批处理文件和shell脚本。Suite类是对相关测试进行组织的一种基于C++的方法。 [↑](#footnote-ref-97)
98. 译注：在“第3章”目录中执行以下编译命令：  
    cl /EHsc StringSuite.cpp ../TestSuite/Suite.cpp ../TestSuite/Test.cpp TrimTest.cpp [↑](#footnote-ref-98)
99. 译注：这正是在编译时要链接为TrimTest生成的目标文件的原因。记住，静态数据虽然属于类，但它的定义必须放在类之外，并且只能定义一次。如果在多个文件中定义，会引起链接错误。 [↑](#footnote-ref-99)
100. 译注：之前的“译注”解释了如何在未使用环境变量包含库搜索路径的情况下何执行编译命令。 [↑](#footnote-ref-100)
101. 译注：如果基类的析构函数不是虚函数，那么当通过基类指针delete一个派生类对象时，只会调用基类的析构函数，不会调用派生类的析构函数。这会导致派生类中分配的资源无法被正确释放，从而造成内存泄漏。 [↑](#footnote-ref-101)
102. 译注：作者说的那两行代码是std::ofstream TRACEFILE\_\_("TRACE.OUT");和#define cout TRACEFILE\_\_。前者创建一个ofstream对象TRACEFILE\_\_，并将它与文件TRACE.OUT关联，用于写入跟踪信息。后者将cout重定义为TRACEFILE\_\_。这意味着，原本应该输出到控制台的任何内容，现在都会被写入到TRACE.OUT文件中。可以在当前.cpp中直接添加这两行代码而不必写#define TRACEON和#include "Trace.h"。 [↑](#footnote-ref-102)
103. 译注：执行cl /EHsc Tracetst.cpp，然后查看生成的TRACE.OUT文件的内容。 [↑](#footnote-ref-103)
104. 译注：直接使用指针访问数组时，由于绕过了模板类的封装，所以模板类的边界检查机制就失效了。 [↑](#footnote-ref-104)
105. 译注：本书中文版几乎混用了delete和deallocate，都说成“释放”。事实上，前者通常是指delete一个指针，而后者通常是指释放动态分配的内存。 [↑](#footnote-ref-105)
106. 我们的关键技术审阅者，Dinkumware. Ltd.的Pete Becker，提醒我们使用宏替换C++关键字是非法的。他评价这种技巧：“这是一种偏门的手段。但是，有时为了弄明白代码为什么不起作用，这种手段是必要的。因此，可以把它放到自己的工具箱中，只是不要在正式发布的代码中使用它。”程序员请务必注意这一点。 [↑](#footnote-ref-106)
107. 感谢C++标准委员会的Reg Charney向我们推荐这个技巧。 [↑](#footnote-ref-107)
108. 译注：例如，cout和cin这些都是iostreams。 [↑](#footnote-ref-108)
109. 译注：作者的意思是说，即使选择使用C标准 I/O 来实现内存跟踪，也不会影响程序中其他部分使用iostreams进行输入输出。 [↑](#footnote-ref-109)
110. 译注：该静态对象在程序启动时创建，并且在程序结束时才被销毁。它的生命周期与整个程序的生命周期相同。 [↑](#footnote-ref-110)
111. 译注：程序都终止了，还有指针没有delete？换言之，还有动态分配的内存没有释放？ [↑](#footnote-ref-111)
112. 译注：标准库、操作系统、第三方库也可能在内部进行内存分配和释放。因此，要避免和它们的冲突。 [↑](#footnote-ref-112)
113. 译注：编译命令是cl /EHsc MemTest.cpp MemCheck.cpp。 [↑](#footnote-ref-113)
114. 译注：例如，vector的扩容。 [↑](#footnote-ref-114)
115. 译注：不要直接上手编码，而是应该“先思考，再编码”。以下建议帮助你更好地从软件工程的角度思考：学习软件工程基础知识；参与实际项目；学习优秀的设计模式；多与同行（同学）交流。当然，最后是多读一读像《C++编程思想》这样有“思想”的书。 [↑](#footnote-ref-115)
116. 译注：例如，一个std::vector<int> v = {1,2,3,4,5};。 [↑](#footnote-ref-116)
117. 译注：假设定义了一个Rational类，其成员变量为numerator（分子）和denominator（分母）。当我们创建一个表示分数4/6的Rational对象时，在构造函数中，我们应该先将4/6约分为2/3，然后将分子设置为2，分母设置为3。具体请参见本书配套资源的“答案”部分的Rational.cpp文件。 [↑](#footnote-ref-117)
118. 译注：本章开头解释了什么是“不变量”或“不变式”。 [↑](#footnote-ref-118)
119. 译注：编译命令是cl /EHsc BuggedSearch.cpp ../TestSuite/Test.cpp。 [↑](#footnote-ref-119)