《C++编程思想》重译版

[美] Bruce Eckel & Chuck Allison 著

周靖 译（https://bookzhou.com）

中文试读版 1-4 章，翻译原稿，仅供参考，

配套资源和试读下载： ys168 网盘>> 百度网盘>> GitHub 项目>>

访问中文版主页，获取最新资讯

清华大学出版社

北 京

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具有良好行为的软件，还会介绍一些设计原则来指导您编写“异常安全”的[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]。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，但实际上鲜有人会去检查这个值。人们不喜欢代码的增多，更不喜欢因此而来的代码可读性的下降。

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，这会导致代码变得复杂、难以维护且容易引入新的错误。[③]

异常处理是 C++语言的重要特性之一，这是一种更好的思考和处理错误的方式。异常处理有以下好处：

[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。

[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。

[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。

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. 再也无法忽视错误。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，直至错误被捕获，或者程序因为没有相应的异常处理程序而终止。[①]

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）。这些函数提供了一种方便的方式来表达这样的意思：“这里有一些问题，可能需要用更复杂的代码来处理它。但是，在本示例中，暂时无需为此分心。”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，那么错误处理可以变得相当简单直接，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，

[①] 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”，造成应用程序终止。

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，尤为严重的是当它跳出作用域时不会自动调用析构函数。[①]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，那么程序的行为会变得“未定义”。

1.2 抛出异常

如果在代码中遇到了异常情况（也就是说，当前上下文中没有足够的信息来帮助自己决定下一步该怎么做），那么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**，如下例所示。

```
//: 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");
```

[①] 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。

```
}

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*作为构造函数的实参。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。我们可以将异常处理简单地想象成一种替代返回机制（但深入推敲的话就会发现问题）。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**[①]）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**[②]。

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，方便调用者看明白如何处理异常。

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

---

① 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。

② 译注：也可以将 stack unwinding 翻译为“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说法。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”。

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(typeN idN)
    // 处理 typeN 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;

// 该类代表奥兹魔法世界
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

异常处理机制有两种基本的模型：**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可挽回，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。[①]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。[②]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。[③]

```
//: 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>
```

[①] 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。.

[②] 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。

[③] 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（即丢失部分类型信息），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。

```
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;
    }
} ///:~
```

您可能以为，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。

下例展示了如何使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[1]），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（handler）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。（详情请参见稍后的 1.5.1 节“资源管理”）。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。

---

[1] 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。

```
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[1]，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。尽管您可能习惯于在函数指针声明后面直接看到一

[1] 译注：换言之，问题修复不了，程序是肯定无法继续运行的。

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。[①]

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。[②]

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;
```

[①] 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。

[②] 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。

```
        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。[①]

```
//: 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;
    }
```

[①] 译注：裸指针是相对于智能指针而言的。裸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裸指针。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“裸”指针的来历，它相当于在“裸奔”。

```
    ~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所示：

```
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。[1]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

[1] 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。

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，您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[1]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）。[2]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[3]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。

---

[1] 译注：本书将 operator 翻译为“操作符”，而不是“运算符”。同理，operand 翻译为“操作数”，而不是“运算子”。

[2] 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。。

[3] 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。

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。[①]下面的例子展示了它是如何工作的。[②]

```
//: 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:
```

[①] 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。

[②] 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。

```
    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我们的类型的指针。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[①]，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。[②]

---

① 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。

② 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

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[①]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[②]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);
```

[①] 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。

[②] 译注：关于构造函数的 initializer，本书可能会混用“初始化器”和“初始化列表”这两种说法。在英语语境中，也经常会混用 initializer 和 initializer list。

```
    } 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。[①]

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[②]。

[①] 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。

[②] 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。

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[①]
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] 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”。

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：[①]

```
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。[②]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！[③]

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

[①] 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。

[②] 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。

[③] 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！

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。[①]

```
//: 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) {
```

[①] 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。

```
            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）。在您自己的 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()[①]。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] 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。

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：[①]

```
//: 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 {
```

[①] 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。

```
        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。[①]如果想指定 f 不抛出任何异常，请使用空列表，如下所示：[②]

```
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）[③]，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[④]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[⑤]。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：[⑥]

---

① 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。

② 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;`。

③ 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。

④ 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。

⑤ 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。

⑥ 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。

```
//: 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）[①]，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

[①] 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。

就是说，不允许更改重写的虚函数的签名。[①]

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）[②]，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。[③]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）。届时会讲到，其 pop 成员函数是这样声明的：

```
void pop();
```

注意，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：

[①] 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。

[②] 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。

[③] 译注：也是自 C++11 后摒弃“异常规范”的原因。

```
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[①]。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。[②]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。[③]

```
//: C01:SafeAssign.cpp
```

[①] 译注：专业的说法是产生了“副作用”。

[②] 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。

[③] 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。

```
// 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。[①]

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。）[②]

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

[①] 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。

[②] 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。

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”），[①]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[②]代码中同步[③]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。[④]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，因此有些人可能会觉得能用异常来代替这些常规的语言机制。但千万不要有这种“冲动”，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，否则会让使用我们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

[①] 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。

[②] 译注：例如，UI 线程。

[③] 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”。

[④] 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。

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，我们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，后跟一个`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做到最少。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，特别是当自己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以及标准 C++库为什么没有使用异常规范的讨论）。[①]

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[②]，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）。

[①] 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。

[②] 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。

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。[①]

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，我们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，而无需关心我们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

[①] 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。

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；[①]
- 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，

[①] 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`。

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。[①]

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，无论您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）[②]，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

[①] 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。

[②] 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”。

量），以及指向调用它的函数（即它的动态父函数）的指针[①]。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。[②]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：[③]

```
//: 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。[④]

[①] 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。

[②] 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。

[③] 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”。

[④] 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 VC++编译器来重现这个例子。要注意的是，-GX 选项已在当前的 Microsoft VC++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”。

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[①]，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[②]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] 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）。

[②] 译注：C++98 以前。

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[①]）。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。[②]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，其异常规范支持抛出四种异常类型：一个 char、一个 int、一个 bool 和您自己的异常类。在 main()中捕获每一个并验证捕获。该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

[①] 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。

[②] 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。

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[①]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”：尽一切努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。[②]

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）[③]，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
```

---

① 译注：说成“防御式编程”也不无不可。

② 译注：注意区分健壮性（鲁棒性地）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。

③ 译注：也可以将不变量称为不变式。

```
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[①]。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。[②]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。[③]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)
```

[①] 译注：称为 open slot。

[②] 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。

[③] 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。

```
#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。[①]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 VC++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）[②]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。[③]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

[①] 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。

[②] 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）。

[③] 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）。

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。[①]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[②]而交付的结果。[③]

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）。[④]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：

[①] 请参见他的 *Programming Language Pragmatics* 一书，Morgan–Kaufmann 出版社于 2000 年出版。

[②] 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。

[③] 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。

[④] 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。）

```
assert(nextSlot <= capacity);
```

这种不变量被称为**类不变量**[①]，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。[②]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[③]，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。[④]

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。[⑤]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

---

① 译注：也称为“类不变式”。

② 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。

③ 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。

④ 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。

⑤ 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。

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！[1]

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。[2]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）[3]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] 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》。

[2] 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。

[3] 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/。

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。[①]

“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[②]），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。[③]

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

[①] 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。

[②] 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。

[③] 译注：换言之，很可能是构成软件系统的其他组件出了问题。

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：[①]

```
//: 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。[②]

[①] 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 VC++编译器，那么执行以下编译命令：cl /EHsc SimpleDateTest.cpp Date.cpp。如果使用 GCC，则执行以下编译命令：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。

[②] 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。

```
//: 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，本例没有对更长的持续时间（duration()的返回值）进

行测试。由于篇幅所限，所以这里不再赘述。但是，想必您已经大致理解了。注意，Date 类的完整实现请参见附录 B 中的 Date.h 和 Date.cpp 文件。[①]

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。[②]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）[③]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);
```

[①] 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。

[②] 欲知详情，请访问 http://sourceforge.net/projects/cppunit。

[③] 这是极限编程（XP）的关键原则之一。

```
        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。[①]

```
//: 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
```

[①] 在“第 2 章”目录中执行以下编译命令：

```
cl /EHsc DateTest.cpp Date.cpp ../TestSuite/Test
```

```
*/ ///:~
```

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[①]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。[②]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_();
    }
```

[①] 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。）

[②] 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。

```
        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。[①]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。[②]

```
//: 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);
```

[①] 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。

[②] 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
```

```
    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。[1]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

[1] 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。

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。[1]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
```

[1] 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。

```
    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。[①]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()

[①] 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。

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[①]）。

```
//: 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：[②]

```
//: C03:Tracetst.cpp {-bor}
#include <iostream>
#include <fstream>
#include "../require.h"
using namespace std;

#define TRACEON
#include "Trace.h"

int main() {
```

[①] 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"。

[②] 译注：执行 cl /EHsc Tracetst.cpp，运行程序，然后查看生成的 TRACE.OUT 文件的内容。

```
    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。[①]）
- 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）[②]、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。[③]

[①] 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。

[②] 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。

[③] 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。

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。[1]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++

[1] 感谢 C++标准委员会的 Reg Charney 向我们推荐这个技巧。

iostreams[1]。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。[2] 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);
```

[1] 译注：例如，cout 和 cin 这些都是 iostreams。

[2] 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。

```
        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。[①]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）。[②]

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

[①] 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。

[②] 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？

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。[①]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：[②]

```
//: 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(原书代码无= nullptr 部分)
    delete r;
```

[①] 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。

[②] 译注：编译命令是 cl /EHsc MemTest.cpp MemCheck.cpp。

```
    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[①]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。[②]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

[①] 译注：例如，vector 的扩容。

[②] 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。

误处理。

经过充分测试的代码能让我们更有信心。如果您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[①]：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[②]，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&);
```

[①] 译注：例如，一个 std::vector<int> v = {1,2,3,4,5};。

[②] 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。

```
    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”[1]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。[2]

```
//: 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
}
```

[1] 译注：本章开头解释了什么是“不变量”或“不变式”。

[2] 译注：编译命令是 cl /EHsc BuggedSearch.cpp ../TestSuite/Test.cpp。

```
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();
} ///:~
```

第 II 部分 标准 C++库

标准 C++不仅集成了所有标准 C 库（并进行了少量增改以支持类型安全性），还新增了它自己的库。这些库比标准 C 中的库强大得多；我们从这些库获得的帮助与提升，就像从 C 升级到 C++所带来的帮助与提升一样。

本书这一部分将深入介绍标准 C++库的关键部分。

整个库最权威（但也最晦涩）的参考资料就是“标准”本身。本贾尼·斯特劳斯特鲁普（Bjarne Stroustrup ）的《C++程序设计语言》仍然是语言和库的可靠参考书。如果需要一本专注于库的参考书，那么尼古拉·约祖蒂斯（Nicolai Josuttis）的《C++准库：自修教程与参考手册》是最佳选择。本书这一部分提供了丰富的描述和示例，旨在为读者解决任何需要用到标准库的问题奠定良好的基础。但是，一些技术和主题很少使用，因此这里不会涵盖。如果有特定的主题在这一部分找不到，请参考另外两本书。本书并不是想取代那些书，而是补充它们。特别是，我们希望您在学习完以下各章的内容之后，能够更容易地理解那些书。

注意，这一部分不会包含详尽的文档来描述标准 C++库中的每个函数和类。我们将完整的描述留给其他人；特别是“C++参考手册”网站（https://zh.cppreference.com）。这是一个优秀的标准库文档在线源，采用网页格式，可以将它保存在自己的计算机上，并在需要查询时直接用浏览器打开。它包含有关 C 和 C++库的完整参考页面（因此非常适合我们所有的标准 C/C++编程问题）。电子文档之所以有效，不仅是因为可以随时随地携带，还因为可以输入关键字来查找自己感兴趣的内容。[①]

当我们积极编程时，这些资源应该可以满足所有对参考资料的需求（可以使用它们来查找这一部分任何不清楚的内容）。附录 A 提供了更多的参考资料。

在这一部分中，第 4 章介绍了标准 C++ `string` 类。它很强大，能简化平时可能遇到的大多数文本处理任务。在 C 中可以对字符串执行的任何操作，都可以在 `string` 类中通过成员函数调用来完成。

第 4 章涵盖了 `iostream` 库，可以利用其中的类来处理文件、字符串目标和系统控制台的输入和输出。

虽然第 5 章“深入理解模板”不针对任何特定的库，但它是接下来两章的必要准备。在第 6 章中，我们将研究标准 C++库提供的泛型算法。由于它们是用模板实现的，因此这些算法可以应用于任何对象**序列**。第 7 章介绍了标准容器及其相关的迭代器。之所以先在第 6 章学习算法，因为只需使用数组和 `vector` 容器（我们从第 1 卷开始就一直在用）就可以

[①] 译注：这一段稍微修改了原书，使之更适合当代的程序员。

把所有算法问题搞清楚。将标准算法与容器结合使用也是很自然的，因此在学习容器之前熟悉算法是件好事。

第 3 章 深入理解字符串

在 C 语言中，使用字符数组进行字符串处理是一项费时费力的工作。使用字符数组的程序员必须仔细区分静态字符串[①]、栈上数组[②]和堆上数组[③]之间的差异。，时刻注意在栈和堆上创建的数组之间的区别，还要注意有时传递一个 char*就可以了，有时则必须拷贝整个数组。

由于字符串操作是如此常见，因此字符数组是误解和 bug 的一个重要根源。尽管如此，许多 C++初学者仍乐于通过自定义字符串类来深入理解对象和类的概念。然而，标准 C++库中的 string 类一劳永逸地解决了字符数组操作中的诸多问题。即使在赋值和拷贝构造期间，它也会跟踪内存。因此，我们可以省去手动管理内存的麻烦，从而将精力集中在更高级别的编程任务上。

本章将深入探讨标准 C++中的 string 类，首先将探讨 C++ string 的概念，并对比它与传统的 C 风格字符数组的差异。[④]我们将介绍可以使用 string 对象执行的操作，还会解释 C++ string 对字符集和字符串数据转换的改进。

文本处理是最古老的编程应用之一，因此 C++ string 大量借鉴了 C 和其他语言中长期使用的思想和术语。这一点都不令人奇怪。当您准备好熟悉 C++字符串时，这个事实应该会让您感到安心。无论选择哪种编程体例，我们都需要对 string 执行以下常规操作：

- 创建或修改 string 中存储的字符序列。
- 检测 string 中是否存在特定元素。
- 在 string 字符的各种表示方案之间进行转换。

本章将解释如何使用 C++ string 对象来完成这些任务。

[①] 译注：也称为“字符串字面值/字面量”或者“字符串常量”，它们一旦定义就不能修改。例如，执行 char *str = "Hello, world!";这行代码后，字符串将存储在程序的只读数据段，变量 str 指向这个字符串的第一个字符。

[②] 译注：在函数内部声明的字符数组，存储在栈上。生命周期仅限于函数作用域，函数结束后自动释放。

[③] 译注：使用 malloc 动态分配的字符数组存储在堆上。生命周期由程序员手动控制，需要手动释放，否则会造成内存泄漏。

[④] 本章一些素材的原创者是南希 • 尼古拉森 （Nancy Nicolaisen）。

3.1 认识字符串

在 C 语言中，字符串不过是一个字符数组，其中最后一个数组元素总是二进制零（通常称为**空终止符**）。C++ string 与其 C 祖先之间存在显著差异。首先，也是最重要的一点，C++ string 隐藏了其中包含的字符序列的物理表示。我们再也不用关心数组的大小或者空终止符了。string 还包含有关其数据的大小和存储位置的某些“内务维护”（housekeeping）信息。

具体来说，C++ string 对象知道它在内存中的起始位置、内容、字符长度以及在 string 对象必须调整其内部数据缓冲区之前可以增长到的字符长度。因此，C++ string 大幅减少了三种最常见和最具破坏性的 C 编程错误：数组越界，尝试通过未初始化或具有错误值的指针访问数组，以及在数组不再占用曾经分配给它的存储空间后留下“空悬指针”（dangling pointer）。

C++ 标准没有规定 string 类的内存布局的具体实现细节。这是为了兼顾灵活性与一致性。编译器厂商可以自由选择不同的实现方式，但必须保证用户在使用 string 对象时获得一致的行为。特别是，标准并未严格定义在什么情况下为字符串对象分配内存。字符串的分配规则允许但不强制使用某种“引用计数”（reference counting）实现[①]。然而，无论实现是否使用了引用计数，其语义都必须相同。换言之，在 C 语言中，每个 char 数组都独占内存中的一个物理区域。而在 C++ 中，单个 string 对象可能会、也可能不会独占内存中的一个物理区域。相反，可以使用引用计数来避免重复拷贝数据。但在这种情况下，这些单独的对象必须看起来（并表现得）好似它们独占唯一的存储区域一样。以下示例程序对此进行了说明：

```
//: C03:StringStorage.h
#ifndef STRINGSTORAGE_H
#define STRINGSTORAGE_H
#include <iostream>
#include <string>
#include "../TestSuite/Test.h"
using std::cout;
using std::endl;
using std::string;

class StringStorageTest : public TestSuite::Test {
public:
    void run() {
        string s1("12345");
```

[①] 译注：“引用计数”是一种内存管理技术，通过记录有多少个指针指向同一块内存区域来决定何时释放这块内存。

```
        // 取决于具体的实现，以下代码既可能会将第一个字符串拷贝到第二个字符串，
        // 也可能会使用引用计数来模拟拷贝动作。
        string s2 = s1;
        test_(s1 == s2);
        // 无论哪种方式，以下语句都只会修改 s1：
        s1[0] = '6';
        cout << "s1 = " << s1 << endl; // 62345
        cout << "s2 = " << s2 << endl; // 12345
        test_(s1 != s2);
    }
};
#endif // STRINGSTORAGE_H ///:~
```

下例展示了如何使用 StringStorageTest 类：[①]

```
//: C03:StringStorage.cpp
//{L} ../TestSuite/Test
#include "StringStorage.h"
int main() {
  StringStorageTest t;
  t.run();
  return t.report();
} ///:~
```

如果某个实现仅在字符串被修改时才创建唯一（独占）的拷贝，我们就说它采用了**写时拷贝**（copy-on-write）策略。当字符串仅作为值参数[②]或用于其他只读场景时，这一策略可以节省时间和空间。

对于一个具体的库实现，它是否使用引用计数对于 string 类的用户应该是透明的。但令人遗憾的是，实际情况并非总是如此。在多线程程序中，几乎不可能安全地使用引用计数实现。[③]

[①] 译注：编译命令是 cl /EHsc StringStorage.cpp ../TestSuite/Test.cpp。

[②] 译注：字符串以“传值”方式传入函数。换言之，在函数内部对字符串的修改，不会反映到原始字符串中。在这种情况下，原始字符串仍然是“只读”的，不需要在堆上重新分配一块空间。当然，如果字符串以“传引用”的方式传入，那就是另一回事了。

[③] 引用计数实现很难做到线程安全，详情请参见赫布·萨特（Herb Sutter）的 *Exceptional C++*一书。要想更多地了解多线程编程，请参见第 10 章。

3.2 创建与初始化 C++字符串

字符串的创建与初始化是一个相当简单明了且灵活的过程。在下面的 SmallString.cpp 示例程序中，第一个字符串 imBlank（我是空串）仅声明而不初始化。换言之，它不包含任何初始值。与 C 字符数组不同的是，C char 数组在初始化之前会包含随机且毫无意义的位模式（bit pattern），而 imBlank 确实包含有意义的信息。该字符串对象被初始化为不包含任何字符，并且可以使用类成员函数正确报告其长度为零且不含数据元素。

下一个字符串 heyMom（妈!）初始化为字符串字面值"我的袜子呢？"。要进行这种形式的初始化，需要提供带引号的字符数组作为 string 构造函数的实参。相比之下，standardReply 直接通过赋值来进行初始化。程序中的最后一个字符串 useThisOneAgain 则重新初始化一个现有的 C++ string 对象。也就是说，这个示例说明字符串对象支持执行以下基本操作：

- 创建一个空 string，推迟到以后再用字符数据来初始化它。
- 将字符串字面值（带引号的字符数组）作为实参传递给构造函数来初始化 string。
- 使用赋值操作符（=）符来初始化字符串。
- 使用一个 string 来初始化另一个 string。

```
//: C03:SmallString.cpp
#include <string>
using namespace std;
int main() {
    string imBlank;
    string heyMom("我的袜子呢？ ");
    string standardReply = "Beamed into deep space on wide angle dispersion?";
    string useThisOneAgain(standardR   eply);
} ///:~
```

这些是最简单的 string 初始化形式，但还存在不少变体，它们提供了更多的灵活性和控制。可以执行以下操作：

- 使用 C char 数组或 C++ string 的一部分，
- 使用 operator+来合并不同的初始化数据源，
- 或者使用 string 的 substr()成员函数来创建子串。

以下程序演示了这些功能：

```
//: C03:SmallString2.cpp
#include <string>
#include <iostream>

using namespace std;

int main() {
```

```
    string s1("What is the sound of one clam napping?");
    string s2("Anything worth doing is worth overdoing.");
    string s3("I saw Elvis in a UFO");

    // 拷贝前 8 个字符
    string s4(s1, 0, 8);
    cout << s4 << endl;  // 输出：What is

    // 拷贝源字符串中间位置的 6 个字符
    string s5(s2, 15, 6);
    cout << s5 << endl;  // 输出：doing

    // 从中间位置拷贝到结尾
    string s6(s3, 6, 15);
    cout << s6 << endl;  // 输出：Elvis in a UFO

    // 合并多个字符串
    string quoteMe = s4 + "that" +
                     // substr()从元素 20 开始拷贝 10 个字符：
                     s1.substr(20, 10) + s5 +
                     // substr()从元素 5 开始拷贝最多 100 个字符，
                     // 或者一直拷贝到字符串末尾：
                     "with" + s3.substr(5, 100) +
                     // 像下面这样拷贝单个字符是允许的：
                     s1.substr(37, 1);
    cout << quoteMe << endl;  // 输出：What is that one clam doing with Elvis in a UFO?
} ///:~
```

string 的 substr()成员函数获取子串的起始位置作为第一个参数，获取要选择的字符数作为第二个参数。这两个参数都有默认值。如果使用带有空参数列表的 substr()，则会生成整个字符串的拷贝，因此是拷贝字符串的一种简便的方法。

以下是程序的输出：

```
What is
doing
Elvis in a UFO
What is that one clam doing with Elvis in a UFO?
```

注意 quoteMe 这个字符串是如何创建的。C++允许在一个语句中混合使用多种 string 初始化技术，这是一个相当灵活且方便的功能。另外要注意的是，最后一个初始化器仅从源 string 中拷贝了一个字符（参见注释）。

另一种稍微有点不好理解的初始化技术涉及到使用 string 迭代器 string::begin()和 string::end()。这种技术将 string 视为容器对象（到目前为止，我们主要使用的是 vector 形式的容器对象，但第 7 章会介绍更多容器），使用**迭代器**来指示字符序列的开头和结尾。这样一来，就可以将两个迭代器传递给 string 构造函数，该构造函数会将一个迭代器到另一个迭代器之间的所有内容拷贝到一个新字符串中。

```
//: C03:StringIterators.cpp
#include <string>
#include <iostream>
#include <cassert>
using namespace std;

int main() {
    string source("测试字符串");
    string s(source.begin(), source.end());
    assert(s == source);
    cout << s; // 应输出：测试字符串
} ///:~
```

迭代器并不限于 begin()和 end()（即字符串的开头和结尾），完全可以在它们的基础上递增 1（++）、递减 1（--）或增加（+）一个任意的整数偏移量，以便从源 string 中提取特定的一部分字符。

C++字符串不能使用单个 char（例如，'a'）、ASCII 码（例如，65）或其他整数值（例如，0x37）来初始化。[1]但是，可以用单个 char 的多个拷贝来初始化。

```
//: C03:UhOh.cpp
#include <string>
#include <cassert>
using namespace std;

int main() {
    // 错误：不能用单个 char 来初始化
    //! string nothingDoing1('a');

    // 错误：不能用整数来初始化
    //! string nothingDoing2(0x37);

    // 但以下合法：
    string okay(5, 'a');
    assert(okay == string("aaaaa"));
} ///:~
```

在本例中，第一个实参指定了要在字符串中存储第二个实参的多少个拷贝。注意，第二个实参只能是单个 char，不能是 char 数组。

3.3 字符串操作

对于有 C 语言编程经验的人来说，肯定不会对那一组用于写入、查找、修改和拷贝 char

[1] 译注：这是为了保证类型安全，避免在字符串和字符之间发生隐式转换，从而导致意想不到的错误。

数组的函数感到陌生。但是，使用这些标准 C 库函数来处理 char 数组时，有两个不好的地方。首先，存在两套组织松散的函数：一套是“普通”函数，另一套则需要指定要处理的字符数量。C char 数组库的函数列表会让毫无防备的用户感到震惊，因为这些函数的名称晦涩难懂，而且大多数难以发音。虽然这些函数的类型和参数数量相对一致，但要正确使用它们，必须注意函数命名和参数传递的细节。

标准 C char 数组工具的第二个固有问题在于，它们都显式地假设 char 数组包含一个终止符。如果由于疏忽或错误而遗漏或覆盖了终止符，那么几乎没有东西可以阻止 C char 数组函数访问超出分配空间范围的内存，这有时会导致灾难性的后果。

C++ string 在便利性和安全性方面有了显著的改进。就实际的字符串处理操作而言，string 类中的成员函数数量大致与 C 库中的函数数量相当。但是，由于重载的存在，其功能变得更加强大。由于命名更合理，而且在适当的地方使用了默认参数，所以 string 类比 C 库中的各种 char 数组函数好用多了。

3.3.1 追加、插入和拼接字符串

C++ string 最有价值和最便利的一个地方是，它们可以根据需要自动扩容，而无需程序员的干预。这不仅使得处理 string 的代码本质上变得更加可靠，而且一个繁琐的“维护”工作也几乎被完全消除了——即跟踪字符串存储空间的边界。例如，如果创建一个 string 对象，并用'X'的 50 个拷贝初始化它，然后又在其中存储"Zowie"的 50 个拷贝，那么该对象会自行重新分配足够的存储空间以适应数据的增长。若处理的字符串改变了大小，但不知道这个改变具体有多大，那么这一特性就能“帮上大忙”。注意，字符串成员函数 append()和 insert()会在字符串增长时透明地重新分配存储空间。

```
//: C03:StrSize.cpp
#include <string>
#include <iostream>
using namespace std;

int main() {
    string mirrorListening("所以期待渡鸦的只言片语。");
    cout << mirrorListening << endl;

    // 字符串当前实际包含多少数据？size 是指当前大小
    cout << "大小 = " << mirrorListening.size() << endl;

    // 不重新分配内存的情况下总共可以存储多少？capacity 是指最大容量
    cout << "容量 = " << mirrorListening.capacity() << endl;

    // 在 mirrorListening[4]之前（第三个汉字之前）插入指定的字符串：
    mirrorListening.insert(4, "守候在除夕");
    cout << mirrorListening << endl;

    // 显示字符串的当前大小和容量
```

```
    cout << "大小 = " << mirrorListening.size() << endl;
    cout << "容量 = " << mirrorListening.capacity() << endl;

    // 确保有足够的空间
    mirrorListening.reserve(500);

    // 将指定的字符串追加到字符串末尾
    mirrorListening.append("天青地黑万籁俱寂没有踪迹，她早已经在镜中给了未来以定义。");
    cout << mirrorListening << endl;

    // 显示字符串的当前大小和容量
    cout << "大小 = " << mirrorListening.size() << endl;
    cout << "容量 = " << mirrorListening.capacity() << endl;
} ///:~
```

上述程序的输出结果如下所示（使用 Microsoft VC++编译器编译）：

```
所以期待渡鸦的只言片语。
大小 = 24
容量 = 31
所以守候在除夕期待渡鸦的只言片语。
大小 = 34
容量 = 47
所以守候在除夕期待渡鸦的只言片语。天青地黑万籁俱寂没有踪迹，她早已经在镜中给了未来以定义。
大小 = 90
容量 = 511
```

这个例子很好地证明了，C++ string 不仅能为我们自动管理内存，还提供了丰富的工具让我们对字符串的存储情况进行精细控制。我们可以非常方便地增加或减少字符串的存储空间。其中，size()函数（等同于 length()函数）返回当前实际存储在字符串中的字符数。capacity()函数告诉我们字符串最多能容纳多少字符而不必重新分配内存。reserve()函数可以提前为字符串预留一定大小的空间，这有助于提高效率，避免频繁的内存重新分配。注意，调用 capacity()时返回的值至少与最近一次调用 reserve()所返回的值一样大。本例没有展示的一个成员函数是 resize()，它用于改变字符串的大小。如果新的大小大于当前字符串的大小，那么会在末尾追加空格；否则会对字符串进行截断处理。注意，resize()的一个重载版本允许追加不同的字符。

至于 string 类的成员函数具体如何为数据分配空间，则要取决于库的实现。本例是用 Microsoft VC++编译器来测试的。但是，使用 g++编译器来测试时，会在不同的边界进行重新分配，如下所示：

```
所以期待渡鸦的只言片语。
大小 = 24
容量 = 24
所以守候在除夕期待渡鸦的只言片语。
大小 = 34
容量 = 48
```

```
所以守候在除夕期待渡鸦的只言片语。天青地黑万籁俱寂没有踪迹，她早已经在镜中给了未来以定义。
大小 = 90
容量 = 500
```

换言之，这种实现会在字边界（每个字都包含偶数字节，因此边界值都是偶数）处重新分配。string 类的架构师努力使其能够混合使用 C char 数组和 C++ string 对象，因此从 Microsoft VC++的输出结果可知，StrSize.cpp 报告的容量数字反映出在这个特定实现中，保留了一个字节以便轻松容纳终止符的插入。（容量本应是 32，48，512，但显示为 31，47，511）。

3.3.2 替换字符串中的字符

insert()函数使用起来特别方便，因为向字符串中插入字符时，我们不必担心会超出存储空间或覆盖紧跟在插入点后的字符。若空间不足，会自动扩容，而且现有字符会自觉向后挪动以容纳新元素。但有的时候，这可能不是我们想要的结果。如果希望字符串的大小保持不变，那么可以使用 replace()函数覆盖字符。replace()有多个重载版本，其中最简单的一个接受三个实参：一个整数表示从字符串的什么位置开始替换，一个整数表示从原始字符串中删除的字符数，以及一个替换字符串（其字符数可能有别于删除的字符数）。下面是一个简单的例子：

```
//: C03:StringReplace.cpp
// 在字符串中实现简单的查找和替换功能
#include <cassert>
#include <string>
using namespace std;

int main() {
    string s("A piece of text");
    string tag("$tag$");
    s.insert(8, tag + ' ');
    assert(s == "A piece $tag$ of text");
    int start = s.find(tag);
    assert(start == 8);
    assert(tag.size() == 5);
    s.replace(start, tag.size(), "hello there");  // 替换字符和被删除的字符在数量上不同
    assert(s == "A piece hello there of text");
} ///:~
```

我们首先在 s 中插入 tag（注意，是在指定的插入位置之前而不是之后插入，本例还在 tag 后添加了一个额外的空格）。然后，我们执行 replace()，找到 tag 并替换了它。

在执行 replace()之前，应该先核实是否找到了要替换的内容。上例是替换成一个 char*（即"hello there"），但它的重载版本也允许替换成 string。下例更完整地演示了 replace()函数：

```
//: C03:Replace.cpp
```

```
#include <cassert>
#include <cstddef> // 为了使用 size_t
#include <string>
using namespace std;

void replaceChars(string& modifyMe,
    const string& findMe, const string& newChars) {

    // 从位置 0 开始，在 modifyMe 中查找要替换的字符串 findMe
    size_t i = modifyMe.find(findMe, 0);

    // 是否找到了要替换的字符串？
    if(i != string::npos)
        // 使用 newChars 替换 findMe
        modifyMe.replace(i, findMe.size(), newChars);
}

int main() {
    string mirrorListening = "一更鼓儿天，"
                             "这一去金川十呀么十七年。";
    string replacement("十八"); // 这是替换字符串
    string findMe("十七");      // 这是要被替换的字符串

    // 在 mirrorListening 中查找"十七"并用"十八"覆盖它
    replaceChars(mirrorListening, findMe, replacement);

    // 验证替换成功
    assert(mirrorListening == "一更鼓儿天，"
                              "这一去金川十呀么十八年。");
} ///:~
```

如果 replace 函数没有找到要替换的字符串，那么它将返回 string::npos。npos 数据成员是 string 类的静态常量成员，表示一个不存在的字符位置。[①]

每次执行 insert()来插入新内容时，都会造成字符串实际占用的存储空间的增长。与之不同的是，如果 replace()操作所替换的内容没有超出字符串的末尾，那么字符串的存储空间是不会增长的。不过，如果用于替换的内容超出了原字符串的末尾，那么字符串的存储空间就会发生增长。下面展示了一个例子。

```
//: C03:ReplaceAndGrow.cpp
#include <cassert>
#include <string>
```

[①] npos 是“no position”的缩写，是字符串分配器的 size_type（默认为 std::size_t）能表示的最大值。

```
using namespace std;

int main() {
    string mirrorListening("二更鼓儿敲，");
    string replacement("敲得泪珠儿对对往下掉。");

    // 第一个实参相当于说：“对超出现有字符串末尾的字符进行替换”：
    mirrorListening.replace(mirrorListening.size() ,
                    replacement.size(), replacement);
    assert(mirrorListening == "二更鼓儿敲，"
                              "敲得泪珠儿对对往下掉。");
} ///:~
```

本例调用的 replace()“替换”的是超出当前字符串末尾的“字符”，这相当于一个追加（append）操作。注意，本例执行的 replace()使字符串的存储空间发生了增长。

您阅读本章的目的可能是想知道如何做一些相当简单的事情，例如将一个字符的所有实例替换成一个不同的字符。在阅读了刚才关于“替换”的主题后，您或许以为已经找到了答案。但是，您马上就会接触到字符组、计数和其他看起来更复杂的东西。难道 string 类没有提供一种简单的方法来实现“全部替换”功能吗？[①]

可以自己使用 find()和 replace()成员函数来轻松地写一个这样的函数，如下所示：

```
//: C03:ReplaceAll.h
#ifndef REPLACEALL_H
#define REPLACEALL_H
#include <string>

// 替换字符串中的所有匹配项。
std::string& replaceAll(std::string& context,
                        const std::string& from,
                        const std::string& to);
#endif // REPLACEALL_H ///:~
```

以下是实现文件：

```
//: C03:ReplaceAll.cpp {O}
#include <cstddef>
#include "ReplaceAll.h"

using namespace std;

// 替换字符串中的所有匹配项。
```

[①] 译注：确实没有现成的，只能自己写。

```
string& replaceAll(string& context,
                   const string& from,
                   const string& to) {
    size_t lookHere = 0;
    size_t foundHere;

    while ((foundHere = context.find(from, lookHere)) != string::npos) {
        context.replace(foundHere, from.size(), to);
        lookHere = foundHere + to.size(); // 查找位置递增替换字符串的大小
    }

    return context;
} ///:~
```

这里使用的 find()版本获取起始查找位置 lookHere 作为第二个实参，并在未发现目标项的前提下返回 string::npos。这个实现的重点在于，变量 lookHere 所容纳的查找位置一定要推进替换字符串的大小。这样一来，如果 from 是 to 的子串，就可以避免在替换后重新查找相同的子串。[①]以下应用程序对 replaceAll 函数进行了测试：[②]

```
//: C03:ReplaceAllTest.cpp
//{L} ReplaceAll
#include <cassert>
#include <iostream>
#include <string>
#include "ReplaceAll.h"
using namespace std;

int main() {
  string text = "要在先祖的序列里寻找自己，"
                "要用生命为时光去乞讨身体。";
  replaceAll(text, "要", "不要");
  assert(text == "不要在先祖的序列里寻找自己，不要用生命为时光去乞讨身体。");
} ///:~
```

可以看出，仅凭 string 类本身并不能解决我们所有可能的问题。许多解决方案都留给了标准库中的算法[③]，因为 string 类可以被视为一个 STL 序列（通过前面讨论过的迭代器）。所有泛型算法都可以在容器内的一个“范围”上工作。通常，该范围是从“容器的开始到

[①] 译注：不这样做会造成死循环。例如，假定 from 是"ab"，to 是"abab"，原字符串是"aabc"。从字符串开头开始查找"ab"，找到第一个"ab"，并用"abab"替换。替换后字符串变为"a**abab**c"。如果 lookHere 容纳的查找位置不前进 to.size()，就会反复找到替换字符串中的"ab"并替换成"abab"。

[②] 译注：编译命令是 cl /EHsc ReplaceAllTest.cpp ReplaceAll.cpp。

[③] 将在第 6 章讨论算法。

结束”。可以将 string 对象视为一个字符容器：我们使用 string::begin()获取范围的开始，使用 string::end()获取范围的结束。以下示例展示了如何使用 replace()算法将字符'X'的所有实例替换为'Y'：

```
//: C03:StringCharReplace.cpp
#include <algorithm>
#include <cassert>
#include <string>

using namespace std;

int main() {
    string s("aaaXaaaXXaaXXXaXXXXaaa");
    replace(s.begin(), s.end(), 'X', 'Y');
    assert(s == "aaaYaaaYYaaYYYaYYYYaaa");
} ///:~
```

注意，这个 replace()并不是作为 string 类的成员函数调用的。此外，与仅执行一次替换的 string::replace()函数不同，replace()算法会将一个字符的所有实例替换为另一个字符。

replace()算法仅适用于容器中单个元素（本例是 char 对象），并且不会替换引号包围的字符数组或 string 对象。由于 string 行为类似于 STL 序列，因此还可以向它应用其他许多算法，以解决那些未被 string 成员函数直接解决的问题。

3.3.3 使用非成员重载操作符来拼接字符串

正在学习 C++字符串处理的 C 程序员会非常高兴地发现，现在可以简单地使用 operator+和 operator+=来合并和附加字符串。这些操作符使字符串拼接[①]在语法上类似于数值数据相加。

```
//: C03:AddStrings.cpp
#include <string>
#include <cassert>
#include <iostream>
using namespace std;

int main() {
    string s1("谁不是");
    string s2("错过了");
    string s3("四下报更的鼓声.");
    string s4("总有人偷偷拨弄镜月的指针");
```

[①] 译注：也可以说成“字符串连接”。

```
    // 找到句点符号的索引位置，rfind()的详情参见下一节
    size_t peroidPos = s3.rfind('.');

    // 使用operator+拼接字符串
    s1 = s1 + s2;
    assert(s1 == "谁不是错过了");

    // 另一种拼接字符串的方法
    s1 += s3;
    assert(s1 == "谁不是错过了四下报更的鼓声.");

    // 在确保索引有效的前提下，对右侧的字符串进行索引
    if (peroidPos < s3.size()) {
        s1 += s4 + s3[peroidPos]; // 在最后添加一个句点
        assert(s1 == "谁不是错过了四下报更的鼓声.总有人偷偷拨弄镜月的指针.");
    } else {
        cout << "索引无效" << endl;
    }
} ///:~
```

使用 operator+和 operator+=操作符，我们可以灵活且方便地拼接字符串数据。在操作符右侧，可以使用几乎任何能求值为一个字符组（其中包含一个或多个字符）的类型。

3.4 在字符串中查找

string 类的 find 系列成员函数用于在给定字符串中定位一个或一组字符。表 3.1 总结了这些函数及其常规用法。

表 3.1 find 系列成员函数

find 系列成员函数	描述
find()	在字符串中查找指定的字符或字符组，并返回第一个匹配项的起始位置；如果没有找到匹配项，则返回 npos。
find_first_of()	查找一个目标字符串，并返回指定字符组中第一个匹配项的位置；如果没有找到匹配项，则返回 npos。
find_last_of()	查找一个目标字符串，并返回指定字符组中最后一个匹配项的位置；如果没有找到匹配项，则返回 npos。
find_first_not_of()	查找一个目标字符串，并返回和指定字符组中的任何字符都不匹配的第一个元素的位置；如果没有找到这样的元素，则返回 npos。

find_last_not_of()	查找一个目标字符串，并返回和指定字符组中的任何字符都不匹配的具有最大下标的元素的位置；如果没有找到这样的元素，则返回 npos。
rfind()	从字符串末尾向前查找指定字符或字符组，并返回匹配项的起始位置；如果没有找到匹配项，则返回 npos。

find()最简单的用法是在字符串中查找一个或多个字符。这个重载版本的 find()获取一个指定了要查找的字符或字符组的参数，并可选择获取另一个参数，该参数指出从字符串中的哪个位置开始查找子串（默认从位置 0 开始查找）。将 find 调用放到循环中，可以轻松地遍历整个字符串，找出字符串中给定字符或字符组的所有实例。

以下程序使用**埃拉托斯特尼筛法**（The Sieve of Eratosthenes）来查找小于 50 的质数。该方法从最小的质数 2（唯一的偶数质数）开始，将 2 的所有后续倍数标记为非质数（合数），并对下一个候选质数重复此过程。SieveTest 类的构造函数在初始化 sieveChars 字符数组时，会设置它的初始大小，并将值'P'写入它的每个成员。类的头文件如下所示：

```
//: C03:Sieve.h
#ifndef SIEVE_H
#define SIEVE_H
#include <cmath>
#include <cstddef>
#include <string>
#include "../TestSuite/Test.h"

using std::size_t;
using std::sqrt;
using std::string;

class SieveTest : public TestSuite::Test {
    string sieveChars;
public:
    // 创建包含 50 个字符的一个字符串，并将每个元素设置为'P'，P 代表质数
    SieveTest() : sieveChars(50, 'P') {}

    void run() {
        findPrimes();
        testPrimes();
    }

    bool isPrime(int p) {
        if(p == 0 || p == 1) return false;
        int root = int(sqrt(double(p)));
        for(int i = 2; i <= root; ++i)
```

```
            if(p % i == 0) return false;
        return true;
    }

    void findPrimes() {
        // 根据定义，0 和 1 都不是质数。
        // 因此，将这些元素改为"N"表示非质数。
        sieveChars.replace(0, 2, "NN");

        // 遍历数组
        size_t sieveSize = sieveChars.size();
        int root = int(sqrt(double(sieveSize)));
        for(int i = 2; i <= root; ++i)
            // 找出所有倍数
            for(size_t factor = 2; factor * i < sieveSize; ++factor)
                sieveChars[factor * i] = 'N';
        // 执行到这里， sieveChars 字符数组的内容是：
        // NNPPNPNPNNNPNPNNNPNPNNNPNNNNNPNPNNNNNPNNNPNPNNNPNN
    }

    void testPrimes() {
        size_t i = sieveChars.find('P');
        while(i != string::npos) {
            test_(isPrime(i++));
            i = sieveChars.find('P', i);
        }

        i = sieveChars.find_first_not_of('P');
        while(i != string::npos) {
            test_(!isPrime(i++));
            i = sieveChars.find_first_not_of('P', i);
        }
    }
};
#endif // SIEVE_H ///:~
```

以下应用程序对 SieveTest 类进行了测试：[①]

```
//: C03:Sieve.cpp
//{L} ../TestSuite/Test
#include "Sieve.h"

int main() {
    SieveTest t;
```

[①] 译注：编译命令是 cl /EHsc Sieve.cpp ../TestSuite/Test.cpp。

```
    t.run();
    return t.report();
} ///:~
```

find()函数可以向前遍历一个 string 对象，检测一个或一组字符的多个实例，而find_first_not_of()则查找不匹配的其他字符或子串。

string 类没有提供用于更改字符串大小写的函数，但可以使用标准 C 库函数 toupper()和tolower()来轻松地自己创建（这两个 C 库函数函数一次改变一个字符的大小写）。下例展示了如何执行不区分大小写的查找：

```
//: C03:Find.h
#ifndef FIND_H
#define FIND_H
#include <cctype>
#include <cstddef>
#include <string>
#include "../TestSuite/Test.h"

using std::size_t;
using std::string;
using std::tolower;
using std::toupper;

// 创建 s 的大写拷贝
inline string upperCase(const string& s) {
    string upper(s);
    for(size_t i = 0; i < s.length(); ++i)
        upper[i] = toupper(upper[i]);
    return upper;
}

// 创建 s 的小写拷贝
inline string lowerCase(const string& s) {
    string lower(s);
    for(size_t i = 0; i < s.length(); ++i)
        lower[i] = tolower(lower[i]);
    return lower;
}

class FindTest : public TestSuite::Test {
    string chooseOne;
public:
    FindTest() : chooseOne("Eenie, Meenie, Miney, Mo") {}

    void testUpper() {
        string upper = upperCase(chooseOne);
        const string LOWER = "abcdefghijklmnopqrstuvwxyz";
        test_(upper.find_first_of(LOWER) == string::npos);
```

```
    }

    void testLower() {
        string lower = lowerCase(chooseOne);
        const string UPPER = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
        test_(lower.find_first_of(UPPER) == string::npos);
    }

    void testSearch() {
        // 区分大小写的查找
        size_t i = chooseOne.find("een");
        test_(i == 8);

        // 查找小写：
        string test = lowerCase(chooseOne);
        i = test.find("een");
        test_(i == 0);
        i = test.find("een", ++i);
        test_(i == 8);
        i = test.find("een", ++i);
        test_(i == string::npos);

        // 查找大写：
        test = upperCase(chooseOne);
        i = test.find("EEN");
        test_(i == 0);
        i = test.find("EEN", ++i);
        test_(i == 8);
        i = test.find("EEN", ++i);
        test_(i == string::npos);
    }

    void run() {
        testUpper();
        testLower();
        testSearch();
    }
};

#endif // FIND_H ///:~
```

以下程序进行了测试：[①]

```
//: C03:Find.cpp
//{L} ../TestSuite/Test
#include "Find.h"
```

[①] 译注：编译命令是 `cl /EHsc Find.cpp ../TestSuite/Test.cpp`。

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

int main() {
    FindTest t;
    t.run();
    return t.report();
} ///:~
```

upperCase()和 lowerCase()函数都遵循相同的模式：拷贝输入字符串并改变大小写。Find.cpp 程序并不是处理大小写敏感性问题的最佳方案，因此我们将在探讨字符串比较时重新审视这个问题（参见 3.4.5 节）。

3.4.1 反向查找

如果需要从 string 的末尾反向查找（以“后入先出”的顺序查找数据），那么可以使用 string 的 rfind()成员函数。

```
//: C03:Rparse.h
#ifndef RPARSE_H
#define RPARSE_H
#include <cstddef>
#include <string>
#include <vector>
#include "../TestSuite/Test.h"

using std::size_t;
using std::string;
using std::vector;

class RparseTest : public TestSuite::Test {
    // 这个私有成员用于存储多个单词
    vector<string> strings;

public:
    void parseForData() {
        // ';' 字符将是定界符
        string s("now.;sense;make;to;going;is;This");
        // 字符串的最后一个元素：
        int last = s.size();
        // 当前单词的开始位置：
        size_t current = s.rfind(';');
        // 从字符串的末尾反向遍历：
        while(current != string::npos) {
            // 将每个单词压入向量，
            // 在拷贝之前递增 current 以避免拷贝定界符：
            ++current;
            strings.push_back(s.substr(current, last - current));
            // 回到刚刚找到的定界符前面，并将 last 设为下一个单词的末尾：
            current -= 2;
```

```
            last = current + 1;
            // 寻找下一个定界符：
            current = s.rfind(';', current);
        }
        // 压入第一个单词——它前面没有定界符：
        strings.push_back(s.substr(0, last));
    }

    void testData() {
        // 以新顺序测试它们：
        test_(strings[0] == "This");
        test_(strings[1] == "is");
        test_(strings[2] == "going");
        test_(strings[3] == "to");
        test_(strings[4] == "make");
        test_(strings[5] == "sense");
        test_(strings[6] == "now.");
        string sentence;
        for(size_t i = 0; i < strings.size() - 1; i++)
            sentence += strings[i] += " ";
        // 手动添加最后一个单词以避免多余的空格：
        sentence += strings[strings.size() - 1];
        test_(sentence == "This is going to make sense now.");
    }

    void run() {
        parseForData();
        testData();
    }
};
#endif // RPARSE_H ///:~
```

以下程序进行了测试：[①]

```
//: C03:Rparse.cpp
//{L} ../TestSuite/Test
#include "Rparse.h"

int main() {
    RparseTest t;
    t.run();
    return t.report();
} ///:~
```

[①] 编译命令是 `cl /EHsc Rparse.cpp ../TestSuite/Test.cpp`。

string 的 rfind()成员函数从字符串的末尾反向查找词元，并报告匹配字符的数组索引；如果失败，则报告 string::npos。

3.4.2 查找一组字符的第一个/最后一个实例

可以利用 find_first_of()和 find_last_of()成员函数创建一个实用工具，去除字符串两端的各种空白字符（空格、制表符、换行符和退格符等）。注意，它不会修改原始字符串，而是返回一个新的字符串。

```
//: C03:Trim.h
// 该通用工具用于去除字符串两端的空白字符
#ifndef TRIM_H
#define TRIM_H
#include <string>
#include <cstddef>

inline std::string trim(const std::string& s) {
    if(s.length() == 0)
        return s;
    std::size_t beg = s.find_first_not_of(" \a\b\f\n\r\t\v");
    std::size_t end = s.find_last_not_of(" \a\b\f\n\r\t\v");
    if(beg == std::string::npos) // 全是空白字符，没有非空白字符，因此全部去除
        return "";
    return std::string(s, beg, end - beg + 1); // 构造新字符串并返回
}
#endif // TRIM_H ///:~
```

第一次测试判断是否为空白字符串（长度为零）；如果是，就不做任何测试，而是返回字符串的一个拷贝。注意，一旦获取了子串的端点（beg 和 end），那么 string 构造函数就会获取子串的起始位置和长度，从旧字符串中构造一个新的字符串。

由于是一个通用工具，因此测试需要彻底，将所有情况都考虑在内。[1]以下是测试类的头文件：

```
//: C03:TrimTest.h
#ifndef TRIMTEST_H
#define TRIMTEST_H
#include "Trim.h"
#include "../TestSuite/Test.h"

class TrimTest : public TestSuite::Test {
    enum { NTESTS = 11 };
    static std::string s[NTESTS];
public:
```

[1] 译注：如果你是直接跳到本章阅读的，请参见第 2 章了解这个测试套件的用法。

```
    void testTrim() {
        test_(trim(s[0]) == "abcdefghijklmnop");
        test_(trim(s[1]) == "abcdefghijklmnop");
        test_(trim(s[2]) == "abcdefghijklmnop");
        test_(trim(s[3]) == "a");
        test_(trim(s[4]) == "ab");
        test_(trim(s[5]) == "abc");
        test_(trim(s[6]) == "a b c");
        test_(trim(s[7]) == "a b c");
        test_(trim(s[8]) == "a \t b \t c");
        test_(trim(s[9]) == "");
        test_(trim(s[10]) == "");
    }

    void run() {
        testTrim();
    }
};
#endif // TRIMTEST_H ///:~
```

以下是测试类的实现：

```
//: C03:TrimTest.cpp {O}
#include "TrimTest.h"

// 初始化静态数据
std::string TrimTest::s[TrimTest::NTESTS] = {
    " \t abcdefghijklmnop \t ",
    "abcdefghijklmnop \t ",
    " \t abcdefghijklmnop",
    "a", "ab", "abc", "a b c",
    " \t a b c \t ", " \t a \t b \t c \t ",
    "\t \n \r \v \f",
    "" // 空白字符串也必须测试
}; ///:~
```

以下是测试程序：[①]

```
//: C03:TrimTestMain.cpp
//{L} ../TestSuite/Test TrimTest
#include "TrimTest.h"
```

[①] 译注：编译命令是 `cl /EHsc TrimTestMain.cpp TrimTest.cpp ../TestSuite/Test.cpp`。

```
int main() {
    TrimTest t;
    t.run();
    return t.report();
} ///:~
```

在 string 数组 s 中，可以看到字符数组被自动转换成了 string 对象。通过这个数组展示的例子，证明虽然会移除两端的空格和制表符，但中间的空格和制表符会得到保留。

3.4.3 从字符串中移除字符

可以使用 erase()成员函数简单且高效地移除字符，该函数获取两个实参：开始移除字符的位置（默认为 0），以及要移除多少个字符（默认为 string::npos）。如果指定的字符数量多于字符串中剩余的字符数，那么剩余的所有字符都会被移除（因此，不带参数调用 erase()会移除字符串中的所有字符）。有的时候，我们需要处理 HTML 文件并移除标记（tag）和特殊字符，从而像在浏览器中呈现的效果一样，只获得纯文本。下例使用 erase()来执行这个任务：

```
//: C03:HTMLStripper.cpp {RunByHand}
//{L} ReplaceAll
// 该过滤器用于移除 HTML 标签和标记
#include <cassert>
#include <cmath>
#include <cstddef>
#include <fstream>
#include <iostream>
#include <string>
#include "ReplaceAll.h"
#include "../require.h"

using namespace std;

string& stripHTMLTags(string& s) {
    static bool inTag = false;
    bool done = false;
    while(!done) {
        if(inTag) {
            // 上一行开始了一个 HTML 标记但没有结束。必须继续查找'>'.
            size_t rightPos = s.find('>');
            if(rightPos != string::npos) {
                inTag = false;
                s.erase(0, rightPos + 1);
            } else {
                done = true;
                s.erase();
            }
```

```
            } else {
                // 查找标记的起始符号：
                size_t leftPos = s.find('<');
                if(leftPos != string::npos) {
                    // 判断标记是否在同一行结束：
                    size_t rightPos = s.find('>');
                    if(rightPos == string::npos) {
                        inTag = done = true;
                        s.erase(leftPos);
                    } else
                        s.erase(leftPos, rightPos - leftPos + 1);
                } else
                    done = true;
            }
        }

        // 移除所有特殊 HTML 字符
        replaceAll(s, "&lt;", "<");
        replaceAll(s, "&gt;", ">");
        replaceAll(s, "&amp;", "&");
        replaceAll(s, " ", " ");
        // 等等...
        return s;
    }

    int main(int argc, char* argv[]) {
        requireArgs(argc, 1,
                    "用法: HTMLStripper 你的测试.html 文件");
        ifstream in(argv[1]);
        assure(in, argv[1]);
        string s;
        while(getline(in, s))
            if(!stripHTMLTags(s).empty())
                cout << s << endl;
    } ///:~
```

这个例子甚至可以移除跨越多行的 HTML 标记。[①]这是通过静态标志 inTag 实现的。任何时候只要找到了标记的起始符号，但没有在同一行找到相应的结束符号，该标志就为 true。stripHTMLTags()函数演示了所有形式的 erase()[②]。这里使用的 getline()是一个在

[①] 为了简化我们的讨论，这个版本没有处理嵌套标记，例如注释。

[②] 有人认为可以通过数学运算来合并一些对 erase()函数的调用，从而减少代码量，提高代码执行效率。例如，将多个 erase()操作合并成一个操作。但是，这种优化方式存在一个潜在的风险。在某些调用中，参与计算的一个操作数是 string::npos（最大的无符号整数）。对这种大数进行计算时，可能会超出整数表示范围，从而导致整数溢出，使整个算法失去意义。

<string>头文件中声明的全局函数，它的优点是可以将任意长度的行存储在其 string 实参中。相反，使用 istream::getline()时，我们就不得不操心字符数组的大小。注意，这个程序使用了本章早些时候介绍的 replaceAll()函数。在下一章中，我们将使用字符串流来创建一个更优雅的解决方案。

3.4.4 字符串比较

字符串比较本质上有别于数字比较。数字有恒定且普遍有意义的值。为了判断两个字符串的大小关系，必须执行字典序（lexical）比较。字典序比较意味着当测试一个字符是否“大于”或“小于”另一个字符时，实际上是在比较字符集排序序列中这些字符的数值形式。以典型的 ASCII 排序序列为例，它为英语中的可打印字符分配了十进制 32 到 127 的值。在 ASCII 排序序列（一个“字典”）中，第一个“字符”是空格，接着是几个常见的标点符号，然后是大写和小写字母。就字母而言，这意味着靠前的字母具有比靠后的字母更低的 ASCII 值。考虑到这些细节，当字典序比较报告 s1“大于”s2 时，意思是说当两者比较时，s1 中第一个不同的字符处于 s2 同一位置的字符之后。

C++提供了多种方式来比较字符串，各有各的优势。最简单的方式是使用非成员的重载操作符函数：operator==，operator!=，operator>，operator<，operator>=和 operator<=。

```
//: C03:CompStr.h
#ifndef COMPSTR_H
#define COMPSTR_H
#include <string>
#include "../TestSuite/Test.h"

using std::string;

class CompStrTest : public TestSuite::Test {
public:
    void run() {
        // 要比较的字符串
        string s1("This");
        string s2("That");
        test_(s1 == s1);
        test_(s1 != s2);
        test_(s1 > s2);
        test_(s1 >= s2);
        test_(s1 >= s1);
        test_(s2 < s1);
        test_(s2 <= s1);
        test_(s1 <= s1);
    }
};
#endif // COMPSTR_H ///:~
```

下面是具体的程序：[1]

```
//: C03:CompStr.cpp
//{L} ../TestSuite/Test
#include "CompStr.h"

int main() {
    CompStrTest t;
    t.run();
    return t.report();
} ///:~
```

使用重载的比较操作符，我们既可以比较整个字符串，也可以比较字符串中的单个字符。注意下面的例子中，比较操作符两侧的实参类型都很灵活。为了提高效率，string类提供了重载的操作符来直接比较 string 对象、用引号包围的文本（字符串字面值）以及指向C风格字符串的指针，而不需要创建临时的string对象。

```
//: C03:Equivalence.cpp
#include <iostream>
#include <string>
using namespace std;

int main() {
    string s2("That"), s1("This");
    // 左值是一个用引号包围的字面值
    // 而右值是一个 string 对象：
    if("That" == s2)
        cout << "匹配" << endl;
    // 左操作数是一个字符串，右操作数则是
    // 指向 C 风格空终止字符串的指针：
    if(s1 != s2.c_str())
        cout << "不匹配" << endl;
} ///:~
```

c_str()函数返回指向 C 风格的“空终止字符串”的一个 const char*，该字符串等同于string 对象的内容。如果需要将字符串传递给标准 C 函数（例如，atoi()或<cstring>头文件中定义的任何函数），那么这个技术就非常好用。将 c_str()返回的值作为非 const 实参传递给任何函数都是错误的。

在string各种重载的操作符中，我们找不到逻辑非（!）或逻辑比较操作符（&&和||）。事实上，也找不到C语言的位操作符&、|、^、~的重载版本。string类重载的非成员比较操作符只能执行完整比较的一个“子集”。在这个子集中，只能对单一字符或字符组执行清晰且无歧义的比较。

---

[1] 译注：编译命令是 cl /EHsc CompStr.cpp ../TestSuite/Test.cpp。

相反，compare()成员函数提供了比非成员操作符更为复杂和精确的比较。它提供了重载版本来执行以下比较：

- 两个完整的字符串，
- 其中一个字符串的一部分与另一个完整的字符串，
- 以及两个字符串的子串。

下例演示了如何使用 compare()和 swap()来比较两个完整的字符串：

```
//: C03:Compare.cpp
// 演示 compare()和 swap()的用法
#include <cassert>
#include <string>
using namespace std;

int main() {
    string first("This");
    string second("That");
    assert(first.compare(first) == 0);
    assert(second.compare(second) == 0);
    // 哪个在字典序上更大?
    assert(first.compare(second) > 0);
    assert(second.compare(first) < 0);
    first.swap(second);
    assert(first.compare(second) < 0);
    assert(second.compare(first) > 0);
} ///:~
```

顾名思义，本例中的 swap()函数用于交换其对象[①]和实参的内容。为了比较一个或两个字符串中的字符子集，需要提供新的实参来指定从哪里开始比较多少个字符。例如，可以使用以下重载版本的 compare()：

```
s1.compare(s1StartPos, s1NumberChars, s2, s2StartPos, s2NumberChars);
```

下面展示了一个例子：

```
//: C03:Compare2.cpp
// 演示重载的 compare()
#include <cassert>
#include <string>
using namespace std;

int main() {
    string first("This is a day that will live in infamy");
```

[①] 译注：在其上调用 swap()的那个对象。

```
    string second("I don't believe that this is what "
                  "I signed up for");
    // 比较两个字符串中的"his is":
    assert(first.compare(1, 7, second, 22, 7) == 0);
    // 比较"his is a"和"his is w":
    assert(first.compare(1, 9, second, 22, 9) < 0);
} ///:~
```

在到目前为止的例子中，我们都使用 C 风格的数组索引语法来引用字符串中的单个字符。C++字符串提供了 s[n]表达式的一个替代方案：at()成员函数。如果一切顺利的话，这两种索引机制在 C++中会产生相同的结果。

```
//: C03:StringIndexing.cpp
#include <cassert>
#include <string>
using namespace std;

int main() {
    string s("1234");
    assert(s[1] == '2');
    assert(s.at(1) == '2');
} ///:~
```

然而，[]和 at()存在一处重要的区别。试图引用越界的数组元素时，如果使用 at()，那么会友好地抛出一个异常。相反，如果使用普通的[]索引语法，那么我们只能自求多福了。

```
//: C03:BadStringIndexing.cpp
#include <exception>
#include <iostream>
#include <string>
using namespace std;

int main() {
    string s("1234");
    // at()通过抛出异常来拯救你:
    try {
        s.at(5);
    } catch(exception& e) {
        cerr << e.what() << endl;
    }
} ///:~
```

负责任的程序员不会使用错误的索引，但如果想利用自动索引检查的好处，那么用 at()替代[]，就可以使自己有机会从对不存在的数组元素的引用中优雅地恢复。用 Microsoft VC++编译器来测试上述程序，结果如下所示：

```
invalid string position
```

at()成员函数抛出 out_of_range 类的一个对象，该类的终极基类是 std::exception。可

以在异常处理程序中捕获该对象，并采取适当的补救措施，例如重新计算有问题的索引或者对数据进行扩展（扩容）。使用 string::operator[]则没有这样的保护，就像在 C 中处理 char 数组一样危险。[①]

3.4.5 字符串和字符 traits

本章前面 Find.cpp 程序中的一个发现引发了我们的深思：为什么标准 C++ string 类不支持不区分大小写的比较？通过深入研究这个问题，我们可以更透彻地理解 C++ string 对象的底层机制。

让我们仔细思考一下“大小写”对字符的影响。希伯来语、波斯语和汉语等语言并不区分大小写，因此对它们来说，大小写的概念毫无意义。如果能有一种机制将某些语言统一为“全大写”或“全小写”，那么设计一个通用的排序方案似乎可行。然而，现实情况更为复杂。许多使用大小写的语言还通过重音符号等来区分字符的意义。这些重音符号在不同的语言中有不同的称呼。例如，西班牙语称为 cedilla，法语称为 circumflex，德语则称为 umlaut。考虑到这些因素，要设计一个能全面涵盖各种语言、区分大小写的排序方案，将会异常复杂且难以实现。

尽管我们通常将 C++ string 看成是一个类，但实际上并非如此。string 类型是对一个更通用的组件——即 basic_string<>模板——的特化。观察标准 C++头文件中 string 的声明便可以理解这一点：[②]

```
typedef basic_string<char> string;
```

为了理解 string 类的本质，下面看看 basic_string<>模板的声明：

```
template<class charT, class traits = char_traits<charT>,
    class allocator = allocator<charT> > class basic_string;
```

第 5 章会详细探讨模板（远超第 1 卷第 16 章的内容）。现在，只需记住 string 类型是在用 char 实例化 basic_string 模板时创建的。在 basic_string<>模板声明内部，我们可以看到下面这一行代码：

```
class traits = char_traits<charT>,
```

[①] 出于上述安全原因，C++标准委员会正在考虑一项提案，在 C++0x 中重新定义 string::operator[]，使其具有与 string::at()完全一致的行为。（译注：很遗憾，直到最新版本的 C++，string::operator[]都不执行边界检查。）

[②] 你的实现可以在此处提供全部三个模板实参（字符类型、字符操作 trait 和内存分配器）。但是，由于最后两个模板参数具有默认实参，所以像这样写就可以了。

它告诉我们由 basic_string<>模板创建的类的行为是由基于 char_traits<>模板的类来规范的。因此，basic_string<>模板可以生成处理各种字符类型的字符串类，而不仅仅是 char 类型。例如，它还可以处理宽字符。为了做到这一点，char_traits<>模板使用字符比较函数 eq()（等于）、ne()（不等于）和 lt()（小于）来控制各种字符集的内容和排序行为。basic_string<>的字符串比较函数依赖于这些函数。[①]

这就是为什么 string 类没有提供不区分大小写的成员函数的原因，这本来就不是它的职责所在。要改变 string 类执行字符比较的方式，必须提供一个不同的 char_traits<>模板，后者定义了对单个字符进行比较的成员函数的行为。

利用刚学到的这些知识，我们可以创建一个新的 string 类来忽略大小写。首先定义一个新的、不区分大小写的 char_traits<>模板，它继承自现有模板。接着，重写必要的成员函数来执行不区分大小写的字符比较。（除了前面提到的三个字符比较成员函数之外，本例还会为 char_traits 中的 find()和 compare()函数提供新的实现。）最后，我们将基于 basic_string 来 typedef 一个新类，但是将它的第二个实参设置为不区分大小写的 ichar_traits 模板。

```
//: C03:ichar_traits.h
// 创建自定义的字符 trait
#ifndef ICHAR_TRAITS_H
#define ICHAR_TRAITS_H
#include <cassert>
#include <cctype>
#include <cmath>
#include <cstddef>
#include <ostream>
#include <string>
#include <initializer_list>  // 缺少的头文件

using std::allocator;
using std::basic_string;
using std::char_traits;
using std::ostream;
using std::size_t;
using std::string;
using std::toupper;
using std::tolower;

struct ichar_traits : char_traits<char> {
    // 我们只更改字符比较函数
    static bool eq(char c1st, char c2nd) {
```

[①] 译注：一些人将 traits 翻译为“特性”，本书选择保留原文。简单地说，traits 参数指定了一个字符串类具体应该如何处理字符。

```
        return toupper(c1st) == toupper(c2nd);
    }
    static bool ne(char c1st, char c2nd) {
        return !eq(c1st, c2nd);
    }
    static bool lt(char c1st, char c2nd) {
        return toupper(c1st) < toupper(c2nd);
    }
    static int compare(const char* str1, const char* str2, size_t n) {
        for(size_t i = 0; i < n; ++i) {
            if(str1 == 0)
                return -1;
            else if(str2 == 0)
                return 1;
            else if(tolower(*str1) < tolower(*str2))
                return -1;
            else if(tolower(*str1) > tolower(*str2))
                return 1;
            assert(tolower(*str1) == tolower(*str2));
            ++str1; ++str2; // Compare the other chars
        }
        return 0;
    }
    static const char*
    find(const char* s1, size_t n, char c) {
        while(n-- > 0)
            if(toupper(*s1) == toupper(c))
                return s1;
            else
                ++s1;
        return 0;
    }
};

typedef basic_string<char, ichar_traits> istring;

inline ostream& operator<<(ostream& os, const istring& s) {
    return os << string(s.c_str(), s.length());
}
#endif // ICHAR_TRAITS_H ///:~
```

本例提供了一个名为 istring 的 typedef。除了所有比较都不区分大小写之外，该类的工作方式在其他所有方面都与普通 string 无异。为了方便编程，我们还提供了一个重载的 operator<<()，以便向控制台打印 istring。下面展示了一个例子：

```
//: C03:ICompare.cpp
// 使用 istring 进行不区分大小写的比较
#include <cassert>
#include <iostream>
#include "ichar_traits.h"
```

```
using namespace std;

int main() {
    // 相同的字母，不同的大小写
    istring first = "tHis";
    istring second = "ThIS";
    cout << first << endl;
    cout << second << endl;
    assert(first.compare(second) == 0);
    assert(first.find('h') == 1);
    assert(first.find('I') == 2);
    assert(first.find('x') == string::npos);
} ///:~
```

这只是一个简单的例子。为了使 istring 完全等同于 string，我们需要创建支持新的 istring 类型的其他必要函数。

<string>头文件通过以下 typedef 提供了一个宽字符串类：

```
typedef basic_string<wchar_t> wstring;
```

对宽字符的支持还体现在宽流（wostream 取代了 ostream，前者也在<iostream>头文件中定义）和头文件<cwctype>中，后者是<cctype>的宽字符版本。配合使用标准库中 char_traits 的 wchar_t 特化版本，我们就可以创建宽字符版本的 ichar_traits 了，如下所示：

```
//: C03:iwchar_traits.h {-g++}
// 创建自定义的宽字符 trait
#ifndef IWCHAR_TRAITS_H
#define IWCHAR_TRAITS_H
#include <cassert>
#include <cmath>
#include <cstddef>
#include <cwctype>  // 为了使用像 toupper 这样的宽字符函数
#include <ostream>
#include <string>

using std::allocator;
using std::basic_string;
using std::char_traits;
using std::size_t;
using std::towlower;
using std::towupper;
using std::wostream;
using std::wstring;

struct iwchar_traits : char_traits<wchar_t> {
    // 我们只更改字符比较函数
    static bool eq(wchar_t c1st, wchar_t c2nd) {
```

```
        return towupper(c1st) == towupper(c2nd);
    }

    static bool ne(wchar_t c1st, wchar_t c2nd) {
        return towupper(c1st) != towupper(c2nd);
    }

    static bool lt(wchar_t c1st, wchar_t c2nd) {
        return towupper(c1st) < towupper(c2nd);
    }

    static int compare(const wchar_t* str1, const wchar_t* str2, size_t n) {
        for (size_t i = 0; i < n; ++i) {
            if (str1 == 0) {
                return -1;
            } else if (str2 == 0) {
                return 1;
            } else if (towlower(*str1) < towlower(*str2)) {
                return -1;
            } else if (towlower(*str1) > towlower(*str2)) {
                return 1;
            }
            assert(towlower(*str1) == towlower(*str2));
            ++str1;
            ++str2; // Compare the other wchar_ts
        }
        return 0;
    }

    static const wchar_t* find(const wchar_t* s1, size_t n, wchar_t c) {
        while (n-- > 0) {
            if (towupper(*s1) == towupper(c)) {
                return s1;
            } else {
                ++s1;
            }
        }
        return 0;
    }
};

typedef basic_string<wchar_t, iwchar_traits> iwstring;

inline wostream& operator<<(wostream& os, const iwstring& s) {
    return os << wstring(s.c_str(), s.length());
}

#endif // IWCHAR_TRAITS_H ///:~
```

可以看出，在适当位置加上一个“w”，即可实现源代码的“改头换面”。测试程序如下所

示：

```
//: C03:IWCompare.cpp {-g++}
// 使用 iwchar_traits 进行宽字符比较
#include <cassert>
#include <iostream>
#include "iwchar_traits.h"
using namespace std;

int main() {
    // 相同的字母，不同的大小写
    iwstring wfirst = L"tHis";
    iwstring wsecond = L"ThIS";
    wcout << wfirst << endl;
    wcout << wsecond << endl;
    assert(wfirst.compare(wsecond) == 0);
    assert(wfirst.find('h') == 1);
    assert(wfirst.find('I') == 2);
    assert(wfirst.find('x') == wstring::npos);
} ///:~
```

遗憾的是，一些编译器对宽字符的支持仍然不佳。[①]

3.5 示例字符串应用程序

仔细阅读本书中的示例代码，会注意在包围着代码的注释中，存在着一些特殊的标记。利用这些标记，布鲁斯编写的一个 Python 程序可以将代码提取到文件中，并设置用于生成代码的 Makefile 文件。例如，行首双斜杠后跟一个冒号表明这是源文件的第一行。该行其余部分包含的信息则描述了文件名、位置以及是否应该仅编译，而不是生成一个完整的可执行程序。例如，在上一个示例程序中，第一行包含字符串 C03:IWCompare.cpp，表明应该将 IWCompare.cpp 这个文件提取到目录 C03 中。

源文件的最后一行包含三个斜杠后跟一个冒号和波浪线。如果第一行在冒号后紧跟一个感叹号，就表明源代码的第一行和最后一行（以后简称为“首尾行”）不应输出到文件中（这是为纯数据文件而设计的）。

布鲁斯开发的 Python 程序不仅仅能提取代码。如果在文件名后有一个“{O}”标记，那么

[①] 译注：在现代 C++中，情况已经发生了变化。主流编译器对宽字符的支持已经相当成熟。GCC、Clang、MSVC 等主流编译器都提供了对宽字符的良好支持，包括模板特化、字符转换、国际化支持等。特别是，现在可以使用 `char16_t` 和 `char32_t` 这两个类型来更明确地表示 16 位和 32 位宽字符。还可以使用 `u8`、`u` 和 `U` 前缀来表示 UTF-8、UTF-16 和 UTF-32 编码的字符串字面值。

其 Makefile 条目就会设置成仅编译文件，而不链接成可执行文件。（第 2 章的测试框架就是如此构建的。）为了将此类文件与其他源代码示例链接，在目标可执行程序的源文件中将包含一个“{L}”指令，如下所示：

```
//{L} ../TestSuite/Test
```

本节将展示一个完整的程序，它提取所有代码以便自己手动编译和检查。可以使用该程序将本书的书稿另存为一个文本文件（例如，TICV2.txt）[①]，并在命令行窗口中执行如下所示的命令来提取本书书稿中的所有代码：[②]

```
C:> extractCode TICV2.txt TheCode
```

该命令读取文本文件 TICV2.txt，并在 TheCode 目录下创建专门的子目录来存储各章的源代码文件。最终的目录树如下所示：

```
TheCode/
   C0B/
   C01/
   C02/
   C03/
   C04/
   C05/
   C06/
   C07/
   C08/
   C09/
   C10/
   C11/
   TestSuite/
```

下面展示了完整程序：

```
//: C03:ExtractCode.cpp {-edg} {RunByHand}
// 提取全书文稿中的代码
#include <cassert>
#include <cstddef>
```

[①] 注意，某些版本的 Microsoft Word 在将文档保存为文本时会错误地将单引号字符替换为扩展 ASCII 字符，即弯引号，这会导致编译错误。我们不知道为什么会发生这种情况。只需手动将弯引号′ 替换为直引号'即可。

[②] 译注：本书中文版配套代码的第 3 章文件夹中有一个示例 TICV2.txt 文件，其中包含了本书前 3 章的所有文本。建议创建一个临时文件夹，将 ExtractCode.cpp 和 TICV2.txt 都拷贝到其中。执行 `cl /EHsc ExtractCode.cpp` 生成程序。手动创建一个 TheCode 子目录。最后，执行 `extractCode TICV2.txt TheCode` 来观察效果。

```
#include <cstdio>
#include <cstdlib>
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

// 旧的非标准 C 头文件，为了使用 mkdir()
#if defined(__GNUC__) || defined(__MWERKS__)
#include <sys/stat.h>
#elif defined(__BORLANDC__) || defined(_MSC_VER) \
    || defined(__DMC__)
#include <direct.h>
#else
#error 不支持的编译器
#endif

// 检查目录是否存在，做法是
// 尝试在其内打开新文件进行输出。
bool exists(string fname) {
    size_t len = fname.length();
    if (fname[len-1] != '/' && fname[len-1] != '\\')
        fname.append("/");
    fname.append("000.tmp");
    ofstream outf(fname.c_str());
    bool existFlag = outf.is_open();
    if (outf) {
        outf.close();
        remove(fname.c_str());
    }
    return existFlag;
}

int main(int argc, char* argv[]) {
    // 检查是否指定了输入文件名
    if (argc == 1) {
        cerr << "用法: extractCode 文件 [目录]" << endl;
        exit(EXIT_FAILURE);
    }

    // 检查指定的输入文件是否存在
    ifstream inf(argv[1]);
    if (!inf) {
        cerr << "打开文件失败: " << argv[1] << endl;
        exit(EXIT_FAILURE);
    }

    // 检查是否提供了可选的输出目录
    string root("./"); // 默认为当前目录
    if (argc == 3) {
```

```
    // 检查指定的输出目录是否存在
    root = argv[2];
    if (!exists(root)) {
        cerr << "不存在此目录: " << root << endl;
        exit(EXIT_FAILURE);
    }
    size_t rootLen = root.length();
    if (root[rootLen-1] != '/' && root[rootLen-1] != '\\')
        root.append("/");
}

// 逐行读取输入文件，并且
// 检查代码定界符。
string line;
bool inCode = false;
bool printDelims = true;
ofstream outf;
while (getline(inf, line)) {
    size_t findDelim = line.find("//" "/:~");
    if (findDelim != string::npos) {
        // 输出最后一行并关闭文件
        if (!inCode) {
            cerr << "代码行匹配错误" << endl;
            exit(EXIT_FAILURE);
        }
        assert(outf);
        if (printDelims)
            outf << line << endl;
        outf.close();
        inCode = false;
        printDelims = true;
    } else {
        findDelim = line.find("//" ":");
        if (findDelim == 0) {
            // 检查'!'指令
            if (line[3] == '!') {
                printDelims = false;
                ++findDelim; // 为下一次搜索跳过'!'
            }
            // 提取子目录名称，如果有的话
            size_t startOfSubdir = line.find_first_not_of(" \t", findDelim+3);
            findDelim = line.find(':', startOfSubdir);
            if (findDelim == string::npos) {
                cerr << "缺少文件名信息\n" << endl;
                exit(EXIT_FAILURE);
            }
            string subdir;
            if (findDelim > startOfSubdir)
                subdir = line.substr(startOfSubdir, findDelim - startOfSubdir);
            // 提取文件名（应该有一个!）
```

```
                    size_t startOfFile = findDelim + 1;
                    size_t endOfFile = line.find_first_of("\t", startOfFile);
                    if (endOfFile == startOfFile) {
                        cerr << "缺少文件名" << endl;
                        exit(EXIT_FAILURE);
                    }
                    // 现在已获得了所有信息，开始构建完整路径名称
                    string fullPath(root);
                    if (subdir.length() > 0)
                        fullPath.append(subdir).append("/");
                    assert(fullPath[fullPath.length()-1] == '/');
                    if (!exists(fullPath))
#if defined(__GNUC__) || defined(__MWERKS__)
                        mkdir(fullPath.c_str(), 0); // 创建子目录
#else
                        mkdir(fullPath.c_str()); // 创建子目录
#endif
                    fullPath.append(line.substr(startOfFile, endOfFile - startOfFile));
                    outf.open(fullPath.c_str());
                    if (!outf) {
                        cerr << "打开 " << fullPath << " 进行输出时失败" << endl;
                        exit(EXIT_FAILURE);
                    }
                    inCode = true;
                    cout << "正在处理 " << fullPath << endl;
                    if (printDelims)
                        outf << line << endl;
                } else if (inCode) {
                    assert(outf);
                    outf << line << endl; // 输出中间的代码行
                }
            }
        }
        exit(EXIT_SUCCESS);
    } ///:~
```

程序中用到了一些条件编译指令。在头文件<sys/stat.h>中定义的 mkdir()函数用于在文件系统中创建目录，这是 POSIX 标准[①]所规定的一个函数。遗憾的是，许多编译器仍然使用一个不同的头文件（<direct.h>）。它们还分别为 mkdir()使用了不同的签名：POSIX 规定有两个参数，而较旧的版本只有一个。因此，程序后面更多的条件编译是为了选择正确的 mkdir()调用。本书的示例一般不会使用条件编译，但这个程序太有用了，不花点额外

[①] POSIX 是一个 IEEE 标准，代表“可移植操作系统接口”（Portable Operating System Interface），是对 UNIX 系统中的许多底层系统调用的一般化。

的功夫是不行的，因为可以用它来提取散布于全书的所有代码。[①]

ExtractCode.cpp 中的 exists()函数通过在一个目录中打开临时文件来测试该目录是否存在。如果打开失败，则说明该目录不存在。可以将文件名作为 char*发送给 std::remove()来删除该文件。

main 程序验证用户提供的命令行实参，然后逐行读取输入文件，寻找特殊的源代码定界符。布尔标志 inCode 指出程序是否正处于源文件的中间部分，因此需要输出这些行。printDelims 指出源代码首行起始标记（即//:）后是否有一个感叹号。如果没有感叹号，那么该标志为 true，表明需要写入首尾行；否则就不写入。这里的重点在于需要先查找结束定界符（即///:~），因为起始标记是一个子集，首先查找起始标记会返回两种情况下的成功查找。[②]如果遇到结束标记，我们就验证是否正在处理源文件（inCode 是否为 true）；否则，文本文件中的定界符布局就存在问题（起始和结束标记顺序乱了）。如果 inCode 为 true，那么一切正常，并且（可选地）写入最后一行并关闭文件。当找到起始标记时，我们会解析目录和文件名并打开文件。本例中使用了以下和字符串相关的函数：length()、append()、getline()、find()的两个版本、find_first_not_of()、substr()、find_first_of()、c_str()以及当然还有 operator<<()。

3.6 小结

与 C 语言的字符数组相比，C++ string 对象为开发者提供了许多显著的优势。由于现在可以使用 string 类来引用字符串，所以基本上已经不再需要使用字符指针了。这有效避免了由于未初始化或指针值错误而引发的一整类软件 bug。

C++的 string 对象能够根据字符串数据的增长，动态且透明地扩展其内部存储空间。当字符串数据超过初始分配的内存限制时，string 对象会自动进行内存管理，从堆中分配额外空间，并在不再需要时释放这些空间。其一致的内存分配机制有效防止了内存泄漏，并且通常比手动实现的内存管理更加高效。

string 类成员函数提供了一整套工具来帮助我们创建、修改和搜索字符串。字符串比较总是区分大小写，但为了解决这个问题，我们可以采取以下做法：将字符串数据复制到 C 风格的空终止字符串并使用不区分大小写的字符串比较函数，将字符串对象容纳的数据临时转换成单一大写或小写，或者干脆自定义一个不区分大小写的字符串类（需重写用于创建

[①] 译注：现代 C++推荐使用文件系统库（filesystem）来创建目录，这是 C++17 引入的新特性。本书中文版配套代码提供了一个修改版的程序。请使用 cl /EHsc ExtractCode_CPP20.cpp /std:c++20 命令来编译，用 extractCode_CPP20 TICV2.txt TheCode 命令来运行（提前创建好 TheCode 目录）。

[②] 译注：即有无感叹号两种情况。

basic_string 的字符 traits）。

3.7 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. 编写并测试一个反转字符串中字符顺序的函数。

2. 回文是一个正反看都相同的词或句子。例如"madam"、"wow"或者"菜油炒油菜"。编写一个程序，从命令行接收一个字符串参数，并利用上个练习中的函数判断并输出该字符串是否为回文。

3. 修改练习 2 的程序，即使英语单词中对称字母的大小写不同也返回 true。例如，“Civic”尽管第一个字母是大写的，但仍然返回 true。

4. 修改练习 3 的程序，使其不仅忽略大小写，还忽略标点符号和空格。例如，"Able was I, ere I saw Elba."和"菜油炒油菜。"应该被认为是回文。

5. 使用以下字符串声明，并仅使用 char 数据类型（不允许使用字符串字面值或魔法数字[①]）：

```
string one("I walked down the canyon with the moving mountain bikers.");
string two("The bikers passed by me too close for comfort.");
string three("I went hiking instead.");
```

来生成以下句子：

```
I moved down the canyon with the mountain bikers. The mountain bikers passed by me too
close for comfort. So I went hiking instead.
```

6. 编写一个名为 replace 的程序，该程序接收三个命令行实参：输入文本文件、要替换的字符串（命名为 from）和替换字符串（命名为 to）。程序应该将输入文件中的所有 from 替换为 to，并将结果写入标准输出。

7. 重复上个练习，但忽略大小写来替换所有 from。

8. 修改练习 3 的程序，使其从命令行接收文件名，然后显示文件中所有不区分大小写的回文单词。不要显示重复的单词，即使它们的大小写不同。不要尝试查找大于单词长度的回

[①] 译注：魔法数字或者魔数是程序中突兀出现的一个或多个数字（数字字面值），不能一眼看出它们的含义。

文（与练习 4 不同）。[1]

9. 修改 3.4.3 节的 HTMLStripper.cpp 程序，使其在遇到 HTML 标记时显示标记的名称，然后显示标记和它的结束标记之间的内容。假设没有嵌套的标记，并且所有标记都有结束标记（形如</*TAGNAME*>）。

10. 编写一个程序，该程序接收三个命令行实参（文件名和两个字符串），并显示文件中同时包含两个字符串的行（Both）、包含任何一个字符串的行（Either）或者不包含任何字符串的行（Neither）。具体如何显示要取决于用户在程序开始时的输入（执行程序后，会询问用户要使用什么匹配模式）。除了 Neither 选项之外，对于任何匹配的字符串，都在字符串的开头和结尾使用星号（*）来突出显示它。

11. 编写一个程序，该程序接收两个命令行实参（一个文件名和一个字符串），并计算字符串在文件中出现的次数，无论它是不是子串（但是忽略重叠）。例如，字符串“ba”在单词“basketball”匹配两次，但字符串“ana”在单词“banana”中只匹配一次（因为发生了重叠）。在控制台中显示字符串在文件中匹配的次数，以及包含该字符串的单词的平均长度。（如果一个单词中字符串出现了多次，那么在计算平均长度时只计算一次。）

12. 编写一个程序，该程序从命令行接收一个文件名并分析字符使用情况，包括标点符号和空格（所有 ASCII 值从 0x21 [33] 到 0x7E [126]的字符，以及空格字符）。也就是说，统计每个字符在文件中出现的次数，然后根据用户在程序开始时的输入，按自然顺序（空格，然后是!、"和#等）或按字符出现频率的升序或降序显示结果。对于空格，请显示单词“Space”而不是字符' '。下面展示了一次示例运行结果：

```
Format sequentially, ascending, or descending (S/A/D): D
t: 526
r: 490
...
```

13. 使用 `find()`和 `rfind()`编写一个程序，该程序接收两个命令行实参（一个文件名和一个字符串），并显示不匹配该字符串的第一个和最后一个单词（及其索引），以及字符串首次和最后一次出现的索引。如果未找到字符串的任何实例，则显示“Not Found”。

14. 使用 `find_first_of` 系列函数（和其他函数）编写一个程序，从文件中移除所有非字母和数字字符（空格和句点除外），并将每个句点后的第一个字母大写。

15. 再次使用 `find_first_of` 系列函数编写一个程序，该程序从命令行接收一个文件名实参，并将文件中的所有数字格式化为货币形式。处理一个数字序列时，只考虑第一个小数点及其后面的数字，直到遇到非数字字符（如字母或特殊符号）为止。如果在这个数字序

[1] 译注：做这个练习时，就不要处理中文了。

列中有其他小数点，它们应该被忽略。将提取出来的数字序列四舍五入到最接近的百分位。例如，字符串"12.399abc29.00.6a"将按照美国货币标准格式化为"$12.40abc$29.01a"。

16. 编写一个程序来接收两个命令行实参（一个文件名和一个数字），并对文件中的每个单词进行打乱处理。具体做法是随机交换单词中的两个字母，交换的次数由第二个命令行实参指定。如果第二个实参是 0，则单词不被打乱；如果是 1，则随机选择一对字母进行交换；如果是 2，则随机选择两对字母进行交换，以此类推。

17. 编写一个程序从命令行接收一个文件名，并显示文件中的句子数（有多少个句点，就有多少个句子）、每一句的平均字符数以及文件中的总字符数。

18. 证明 at() 成员函数确实会在尝试越界访问时抛出异常，而索引操作符（[]） 不会。

第 4 章 iostream

除了将标准 I/O 封装成一个类（例如，cin 和 cout），还可以利用 iostream 库实现更全面的 I/O 操作。[①]

如果能使所有常用的“容器”——标准 I/O、文件甚至内存块——看起来都一样，从而只需要记住一个接口，那么不是很“酷”吗？这正是 iostream 库背后的设计理念。这些输入输出流比标准 stdio 库中的各种乱七八糟的函数更容易使用、更安全，有时甚至更高效。

在学习 C++时，iostream 往往是初学者接触的第一个库。本章将解释为什么说 iostream 比传统 C 语言中的 stdio（标准输入输出）库更优秀。另外，除了讨论控制台输入输出，我们还会深入探讨文件流（用于处理文件）和字符串流（用于处理字符串数据）的特性及使用方法。

4.1 iostream 的设计宗旨

一些人可能会觉得奇怪，好好的 C 库为什么不继续用呢？为何不把 C 库包装在一个类里就完事了呢？有的时候，这确实一个不错的解决方案。例如，假设想确保一个 stdio FILE 指针所代表的文件总是被安全地打开并且正确地关闭，而不依赖于用户记得调用 close()函数，那么可以像下面这样尝试：

```
//: C04:FileClass.h
// 封装了 stdio FILE 指针
#ifndef FILECLASS_H
#define FILECLASS_H
#include <cstdio>
#include <stdexcept>

class FileClass {
    std::FILE* f;
public:
    struct FileClassError : std::runtime_error {
        FileClassError(const char* msg)
            : std::runtime_error(msg) {}
};

FileClass(const char* fname, const char* mode = "r");
~FileClass();
std::FILE* fp();
```

[①] 译注：本书在提到 iostream 库时会保留原文，而在提到各种输入输出流时会使用中文。

```
}; #endif // FILECLASS_H ///:~
```

在 C 中执行文件 I/O 时，使用的是指向 FILE 结构体的裸指针[①]。但是，这个类封装了指针，并使用构造函数和析构函数来确保它被正确地初始化和清理。第二个构造函数实参指定了文件模式，默认为"r"，代表"read"（只读）。

要获取指针的值以便在文件 I/O 函数中使用，可以使用 fp()这个取值函数（fp 代表“文件指针”）。以下是各个成员函数的定义：

```
//: C04:FileClass.cpp {O}
// FileClass 类的实现
#include "FileClass.h"
#include <cstdlib>
#include <cstdio>
using namespace std;

FileClass::FileClass(const char* fname, const char* mode) {
    if((f = fopen(fname, mode)) == 0)
        throw FileClassError("打开文件时出错");
}

FileClass::~FileClass() { fclose(f); }

FILE* FileClass::fp() { return f; } ///:~
```

构造函数调用 fopen()，这和平时没什么两样，但它还确保了结果不为零（零表示打开文件失败）。如果文件没有按预期打开，那么会抛出异常。

析构函数关闭文件，取值函数 fp()返回私有数据成员 f（即文件指针）。下面是一个使用了 FileClass 的简单示例：[②]

```
//: C04:FileClassTest.cpp
//{L} FileClass
#include <cstdlib>
#include <iostream>
#include "FileClass.h"
using namespace std;

int main() {
    try {
      FileClass f("FileClassTest.cpp");
```

[①] 译注：裸指针是相对于智能指针而言的。裸指针要求程序员手动管理分配的内存。

[②] 译注：编译命令是 cl /EHsc FileClassTest.cpp FileClass.cpp。运行这个程序会显示指定文件的全部内容。

```
      const int BSIZE = 100;
      char buf[BSIZE];
      while(fgets(buf, BSIZE, f.fp()))
          fputs(buf, stdout);
    } catch(FileClass::FileClassError& e) {
      cout << e.what() << endl;
      return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
} // 文件由析构函数自动关闭
///:~
```

本例创建 FileClass 对象，并用它在标准的 C 文件 I/O 函数调用中调用 fp()。用完后便不用管了，文件会在其作用域结束时由析构函数自动关闭。

注意，即使 FILE 指针是私有的，它也不是特别安全，因为 fp()可以读取它。由于唯一的作用似乎就是保证初始化和清理，所以是不是可以将其设为public，或者干脆使用结构体呢？答案是否定的。注意，虽然可以使用 fp()获取 f 的拷贝，但不能向 f 赋值——这完全由类控制。但是，在捕获fp()返回的指针之后，客户端程序员仍然可以对结构体元素进行赋值甚至关闭它。因此，为了获得真正的安全性，我们的重点在于确保 FILE 指针始终有效，而不在于保证结构体元素的正确性。

要想获得真正的安全性，就必须阻止用户直接访问 FILE 指针。[1]在下面这个示例“完全封装”类中，所有标准文件 I/O 函数的某个版本都应该作为类的成员出现，从而确保这个C++类提供与 C 语言相同的一套文件操作功能。

```
//: C04:Fullwrap.h
// 本例完全隐藏了文件 I/O
#ifndef FULLWRAP_H
#define FULLWRAP_H
#include <cstddef>
#include <cstdio>

// 取消对标准库中这些宏的定义，
// 是为了避免与自定义的同名函数产生冲突。
#undef getc
#undef putc
#undef ungetc

using std::size_t;
using std::fpos_t;

class File {
```

[1] 译注：换言之，要完全隐藏文件 I/O。

```
    std::FILE* f;
    std::FILE* F(); // 该函数返回经过检查的 f 指针
public:
    File(); // 创建对象，但不打开文件
    File(const char* path, const char* mode = "r");
    ~File();
    int open(const char* path, const char* mode = "r");
    int reopen(const char* path, const char* mode);
    int getc();
    int ungetc(int c);
    int putc(int c);
    int puts(const char* s);
    char* gets(char* s, int n);
    int printf(const char* format, ...);
    size_t read(void* ptr, size_t size, size_t n);
    size_t write(const void* ptr, size_t size, size_t n);
    int eof();
    int close();
    int flush();
    int seek(long offset, int whence); // whence 不是拼写错误，代表从哪里开始 seek
    int getpos(fpos_t* pos);
    int setpos(const fpos_t* pos);
    long tell();
    void rewind();
    void setbuf(char* buf);
    int setvbuf(char* buf, int type, size_t sz);
    int error();
    void clearErr();
};
#endif // FULLWRAP_H ///:~
```

这个类包含了<cstdio>中的几乎所有文件I/O函数，只是缺少vfprintf()；[1]但这个类实现了printf()成员函数来提供类似的功能。

File类包含与之前的例子相同的构造函数，同时新增了一个默认构造函数。如果想创建一个由File对象构成的数组，或者想将File对象用作另一个类的成员，但那个类的初始化不是在构造函数中发生，而是在对象创建之后发生，那么默认构造函数就相当重要了。

默认构造函数将私有FILE指针f设为零。但现在每次引用f时，都会检查它的值是否为零。这是通过F()函数来实现的，F()是私有的，因为它设计为只由其他成员函数使用。我们不想让用户直接访问底层的FILE结构体。

这个解决方案其实还算不错。它相当实用，而且完全可以创建相似的类来处理标准（控制

[1] 译注：vfprintf()涉及可变参数列表，实现起来比较复杂。

台）I/O 和内存块格式化（in-core formatting，即向内存块而不是文件或控制台进行读/写）。

然而，在处理可变参数的函数时，这个方案存在一些问题。这是因为此类函数依赖于运行时解释器。解释器在运行时解析格式字符串，并根据它来处理可变参数。这种方式存在以下几个弊端：

1. **代码膨胀：**即便只用到了解释器功能的一小部分，也需要将整个解释器加载到可执行体中。因此，如果使用 `printf("%c", 'x');`，那么将获得整个包，其中包括打印浮点数和字符串的部分。没有任何一个标准选项可以帮助我们减少程序占用的空间。
2. **性能开销：**由于解释在运行时发生，因此性能开销无法避免。这颇令人沮丧，因为所有信息在编译时都通过格式字符串确定好了，但直到运行时才被求值。然而，如果可以在编译时解析格式字符串中的实参，那么就能直接进行函数调用，这些调用可能比运行时解释器快得多（尽管 `printf()`系列函数的性能一般已经非常优化了）。
3. **缺乏编译时检查：**由于格式字符串直到运行时才会求值，因此无法进行编译时的错误检查。如果曾尝试过排查因为在 `printf()`语句中使用了数量或类型有误的实参而造成的 bug，那么必然不会对此感到陌生。C++花费大力气完善编译时错误检查，以便尽早发现错误，并让程序员的工作更轻松。所以，作为一个 I/O 库，放弃类型安全似乎颇为可惜，尤其是考虑到几乎所有程序都要执行繁重的 I/O 操作。
4. **扩展性差。**对于C++，最关键的问题是`printf()`系列函数的扩展性不佳。它们实际上只设计用于处理 C 中的基本数据类型（`char`、`int`、`float`、`double`、`wchar_t`、`char*`、`wchar_t*`和 `void*`）及其变体。有人或许认为，每次添加一个新类时，都可以添加重载的 `printf()`和 `scanf()`函数（以及它们用于文件和字符串的变体）。但请记住，只有在参数列表中写明了不同的类型，才可算得上对函数的“重载”，而 `printf()`系列函数将其类型信息隐藏在格式字符串和可变参数列表中。对于像 C++这样的语言，其目标是能轻松地添加新的数据类型。因此，这个限制令人不可接受。

4.2 iostream 来救场

综上所述，输入输出（I/O）是 C++标准类库的首要关注点之一。由于“Hello, World”是几乎是每个人在学习一门新语言时编写的第一个程序，而且几乎所有程序都需要执行 I/O 操作，因此在 C++中，I/O 库必须设计得特别容易使用。除此之外，它还面临着一个更大的挑战，那就是必须适应任何新的类。因此，这些约束条件要求这个基础类库的设计必须足够出色。本章除了清楚地描述 I/O 和格式化方面的显著优势，还会顺带展示一个功能强大的 C++库应该是如何工作的。

4.2.1 插入符和提取符

流（stream）本质上是一种传输和格式化固定宽度（定宽）字符的对象。我们可以使用输入流（通过 `istream` 类的派生类）、输出流（使用 `ostream` 对象）或者同时进行输入和输

出的流（使用从 iostream 派生的对象）。iostream 库还提供了上述类型的派生类，其中包括用于文件 I/O 的 ifstream、ofstream 和 fstream，以及用于与 C++标准 string 类交互的 istringstream、ostringstream 和 stringstream。所有这些**流类**都有几乎相同的接口，因此无论处理的是文件、标准 I/O、内存区域还是 string 对象，都可以采取统一的方式来使用流。在本章中学习的统一接口不仅适用于现有的类，还适用于未来为支持新类而添加的扩展。[①]其中，有些函数可以实现我们的格式化命令，而有些函数可以直接读取和写入字符，不涉及格式化。

前面提到的流类实际上是模板的**特化**，[②]就像标准 string 类是 basic_string 模板的特化一样。图 4.1 展示了 iostream 继承层次结构中的基本类。

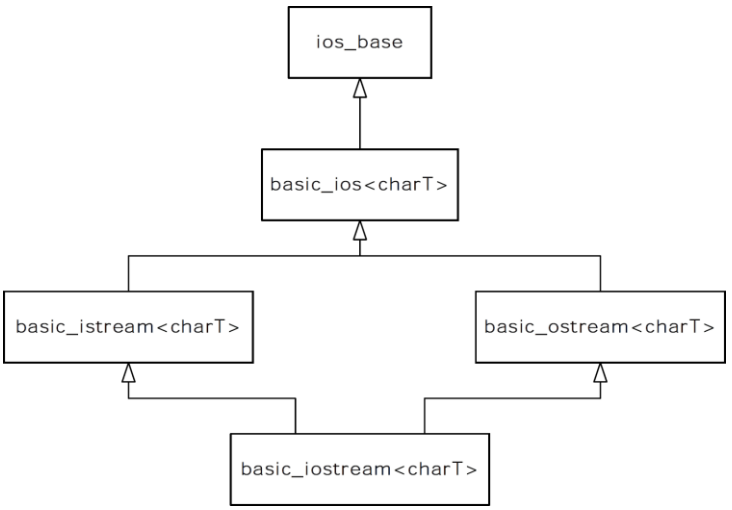

图 4.1 C++ iostream 继承层次结构中的基本类

其中，ios_base 类声明了所有流共同的东西，这些东西不依赖于流所处理的字符类型。这些声明主要用于管理相关的常量和函数，其中一些将在本章讲述。图中的其他类都是模板，以底层的字符类型作为实参。例如，istream 类就是基于其中一个模板定义的：

```
typedef basic_istream<char> istream;
```

前面提到的类都是通过相似的类型定义来定义的。还有一些流类使用 wchar_t（即第 3 章讨论过的宽字符类型）代替了 char，具体将在本章最后讨论。basic_ios 模板定义了通用

[①] 译注：例如，可以重载<<操作符来定义任意复杂的数据类型的输出格式。这就是一种“扩展”。

[②] 第 5 章将更深入地探讨模板的“特化”。

于输入和输出的函数，但这些函数需要依赖于底层的字符类型（我们不会过多地使用这些）。basic_istream 模板定义了泛型输入函数，而 basic_ostream 定义了泛型输出函数。稍后讨论的文件和字符串流类为各自具体的流类型增加了功能性。

iostream 库重载了两个操作符以简化各种输入输出流的使用。其中，<<通常称为输入输出流的**插入符**（inserter），而>>通常称为**提取符**（extractor）。

提取符根据目标对象的类型解析期望的信息。为了体会这样的一个例子，可以尝试使用一下 cin 对象，它是 C stdin 的等价物，也就是“可重定向的标准输入”。该对象在包含<iostream>头文件时就已经预定义好了。

```
int i;
cin >> i;

float f;
cin >> f;

char c;
cin >> c;

char buf[100];
cin >> buf;
```

每种内置数据类型（例如，int）都重载了>>操作符。当然也可以为自己的类型重载该操作符，稍后就讲解具体如何做。

为了查看各种变量中的内容，我们可以使用 cout 对象（对应于标准输出；另外还有一个 cerr 对象对应于标准错误）和插入符<<。

```
cout << "i = ";
cout << i;
cout << "\n";
cout << "f = ";
cout << f;
cout << "\n";
cout << "c = ";
cout << c;
cout << "\n";
cout << "buf = ";
cout << buf;
cout << "\n";
```

但这样写可就太啰嗦了，而且并不见得比 printf()好——虽然类型检查确实有所增强。幸好，重载的插入符和提取符经过了精心设计，可以链接成更复杂的表达式，这样不仅写起来更容易，可读性也更好了，如下所示：

```
cout << "i = " << i << endl;
cout << "f = " << f << endl;
```

```
cout << "c = " << c << endl;
cout << "buf = " << buf << endl;
```

要为自己的类定义插入符和提取符，只需重载相应的操作符即可，具体步骤如下所示：

1. 将第一个参数设为流的非 const 引用（istream 用于输入，ostream 用于输出）。
2. 通过处理对象的各个组成部分，将数据插入流或者从流中提取。
3. 返回对流的引用。

流应该是非 const 的，因为处理流数据会改变流的状态。如上例所示，通过返回流，可以在单个语句中链接不同的流操作。

下面来看一个例子，让我们考虑如何以 MM-DD-YYYY 这种格式输出 Date 对象。我们像下面这样重载插入符：

```
ostream& operator<<(ostream& os, const Date& d) {
    char fillc = os.fill('0');
    os << setw(2) << d.getMonth() << '-'
       << setw(2) << d.getDay() << '-'
       << setw(4) << setfill(fillc) << d.getYear();
    return os;
}
```

注意，不能将该函数作为 Date 类的一个成员，因为<<操作符的左操作数必须是输出流。ostream 类的 fill()成员函数会在输出域宽度（由 setw()操纵元确定）大于输出数据需要的宽度时更改填充字符。这里使用的是字符'0'，这样十月之前的月份会显示前导零，比如九月显示为“09”。fill()函数还会返回之前的填充字符（默认为单个空格），以便稍后使用 setfill()操纵元来恢复默认设定。本章稍后会深入讨论操纵元。[①]


```
ostream& operator<<(ostream& os, const Date& d) {
    char fillc = os.fill('0');
    os << setw(2) << d.getMonth() << '-'
       << setw(2) << d.getDay() << '-'
       << setw(4) << setfill(fillc) << d.getYear();
    return os;
}
```


定义提取符时需要更小心一些，因为输入数据可能会出现问题。为了标记流错误，一个方法是设置流的**失败标志位**（fail bit），如下所示：

[①] 译注：操纵元（manipulator）是以非传统方式调用的函数。调用操纵元后，它本身又会调用一个成员函数。操纵元位于插入符<<之后，好似它本身就是下一个要输出的项。和传统函数一样，操纵元可以有、也可以没有参数。setw、setfill 和 endl 等都是操纵元。另外，也可以将操纵元称为“操纵符”。——摘自《C++入门经典》第 10 版，清华大学出版社出版。

```
istream& operator>>(istream& is, Date& d) {
    is >> d.month;
    char dash;
    is >> dash;
    if (dash != '-') {
        is.setstate(ios::failbit);
    }
    is >> d.day;
    is >> dash;
    if (dash != '-') {
        is.setstate(ios::failbit);
    }
    is >> d.year;
    return is;
}
```

一旦流中设置了失败标志位，后续所有流操作都会被忽略，直至流恢复到良好状态（稍后解释）。这就是上述代码即使设置了 ios::failbit 也会继续提取数据的原因。[1]注意，这个实现在某种程度上放宽了限制，因为它允许日期字符串中的数字和短划线之间有空白（>> 操作符在读取内置类型时会默认跳过空白）。下面展示了这个提取符能正确处理的有效日期字符串：

```
"01-29-2025"
"1-29-2025"
"01 - 29 - 2025"
```

但下面这些日期字符串是无效的：

```
"A-10-2003"  // 不允许字母字符
"08%10/2003" // 只允许短划线作为分隔符
```

本章后面的 4.3 节“处理流错误”会详细讨论流的状态。

4.2.2 常规用法

正如 Date 提取符所展示的那样，我们必须警惕错误的输入。如果输入产生了一个意外的值，那么整个过程会被扭曲，并且很难恢复。此外，在默认情况下，格式化的输入以空白字符作为定界符。例如，假定将本章前面展示的代码片段整合成单一的程序：

```
//: C04:Iosexamp.cpp {RunByHand}
// iostream 示例
#include <iostream>
using namespace std;
```

---

[1] 后续流操作只是被忽略，并不是说不执行。即使设置了 failbit，后续流操作还是会执行，只是会失败。

```
int main() {
    int i;
    cin >> i;

    float f;
    cin >> f;

    char c;
    cin >> c;

    char buf[100];
    cin >> buf;

    cout << "i = " << i << endl;
    cout << "f = " << f << endl;
    cout << "c = " << c << endl;
    cout << "buf = " << buf << endl;

    cout << flush;
    cout << hex << "0x" << i << endl;
} ///:~
```

并在控制台向它提供以下输入：

```
12 1.4 c this is a test
```

您或许以为，像上面这样输入就好比像下面这样一行一行地输入，因此输出也一样：

```
12
1.4
c
this is a test
```

但实际的输出有些令人意外：

```
i = 12
f = 1.4
c = c
buf = this
0xc
```

注意，buf 只获得了第一个单词，因为输入例程会寻找作为定界符的空白字符，而它在“this”后面看到了一个空格（也可以是制表符等其他空白字符）。此外，如果连续的输入字符串长度超过为 buf 分配的存储空间，那么会造成缓冲区的溢出。

在实际应用程序中，我们通常希望一次一行地从交互式程序中获取输入作为字符序列，扫描它们，然后在安全地进入缓冲区后再执行转换。这样就不必担心输入例程在遇到意外数据时卡住。

另一个需要考虑的问题是**命令行界面**（CLI）的概念。在过去，当控制台只是一台打字机时，使用 CLI 是有道理的。但是，现在时代不同了，图形用户界面（GUI）占据了主导地位。在这种情况下，控制台 I/O 的意义何在？除了简单的示例或测试，我们似乎可以完全忽略 cin，并采取以下措施：

1. 如果程序需要输入，那么可以从文件中读取输入——很快就会看到，可以非常方便地使用输入输出流来操作文件。文件输入输出流在 GUI 环境下依然有效。
2. 就像刚才建议的那样，在不尝试转换的情况下逐行读取输入。当输入处于某个位置，确定转换不会导致问题时，再安全地扫描它。
3. 输出的情况则不一样。如果使用 GUI，那么 cout 可能无法直接显示输出，这时需要将输出发送到文件或者使用 GUI 提供的显示功能。否则，将输出发送到 cout 通常是合理的。无论哪种情况，iostream 的输出格式化功能都非常实用。

另一个常见的实践可以在大型项目中节省编译时间。例如，考虑如何在头文件中声明本章前面介绍的两个 Date 流操作符。由于只需包含函数原型，因此不需要在 Date.h 中包含整个<iostream>头文件。标准实践是仅声明类，如下所示：

```
class ostream;
```

这是一种古老的接口与实现分离技术，通常称为**前向声明**（即 forward declaration；此时，ostream 会被认为是一个**不完整类型**，因为编译器还没有看到类定义）。

然而，由于以下两个原因，这种做法并不能直接奏效：

1. 流类是在 std 命名空间中定义的。
2. 它们是模板类。

因此，正确的声明应该是：

```
namespace std {
    template<class charT, class traits = char_traits<charT>> class basic_ostream;
    typedef basic_ostream<char> ostream;
}
```

可以看出，和 string 类一样，流类也使用了第 3 章提到的字符 traits 类。由于为每个想要引用的流类键入所有这些代码显得非常繁琐，因此标准提供了一个头文件<iosfwd>[①]来帮助我们做这件事。下面展示了完善过的 Date 类头文件：[②]

```
// Date.h
```

[①] 译注：fwd 是“前向声明”的简称。

[②] 译注：完整的 Date.h 和 Date.cpp 请参见第 2 章和附录 B。

```
#include <iosfwd>

class Date {
public:
    friend std::ostream& operator<<(std::ostream&, const Date&);

    friend std::istream& operator>>(std::istream&, Date&);
    // 等等...
};
```

4.2.3 逐行输入

逐行提取输入时，我们有三个选择：

- 成员函数 get()，
- 成员函数 getline()，
- 以及在<string>头文件中定义的全局函数 getline()。

前两个函数支持接收三个实参：

1. 一个指针，它指向用于存储结果的字符缓冲区。
2. 缓冲区的大小（防止溢出）。
3. 终止符，以确定何时停止读取输入。

终止符默认为换行符'\n'，我们一般都使用这个。这两个函数在遇到终止符时会在结果缓冲区中存储一个零。

那么，这两个函数到底有什么区别？两者的区别虽然细微，但却很重要：get()在“看到”输入流中的定界符时停止，但不会从输入流中提取该定界符。因此，如果再次使用相同的定界符调用 get()，它会立即返回而不提取任何输入（因此，为了避免这个问题，可以在下一个 get()语句中使用不同的定界符或者不同的输入函数）。另一方面，getline()函数会从输入流中提取定界符，但仍然不会将其存储在结果缓冲区中。

在<string>中定义的 getline()函数非常好用。它不是成员函数，而是 std 命名空间中声明的一个独立函数。它只接收两个非默认实参：输入流和要填充的 string 对象。和它的同名函数一样，它读取字符直到遇到定界符（默认是'\n'），然后消耗并丢弃定界符。这个函数的优点在于，它会将数据读取到一个 string 对象中，因此不必担心缓冲区大小的问题。

需要逐行处理文本文件时，我们通常都会用到上述某个 getline()函数。

get()的重载版本

get()函数还有其他三个重载版本：一个无参，使用 int 返回值来返回下一个字符；一个

使用引用将字符填充到其 char 实参中；另一个则直接存储到另一个 iostream 对象的底层缓冲区结构中。本章稍后会探讨最后一个重载。

读取原始字节

如果确切地知道自己正在处理的内容，并且想将字节直接移动到一个变量、数组或内存中的结构体中，那么可以使用未格式化的 I/O 函数 read()。该函数的第一个实参是指向目标内存的指针，第二个是要读取的字节数。如果之前已将信息存储到文件中，例如使用输出流对应的 write()成员函数以二进制形式存储，那么该函数就特别有用了（当然，要使用相同的编译器）。稍后会展示所有这些函数的示例。[①]

4.3 处理流错误

前面展示的 Date 提取符在特定条件下会设置流的失败标志位。那么，用户如何知道发生了失败呢？可以通过调用某些流成员函数来检测是否发生了错误状态。或者，如果不关心具体的错误是什么，那么可以在布尔上下文中对流进行求值。这两种技术都源于流错误标志的状态。

4.3.1 流状态

从 ios 派生的 ios_base 类定义了 4 个标志位[②]，可以用它们来测试流的状态。表 4.1 对此进行了总结。

表 4.1 流状态标志位[③]

标志	含义
badbit	发生了一些致命的（可能是物理上的）错误。该流应被视为不可用。
eofbit	发生了“输入结束”（要么是因为遇到了文件流的物理末尾，要么是因为用

[①] 译注：本书中文版配套代码在第 4 章目录下提供了一个 ReadWriteBinaryFile.cpp 程序，演示了如何在一个名为 data.bin 的文件中存取二进制数据。

[②] 由于是从 ios 派生的，所以可以写 ios::failbit 而不是 ios_base::failbit，从而少打几个字。

[③] 译注：在实际应用中，多个标志位可能会同时被设置。例如，当达到文件末尾时，eofbit 和 failbit 都会被设置。检查流状态时，我们一般使用 rdstate()成员函数来获取所有标志位的状态。

	户终止了控制台流，例如使用 Ctrl-Z 或 Ctrl-D）。
failbit	一个 I/O 操作失败，很可能是因为无效的数据（例如，在尝试读取数字时发现了字母）。流仍然可用。当发生“输入结束”时，也会设置 failbit 标志。
goodbit	一切正常；没有错误。输入尚未结束。

可以调用相应的、返回布尔值的成员函数来检测是否发生了这些情况。流成员函数 good()在其他三个标志位都没有被设置时返回 true。eof()函数在 eofbit 被设置时返回 true；如果尝试从没有更多数据的流（通常是文件）中读取，就会发生这种情况。由于在试图读取物理介质末尾之外的数据时会发生“输入结束”，因此 failbit 也会被同时设置，以指示“预期”的数据未成功读取。fail()函数在 failbit 或 badbit 被设置时返回 true，而 bad()函数仅在 badbit 被设置时才返回 true。[①]

一旦流状态中的任何一个错误标志被设置，它们就会一直保持设置状态。但是，这并不总是我们想要的结果。在读取文件时，我们可能希望在发生“输入结束”之前重新定位到文件某个早期位置。但是，仅仅移动文件指针并不会自动重置 eofbit 或 failbit。在这种情况下，需要自己使用 clear()函数来重置它们，如下所示：

```
myStream.clear(); // 清除所有错误标志位，使流恢复正常状态
```

调用 clear()之后，如果立即调用 good()，它将返回 true。正如在之前的 Date 提取符中看到的，setstate()函数设置了传递给它的标志位。事实证明，setstate()不会影响其他标志位。换言之，如果它们已经被设置，那么会保持设置状态。如果想设置某些标志位，同时重置其他所有标志位，那么可以调用 clear()的重载版本，并传入一个代表想要设置的标志位的按位表达式，例如：

```
myStream.clear(ios::failbit | ios::eofbit);
```

大多数时候，我们对单独的流状态标志位没有兴趣。相反，只是想知道是否一切正常。例如，从头到尾读取一个文件时，只想知道输入数据何时耗尽。在这种情况下，可以使用为 void*定义的一个转换函数。当流出现在布尔表达式中时，就会自动调用该函数。[②]下例展示了如何使用这种“惯用法”（idiom）来一直读取流直至“输入结束”：

```
int i;
while (myStream >> i)
    cout << i << endl;
```

[①] 译注：将标志位设为 1，就称为“设置”了这个标志位。“清除”一个标志位意味着把它设为 0。

[②] 译注：当流出现在布尔表达式中时，编译器会尝试将其转换为布尔值。这个转换通常是通过检查流的内部状态来实现的。

记住，operator>>()返回其流实参，所以上述 while 语句是将流作为一个布尔表达式来测试。这个特定的例子假设输入流 myStream 包含由空白字符分隔的整数。ios_base::operator void*()函数直接在其流上调用 good()并返回结果。[①]由于大多数流操作都返回它们自身的流，因此这种惯用法虽然有点“非标准”，但还是挺好用的。

4.3.2 流和异常

输入输出流作为 C++的一部分早已存在，远早于异常机制的引入。因此，许多人早已习惯了手动检查流状态。为了向后兼容，现在仍然支持这种做法。但是，现代的输入输出流可以改为抛出异常，而无需手动检查。流成员函数 exceptions()接收代表异常状态标志位的一个参数[②]，这些标志位代表希望抛出的异常。每当流遇到指定的异常状态时，就会抛出一个 std::ios_base::failure 类型的异常，该异常继承自 std::exception。

虽然可以为表 4.1 的 4 种流状态中的任何一种触发“失败异常”，但为所有状态都启用异常并不是一个好主意。如第 1 章所述，异常应该用来处理真正“异常”的情况，而 eof（到达文件尾）并不能算是“异常”——它是预料之中的！因此，通常只需要为 badbit 所表示的错误启用异常，如下所示：

```
myStream.exceptions(ios::badbit);
```

可以为每个流分别启用异常，因为 exceptions()是流的成员函数。exceptions()函数返回一个位掩码[③]（类型为 iostate，它可以转换为成 int，但具体如何转换要取决于编译器），表明哪些流状态会导致异常。如果这些状态已被设置，那么会立即抛出异常。当然，如果在处理流时使用异常，那么最好准备着捕获它们，这意味着需要用一个配有 ios::failure 异常处理程序的 try 块来包围所有流操作。许多程序员发现这样做很繁琐，因此他们在预期可能出错的地方手动检查状态（例如，他们可能认为大多数时候 bad()都不会返回 true[④]）。这也是为什么让流抛出异常是可选的而非默认行为的另一个原因。无论如何，我们可以自行选择如何处理流错误。和其他情况一样，我们出于同样的原因也推荐在这里使

[①] 习惯上优先使用 operator void*()而不是 operator bool()，这是因为从 bool 到 int 的隐式转换可能会引发意外。例如，可能错误地将流对象放到了一个可以应用整数转换的上下文中。相反，operator void*()函数只有在布尔表达式主体中才会被隐式调用。

[②] 译注：参数是一个整数，由多个异常状态标志位通过“按位 OR”运算得到。

[③] “位掩码”（bitmask）是一个特殊的整型，专门用于容纳由单独的位表示的“标志”。换言之，每一位都可以看作是一个开关，0 表示关闭，1 表示开启。不同的位对应不同的标志。

[④] 译注：许多程序员在使用流时，并不认为会经常遇到严重错误（由 bad()函数返回 true 表示）。因此，他们认为手动检查流状态就足够了，而不需要使用异常处理机制。

用异常来处理错误。

4.4 文件输入输出流

使用 iostream 中的类来操控文件比使用 C 语言中的 stdio 要容易和安全得多。打开一个文件只需创建一个对象——剩下的事情构造函数会负责。不需要显式关闭文件（非要手动关闭也可以使用 close()成员函数），因为当对象超出作用域时，析构函数会自动关闭它。要创建一个默认用于输入的文件，可以创建一个 ifstream 对象。要创建一个默认用于输出的文件，可以创建一个 ofstream 对象。fstream 对象则可以同时进行输入和输出。

图 4.2 展示了各种文件流类（名称中带 f 的）在 iostream 继承层次结构中的关系。

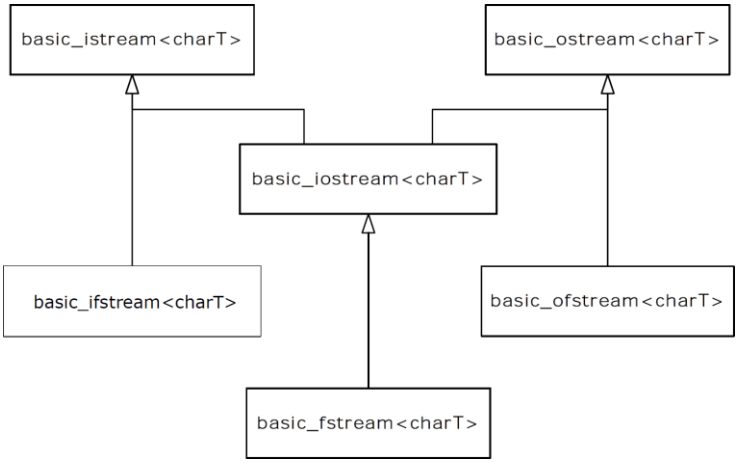

图 4.2 C++ iostream 继承层次结构中的文件流类

和往常一样，我们使用的类其实是模板的“特化”。例如，用于处理由 char 构成的文件的 ifstream 实际是这样定义的：

```
typedef basic_ifstream<char> ifstream;
```

4.4.1 文件处理示例

下例展示了到目前讨论的许多特性。注意，包含<fstream>是为了使用其中声明的文件 I/O 类。另外，虽然许多平台都会在包含<fstream>后自动包含<iostream>，但并没有强制编译器这样做。因此，为了确保代码的可移植性，请始终包含这两个头文件。

```
//: C04:Strfile.cpp
// 演示文件流 I/O，
// 并演示 get()和 getline()之间的区别。
#include <fstream>
```

```
#include <iostream>
#include "../require.h"
using namespace std;

int main() {
    const int SZ = 100; // 指定缓冲区大小
    char buf[SZ];
    {
        ifstream in("Strfile.cpp");  // 读取
        assure(in, "Strfile.cpp");   // 验证是否打开
        ofstream out("Strfile.out"); // 写入
        assure(out, "Strfile.out");  // 验证是否打开
        int i = 1;                   // 行号计数器

        // 较为不便的逐行输入：
        while (in.get(buf, SZ)) { // 在输入中保留\n
            in.get();             // 丢弃下一个字符（\n）
            cout << buf << endl;  // 必须添加\n
            // 文件输出和标准 I/O 无异：
            out << i++ << ": " << buf << endl;
        }
    } // 析构函数关闭 in 和 out

    ifstream in("Strfile.out");
    assure(in, "Strfile.out");
    // 较为方便的逐行输入：
    while (in.getline(buf, SZ)) { // 移除\n
        char* cp = buf;
        while (*cp != ':')
            ++cp;
        cp += 2; // 跳过": "
        cout << cp << endl; // 还是必须要添加\n
    }
} ///:~
```

在创建了 ifstream 和 ofstream 对象后，都分别紧随着一个 assure()函数调用，其目的是保证文件成功打开。这里再次强调，在编译器期望布尔结果的情况下，对象会生成一个指示成功或失败的值。

第一个 while 循环演示了两种形式的 get()函数的用法。第一种形式将字符读入缓冲区，并在读取了 SZ-1 个字符或者遇到第三个实参（默认为换行符'\n'）时在缓冲区中插入零（'\0'）终止符。get()函数将终止符保留在输入流中，因此必须通过 in.get()丢弃该终止符，这是 get()的无参版本，它读取一个字节并作为 int 返回。除此用这种方法来丢弃终止符，还可以考虑使用 ignore()成员函数。该函数有两个默认参数。第一个参数是要丢弃的字符数量，默认为 1。第二个参数是在遇到后便退出 ignore()函数的字符，默认为 EOF。

接着是两个看起来相似的输出语句：一个输出到标准输出流 cout，一个输出到文件 out。注意这里有多么方便——由于 cout 和 out 都是 ostream 对象，它们共享相同的操作接口，因此我们无需为不同的输出目标编写不同的格式化代码。第一个语句将代码行回显到控制台，第二个语句则将代码行连同行号（和一个冒号）写入新文件。

为了演示 getline()，程序打开刚才创建的文件并在去除行号（和冒号）后将内容输出到控制台。为了确保文件在打开供读取之前已正确关闭，我们在这里有两个选择。可以在程序的第一部分周围加上大括号，以强制 out 对象超出作用域，从而自动调用析构函数并关闭文件，这正是本例采取的做法。除此之外，还可以为两个文件调用 close()；如果选择这个方案，甚至可以通过调用 open()成员函数来重用 in 对象。

第二个 while 循环演示了当 getline()遇到终止字符时（第三个参数，默认为'\n'），会从输入流中移除该字符。尽管 getline()像 get()一样会在缓冲区中放入零（空）终止符'\0'，但它仍然不会插入终止字符。[①]

这个示例，以及本章的大多数示例，都假定每次调用 getline()的任何重载版本时，都会遇到换行符。如果不是这样，会设置流的 eofbit 标志位状态，getline()调用将返回 false，导致程序丢失最后一行输入。[②]

4.4.2 打开模式

可以通过覆盖构造函数的默认实参来控制文件的打开模式。表 4.2 总结了控制文件打开模式的各种标志。

表 4.2 文件的打开模式

标志	作用
ios::in	打开输入文件。作为 ofstream 的打开模式可以防止截断现有文件。[③]
ios::out	打开输出文件。若应用于 ofstream 且没有设置 ios::app、ios::ate 或 ios::in，就隐含同时设置了 ios::trunc。
ios::app	打开输出文件并仅进行追加。

[①] 译注：由于 getline()已经移除了行末的换行符，所以这里需要手动添加 endl，以便在输出后换行。

[②] 译注：别着急，4.5 节会展示如何用一种更优雅的方式输出整个文件的内容。

[③] 译注：所谓“截断”（truncating）现有文件，就是指清空现有文件的内容，而不是在现有内容后追加。

ios::ate	打开现有文件（输入或输出）并定位到末尾。[①]
ios::trunc	如果文件已存在，则截断旧文件。
ios::binary	以**二进制模式**打开文件。默认为**文本模式**。

可以通过“按位 OR”运算来组合这些标志。

binary 标志尽管是可移植的[②]，但它只有在某些非 UNIX 系统（例如，基于 MS-DOS 的操作系统）上才起作用，这些系统对于换行符的存储有特殊约定。例如，在 MS-DOS 系统中，在默认的文本模式下，每次输出换行符（'\n'）时，文件系统实际上会输出两个字符，即回车/换行对（CRLF），也就是一对 ASCII 字符 0x0D 和 0x0A。相反，在文本模式下将这种文件读回内存时，每当遇到这个字节对，程序就会把它们替换为一个'\n'。如果想绕过这种特殊处理，可以采取二进制模式打开文件。二进制模式与是否可以写入原始字节无关。相反，总是可以写入原始字节（通过调用 write()）。但是，如果真的打算使用 read()或 write()函数，那么还是应该以二进制模式打开文件，因为这些函数需要一个字节计数参数。在这些情况下，多余的'\r'字符会导致字节数不匹配。如果打算使用本章稍后讨论的流定位命令，那么也应该以二进制模式打开文件。

可以通过声明一个 fstream 对象来同时进行文件的输入和输出。声明 fstream 对象时，必须使用前面提到的充分的打开模式标志，让文件系统知道我们是要进行输入、输出还是两者都要。为了从输出切换到输入，需要对流进行 flush 处理[③]或者改变文件位置。为了从输入切换到输出，则要改变文件位置。为了通过 fstream 对象创建新文件，可以在构造函数调用中使用 ios::trunc 打开模式标志来进行输入和输出。[④]

[①] 译注：ate 是 at end 的简称。不是说要“吃掉”文件。ios::app 和 ios::ate 的区别在于，前者只允许在文件尾添加数据，而后者允许在文件的任何位置写入，甚至可以覆盖旧数据（换言之，虽然初始位置在文件末尾，但可以通过 seekp 等操作将文件指针移动到任意位置，进行读写操作。）

[②] 译注：一种语言特性如果说它“可移植”，那么意味着在任何平台上都能正常编译。

[③] 译注：fstream 对象的 flush 操作是指将缓冲区中的数据强制写入到实际的文件中。换句话说，就是把内存中暂存的数据立即同步到磁盘上。有些人将 flush 翻译成“刷新”，但其实 flush 在技术文档中的意思和日常生活中一样，即“冲洗(到别处)”。例如，我们会说“冲水”，不会说“刷新水”。

[④] 译注：与 ios::out 和 ios::in 配合使用。例如，fstream file("myFile.txt", ios::out | ios::trunc);。

4.5 iostream 缓冲处理

良好的设计实践要求在创建一个新类时，向类的用户尽可能隐藏底层实现细节。只公开他们需要知道的部分，将其余部分设为私有以避免混淆。使用插入符和提取符时，无论处理的是标准 I/O、文件、内存，还是某个新创建的类或设备，通常不知道也不关心字节是在哪里生成或消耗的。

然而，有时需要与负责生成和消耗[①]字节的 iostream 部分进行通信。为了给这一部分提供一个通用的接口，同时隐藏其底层实现，标准库将其抽象成一个名为 streambuf 的类。每个 iostream 对象都包含指向某种类型的 streambuf 的一个指针。具体类型则取决于它处理的是标准 I/O、文件还是内存等。我们可以直接访问 streambuf；例如，可以在 streambuf 中移入和移出原始字节，而无需通过包围它的 iostream 进行格式化。这是通过调用 streambuf 对象的成员函数来完成的。[②]

目前只需记住，每个 iostream 对象都包含指向一个 streambuf 对象的指针，而且 streambuf 对象有一些成员函数，可以根据需要进行调用。文件和字符串流分别有专门类型的流缓冲区，如图 4.3 所示。

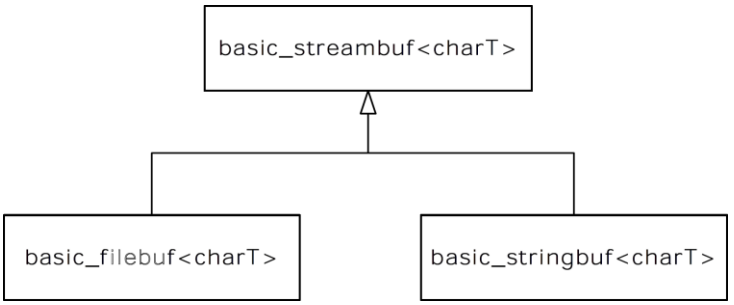

图 4.3 不同类型的 streambuf

为了能访问 streambuf，每个 iostream 对象都提供了一个名为 rdbuf()的成员函数，它返回指向该对象的专属 streambuf 的指针。拿这个指针后，就可以调用底层 streambuf 的任何成员函数了。不过，可以用 streambuf 指针做的最有趣的事情之一是使用<<操作符将其连接到另一个 iostream 对象上。这样就会将对象中的所有字符排空到<<左侧的对象中。如果想将所有字符从一个 iostream 移动到另一个 iostream 中，那么不必经历逐字符或逐行读取的繁琐过程（以及潜在的编码错误）。这是一种更为优雅的方法。

[①] 译注：其实 produce 和 consume 更通俗的说法就是“生产”和“消费”。

[②] 译注：我们平常一般就是通过 streambuf 来操作直接操作底层的字节流。

下面展示了一个简单的程序，它打开一个文件并将内容发送到标准输出（类似于上个例子）。

```
//: C04:Stype.cpp
// 将文件打印到标准输出
#include <fstream>
#include <iostream>
#include "../require.h"
using namespace std;

int main() {
    ifstream in("Stype.cpp");
    assure(in, "Stype.cpp");
    cout << in.rdbuf(); // 输出整个文件的内容
} ///:~
```

本例创建了一个 ifstream，使用当前程序的源代码文件作为参数。如果文件无法打开，assure()函数会报告失败。所有工作实际都发生在下面这个语句中：

```
cout << in.rdbuf();
```

这会将文件的全部内容发送到 cout。这样写不仅代码更简洁，而且通常比逐字节移动更有效率。

get()的一种形式能直接向另一个对象的 streambuf 写入。第一个参数是对目标 streambuf 的引用，第二个参数是终止字符（默认为'\n'），遇到该字符 get()函数就会停止。因此，还可以采取另一种方式将文件打印到标准输出，如下例所示：

```
//: C04:Sbufget.cpp
// 将文件复制到标准输出
#include <fstream>
#include <iostream>
#include "../require.h"
using namespace std;

int main() {
    ifstream in("Sbufget.cpp");
    assure(in);
    streambuf& sb = *cout.rdbuf(); // 指针解引用
    while (!in.get(sb).eof()) {
        if (in.fail())             // 发现空行
            in.clear();            // 清除所有错误标志位，使流恢复正常状态
        cout << char(in.get());    // 处理'\n'
    }
} ///:~
```

rdbuf()函数返回的是一个指针，因此必须进行解引用[①]，以满足函数需要“看到”一个对象的要求。流缓冲区本身就没有设计成可供拷贝（它们没有拷贝构造函数），因此我们将sb定义为对cout的流缓冲区的引用。针对输入文件中存在空行的情况（这个文件确实有），我们需要调用fail()和clear()。当get()的这个特定的重载版本看到连续两个换行符时（代表空行），它会设置输入流的failbit，因此必须调用clear()来重置以便流可以继续读取。经过这些处理后，对get()的第二个调用就可以正常提取并回显它遇到的每个换行定界符了。[②]（记住，get()函数不像getline()那样“无脑”地提取其定界符。）

您可能不会经常用到这样的技术，但最起码应该知晓它的存在。[③]

4.6 在输入输出流中定位

每种类型的iostream都有“下一个”字符的概念；这要么是指下一个字符来自何处（对于istream），要么是指下一个字符去往何处（对于ostream）。有的时候，我们需要在流中移动这个位置。可以使用两种方式来实现这种“定位”。一种方式是使用流中的一个绝对位置，称为streampos。第二种方式是像C标准库中用于处理文件的fseek()函数一样，从文件的开头、结尾或当前位置移动指定数量的字节。

如果打算使用streampos，那么首先需要调用一个“告知”（tell）函数。对于ostream来说，这个函数是tellp()；对于istream来说则是tellg()。（“p”代表“put指针”，“g”则代表“get指针”。众所周知，put和get代表两个不同的方向）。该函数返回一个streampos指针。以后为ostream调用seekp()或者为istream调用seekg()时就可以使用该指针，从而回到流中的那个位置。

第二种方式是相对定位，使用了seekp()和seekg()的重载版本。第一个参数指定要移动的字符数：它可以是正数或负数。第二个参数则指定定位方向。表4.3总结了不同的定位方向。

表4.3 定位方向

参数	方向

[①] 译注：也可以将“解引用”（dereference）说成“提领”。

[②] 译注：因为不会再受到因为连续两个换行符而产生的failbit的干扰。

[③] Langer & Kreft的*Standard C++ iostreams and Locales*一书（Addison–Wesley，1999）更深入地探讨了流缓冲区和流。

ios::beg	从流的开头
ios::cur	从流的当前位置
ios::end	从流的结尾

下例展示了如何在文件中移动。记住，我们并非只能像 C 的 stdio 那样在文件中进行定位。使用 C++，可以在任何类型的 iostream 中定位（虽然像 cin 和 cout 这样的标准流对象明确禁止这样做）。

```
//: C04:Seeking.cpp
// 在 iostream 中定位
#include <cassert>
#include <cstddef>
#include <cstring>
#include <fstream>
#include "../require.h"
using namespace std;

int main() {
    const int STR_NUM = 5, STR_LEN = 50; // 定义字符串数量和每个字符串的长度
    char origData[STR_NUM][STR_LEN] = {
        "我只能在没有哀愁的梦境里回来。",
        "我怕你亲吻我脸庞发现我满眼悲悔，",
        "我怕你追问离去的父亲。",
        "你年迈的孩子。",
        "我怕我见到你我会愧对原来的自己！"
    };
    char readData[STR_NUM][STR_LEN] = {{ 0 }};
    ofstream out("虞美人·故乡.bin", ios::out | ios::binary);
    assure(out, "虞美人·故乡.bin");
    for(int i = 0; i < STR_NUM; i++)
        out.write(origData[i], STR_LEN);
    out.close();

    // 重新打开文件
    ifstream in("虞美人·故乡.bin", ios::in | ios::binary);
    assure(in, "虞美人·故乡.bin");

    // 读取文件中存储的第一行文本
    in.read(readData[0], STR_LEN);
    assert(strcmp(readData[0], "我只能在没有哀愁的梦境里回来。") == 0);

    // 从文件尾向前移动 -STR_LEN 个字节
    in.seekg(-STR_LEN, ios::end); // 定位到了文件中存储的最后一行开头
    in.read(readData[1], STR_LEN);
```

```
    assert(strcmp(readData[1], "我怕我见到你我会愧对原来的自己！") == 0);

    // 绝对定位（类似于对文件使用 operator[]）
    in.seekg(3 * STR_LEN); // 定位到文件中存储的第 4 行文本开头
    in.read(readData[2], STR_LEN); // 读取第 4 行文本
    assert(strcmp(readData[2], "你年迈的孩子。") == 0);

    // 从当前位置向后（反向）移动
    in.seekg(-STR_LEN * 2, ios::cur); // 从第 4 行末尾定位到了第 3 行开头
    in.read(readData[3], STR_LEN);
    assert(strcmp(readData[3], "我怕你追问离去的父亲。") == 0);

    // 从文件的开头定位
    in.seekg(1 * STR_LEN, ios::beg);
    in.read(readData[4], STR_LEN); // 定位到了第 2 行开头
    assert(strcmp(readData[4], "我怕你亲吻我脸庞发现我满眼悲悔，") == 0);
} ///:~
```

该程序使用二进制输出流将刀郎的《虞美人·故乡》歌词片断写入文件。由于我们把它作为一个 ifstream 文件输入流重新打开，所以需要使用 seekg()来定位“get 指针”。如你所见，可以从文件的开头或结尾，或者从文件的当前位置定位。显然，必须提供一个正数来进行前向移动，提供负数来进行后向（反向）移动。

在理解了 streambuf 和如何定位后，就可以开始学习如何用另一种方法来创建流对象以支持文件的同时读和写了（除了使用 fstream 对象外）。以下代码首先创建一个带有标志的 ifstream，指明它是一个输入和输出文件。由于不能向 ifstream 写入数据，所以需要创建一个带有底层 streambuf 的 ostream：

```
ifstream in("filename", ios::in | ios::out);
ostream out(in.rdbuf());
```

您可能会好奇，向这两个对象写入时会发生什么。下面展示了一个例子：

```
//: C04:Iofile.cpp
// 读写同一个文件。
#include <fstream>
#include <iostream>
#include "../require.h"
using namespace std;

int main() {
    ifstream in("Iofile.cpp");
    assure(in, "Iofile.cpp");
    ofstream out("Iofile.out");
    assure(out, "Iofile.out");
    out << in.rdbuf(); // 拷贝文件
    in.close();
    out.close();
```

```
    // 打开文件供读写:
    ifstream in2("Iofile.out", ios::in | ios::out);
    assure(in2, "Iofile.out");
    ostream out2(in2.rdbuf());
    cout << in2.rdbuf(); // 从头打印整个文件
    out2 << "Where does this end up?";
    out2.seekp(0, ios::beg);
    out2 << "And what about this?";
    in2.seekg(0, ios::beg);
    cout << in2.rdbuf();
} ///:~
```

前 5 行代码将当前程序的源代码拷贝到一个名为 Iofile.out 的文件中，然后关闭文件。这样就得到了一个安全的文本文件供操作。然后，使用之前提到的技术创建两个对象，它们向同一个文件进行读写。执行 `cout << in2.rdbuf();`后，可以看到“get 指针”被初始化为文件的开头。然而，“put”指针默认设置为指向文件末尾。这从输出中就可以看出，因为`"Where does this end up?"`被追加到了文件末尾。但是，如果用 `seekp()`将 `put` 指针移到文件开头，那么后面插入的所有文本都会覆盖现有文本。然后，用 `seekg()`将 `get` 指针移到文件开头后，就可以看到这两次写入的内容。[1]最后，程序再次显示整个文件。当 `out2` 超出作用域并调用其析构函数时，文件会自动保存并关闭。

4.7 字符串输入输出流

字符串流直接与内存而不是文件或标准输出交互。它使用与 `cin` 和 `cout` 相同的读取和格式化函数来操纵内存中的字节。在古老的计算机术语中，人们将内存称为**核心**（core），因此这种功能曾被称为**核心格式化**（in-core formatting）。当然，现在没人这么说了。

字符串流的类名与文件流的类名相呼应。如果想创建一个从中提取字符的字符串流，那么可以创建一个 `istringstream`。相反，如果想向字符串流中插入字符，那么可以创建一个 `ostringstream`。所有字符串流的声明都在标准头文件`<sstream>`中。和往常一样，其中包含了顺应 `iostream` 继承层次结构的类模板，如图 4.4 所示。

[1] 译注：一般可以将 put 指针和 get 指针分别称为“写指针”和“读指针”。

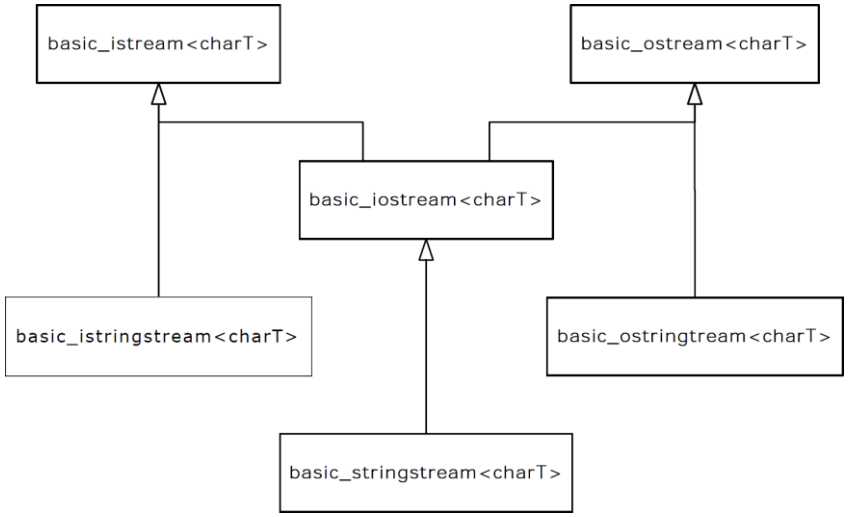

图 4.4 iostream 继承层次结构中的字符串流类

4.7.1 输入字符串流

为了通过流操作从字符串中读取，需要创建一个 istringstream 对象并用字符串来初始化。以下程序展示了如何使用 istringstream 对象：

```
//: C04:Istring.cpp
// 演示输入字符串流 istringstream
#include <cassert>
#include <cmath>  // 为了使用 fabs()
#include <iostream>
#include <limits> // 为了使用 epsilon()
#include <sstream>
#include <string>
using namespace std;

int main() {
    istringstream s("47 1.414 This is a test");
    int i;
    double f;
    s >> i >> f; // 空白字符分隔的输入
    assert(i == 47);
    double relerr = (fabs(f) - 1.414) / 1.414;
    assert(relerr <= numeric_limits<double>::epsilon());
    string buf2;
    s >> buf2;
    assert(buf2 == "This");
    cout << s.rdbuf(); // " is a test"
} ///:~
```

可以看出，将字符字符串转换为有类型的值时，相较于 atof()或 atoi()这样的标准 C 库函数，这是一种更加灵活和通用的方法，尽管 C 的库函数在执行单一（而不是批量）的转换时可能更高效。

在表达式 s >> i >> f 中，第一个数字被提取到 i 中，第二个数字被提取到 f 中。注意，这两个数字并不是“第一个以空白字符分隔的字符集”，因为它依赖于想要提取到的数据类型。例如，如果将字符串换成"1.414 47 This is a test"，那么 i 将获得值 1，因为输入例程会在小数点处停止。然后，f 会获得 0.414。如果想要将浮点数分解成整数和小数部分，这样做是没有问题的。否则，这在其他情况下似乎是一个错误。第二个 assert()计算了读取的值与预期值之间的相对误差；相较于直接比较两个浮点数是否相等，这是一种更好的做法。由<limits>定义的 epsilon()返回的常量代表了双精度数的机器 epsilon，这是在执行双精度数比较时，我们所能期待的最佳容差。[①]

您可能已经猜到，buf2 并不会获得剩余的字符串，而只会获得下一个以空白字符分隔的单词。一般来说，如果知道输入流的确切数据序列，并且需要转换为除了字符字符串[②]以外的其他某种类型，那么最好使用 iostream 提供的提取符（>>）。然而，如果想一次性提取剩余的字符串，并将其发送到另一个 iostream，那么可以像本例这样使用 rdbuf()。

为了测试本章开头展示的 Date 提取符，我们使用了以下测试程序和输入字符串流：[③]

```
//: C04:DateIOTest.cpp
//{L} ../C02/Date
#include <iostream>
#include <sstream>
#include "../C02/Date.h"  // Date 类在第 2 章定义
using namespace std;

void testDate(const string& s) {
    istringstream os(s);
    Date d;
    os >> d;
    if (os)
        cout << d << endl;
    else
        cout << "输入错误： \"" << s << "\"" << endl;
```

[①] 译注：机器 epsilon（machine epsilon）是一个数值，通常表示为 ϵ，它代表了浮点数系统中能够区分两个相邻的非零浮点数的最小正数。具体来说，对于任何非零浮点数 x，它是能够使 x + ϵ 与 x 不相等的最小正数。因此，当两个浮点数的差值小于 ϵ 时，我们通常认为这两个浮点数是相等的。

[②] 译注：这里强调了字符串中包含的都是真正的“字符”，而不是“数字”等其他目标转换类型。

[③] 译注：编译命令是 cl /EHsc DateIOTest.cpp ../C02/Date.cpp。

```
}

int main() {
    testDate("08-10-2023");
    testDate("8-10-2023");
    testDate("08 - 10 - 2023");
    testDate("A-10-2023");
    testDate("08%10/2023");
} ///:~
```

main()中的每个字符串字面值都以“传引用”的方式传递给 testDate()，后者又将其包装到一个 istringstream 中，以便测试本章开头为 Date 对象编写的流提取符。除此之外，testDate()函数还测试了插入符 operator<<()。

4.7.2 输出字符串流

为了创建输出字符串流，只需创建一个 ostringstream 对象。它负责管理一个动态大小的字符缓冲区，其中容纳了我们插入的任何内容。为了将格式化结果作为 string 对象获取，我们需要调用 str()成员函数，如下例所示：

```
//: C04:Ostring.cpp {RunByHand}
// 演示输出字符串流 ostringstream
#include <iostream>
#include <sstream>
#include <string>
using namespace std;

int main() {
    cout << "输入一个整数、一个浮点数和一个字符串： ";
    int i;
    float f;
    cin >> i >> f;
    cin >> ws; // 清除空白字符
    string stuff;
    getline(cin, stuff); // 获取行内其余内容
    ostringstream os;
    os << "整数 = " << i << endl;
    os << "浮点数 = " << f << endl;
    os << "字符串 = " << stuff << endl;
    string result = os.str();
    cout << result << endl;
} ///:~
```

这与前面的 Istring.cpp 示例相似，后者获取一个整数和一个浮点数。下面展示了程序的一次示例运行（用户输入加粗显示。注意，我们故意在 10 和 2.5 之间输入了空格和制表符等“空白字符”）：

```
输入一个整数、一个浮点数和一个字符串： 10          2.5    罗刹国 向东 两万六千里
```

```
整数 = 10
浮点数 = 2.5
字符串 = 罗刹国 向东 两万六千里
```

可以看出，和其他输出流一样，可以使用<<操作符和 endl 操纵元等常规格式化工具将字节发送到 ostringstream。str()函数每次调用都会返回一个新的 string 对象，不会影响字符串流的底层 stringbuf 对象。

第 3 章介绍了一个名为 HTMLStripper.cpp 的程序，它用于移除文本文件中的所有 HTML 标记和特殊代码。正如当时承诺的那样，可以使用字符串流来创建一个更优雅的解决方案，如下所示：[①]

```
//: C04:HTMLStripper2.cpp {RunByHand}
//{L} ../C03/ReplaceAll
// 该过滤器用于移除 HTML 标签和标记
#include <cstddef>
#include <cstdlib>
#include <fstream>
#include <iostream>
#include <sstream>
#include <stdexcept>
#include <string>
#include "../C03/ReplaceAll.h"
#include "../require.h"
using namespace std;

string& stripHTMLTags(string& s) throw(runtime_error) {
    size_t leftPos;
    while ((leftPos = s.find('<')) != string::npos) {
        size_t rightPos = s.find('>', leftPos + 1);
        if (rightPos == string::npos) {
            ostringstream msg;
            msg << "不完整的 HTML 标签开始于位置："
                << leftPos;
            throw runtime_error(msg.str());
        }
        s.erase(leftPos, rightPos - leftPos + 1);
    }
    // 移除所有特殊 HTML 字符
    replaceAll(s, "&lt;", "<");
    replaceAll(s, "&gt;", ">");
    replaceAll(s, "&amp;", "&");
    replaceAll(s, " ", " ");
```

[①] 译注：编译命令是 cl /EHsc HTMLStripper2.cpp ../C03/ReplaceAll.cpp。可以在第 4 章目录中执行 HTMLStripper2 test.html 来进行测试。

```
    // 等等...
    return s;
}

int main(int argc, char* argv[]) {
    requireArgs(argc, 1, "用法: HTMLStripper2 你的测试.html 文件");
    ifstream in(argv[1]);
    assure(in, argv[1]);
    // 将整个文件读入字符串；然后移除特殊字符
    ostringstream ss;
    ss << in.rdbuf();
    try {
        string s = ss.str();
        cout << stripHTMLTags(s) << endl;
        return EXIT_SUCCESS;
    } catch (runtime_error& x) {
        cout << x.what() << endl;
        return EXIT_FAILURE;
    }
} ///:~
```

在这个程序中，我们对输入文件流执行 rdbuf()调用，并将结果插入一个 ostringstream 中，从而实现对整个文件的读取。现在，查找各种特殊的 HTML 定界符对并删除它们变得简单多了，无需担心跨越行边界的问题。相反，第 3 章的早期版本在这上面花费了不少功夫。

下例展示了如何使用双向（即读/写）字符串流：

```
//: C04:StringSeeking.cpp {-bor}{-dmc}
// 对字符串流进行读取和写入
#include <cassert>
#include <sstream>
#include <string>
using namespace std;

int main() {
    string text = "We will hook no fish";
    stringstream ss(text);
    ss.seekp(0, ios::end);
    ss << " before its time.";
    assert(ss.str() == "We will hook no fish before its time.");
    // 将"hook"改为"ship"
    ss.seekg(8, ios::beg);
    string word;
    ss >> word;
    assert(word == "hook");
    ss.seekp(8, ios::beg);
    ss << "ship";
    // 将"fish"改为"code"
```

```
    ss.seekg(16, ios::beg);
    ss >> word;
    assert(word == "fish");
    ss.seekp(16, ios::beg);
    ss << "code";
    assert(ss.str() == "We will ship no code before its time.");
    ss.str("A horse of a different color."); // 演示 str()的重载版本
    assert(ss.str() == "A horse of a different color.");
} ///:~
```

和往常一样，移动 put 指针需要调用 seekp()，重新定位 get 指针则需要调用 seekg()。尽管本例没有演示，但字符串流比文件流稍微宽容一些，因为可以在任何时候从读取切换到写入，反之亦然。不需要重新定位 get 或 put 指针，也不需要对流进行 flush 处理（强制将内存数据写入磁盘文件）。这个程序还演示了 str()的重载版本，它用新字符串替换流的底层 stringbuf。

4.8 输出流格式化

iostream 设计的目标是帮助我们轻松移动和/或格式化字符。但是，如果不提供 C 的 printf()系列函数所提供的大部分格式化功能，那么所有这一切都将毫无意义。本节将探讨 iostream 可以使用的所有输出格式化函数，以便按照自己希望的方式对字节进行格式化。

iostream 中的格式化函数一开始可能会让人有些困惑，因为控制格式化的方式往往不止一种：既可以通过成员函数来控制，也可以通过操纵元来控制。进一步带来混淆的是，有一个泛型成员函数可以通过设置状态标志来控制格式化，比如左对齐或右对齐、使用大写字母来表示十六进制记数法以及始终为浮点值使用小数点等等。相反，又用一些独立的成员函数来设置和读取代表填充字符、域宽和精度的值。

为了澄清这一切，我们首先探讨 iostream 的内部格式化数据，以及可以修改这些数据的成员函数。（如果愿意，一切都可以通过成员函数来控制。）我们将单独介绍操纵元。

4.8.1 格式标志

ios 类的一些数据成员存储了与流相关的所有格式化信息。其中一部分数据成员具有取值范围，并且存储在变量中。[1]这些数据包括浮点数精度、输出域宽以及在输出中使用的填充字符（通常是空格）。其余格式化信息由标志（flag）确定，这些标志通常组合在一起以节省空间，并且统称为**格式标志**（format flags）。在 ios 的成员函数中，ios::flags()用于查询格式标志的当前值，它无参并返回一个 fmtflags 类型的对象（通常定义为 long 的别

[1] 译注：例如，std::cout.precision(3);。

名），其中包含了当前的格式标志。其余成员函数则用于更改格式标志，并返回之前的格式标志值，如下所示：

```
fmtflags ios::flags(fmtflags newflags);
fmtflags ios::setf(fmtflags ored_flag);
fmtflags ios::unsetf(fmtflags clear_flag);
fmtflags ios::setf(fmtflags bits, fmtflags field);
```

第一个函数会强制改变**所有**标志位。虽然有时需要如此，但我们更多时候是用其他三个函数一次改变一个标志位。

setf()具体怎么使用可能会令人有点困惑。要知道应该使用哪个重载版本，必须知道要改变的是哪种类型的标志。有两种标志：简单地开或关的标志，以及与其他标志协同工作的标志。开/关标志最容易理解，因为可以通过 setf(fmtflags)来开启它们，并通过 unsetf(fmtflags)来关闭它们。表 4.4 总结了这些标志。

表 4.4 开/关标志

开/关标志	**作用**
ios::skipws	控制是否忽略空白字符。对于输入流，默认是开启的。
ios::showbase	控制在打印整数值时是否显示基数前缀（需要先设置进制，例如 std::dec、std::oct 或 std::hex）。输入流在 showbase 开启时也会识别基数前缀。
ios::showpoint	显示浮点值的小数点和尾随零。
ios::uppercase	以大写字母 A-F 表示十六进制值，并以 E 表示科学记数。
ios::showpos	为正数显示正号（+）。
ios::unitbuf	unitbuf 是指“单元缓冲”（Unit Buffering）。每次插入后，都立即对流进行 flush 处理。

例如，为了让 cout 显示加号，可以说 cout.setf(ios::showpos)。要停止显示正号，则说 cout.unsetf(ios::showpos)。

unitbuf 标志控制单元缓冲，这意味着每次插入后，数据会立即 flush 到输出流。这在进行错误跟踪时很有用，因为若程序崩溃，数据仍会被写入日志文件。以下程序演示了单元缓冲：

```
//: C04:Unitbuf.cpp {RunByHand}
#include <cstdlib> // 为了使用 abort()
```

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

int main() {
    ofstream out("log.txt");
    out.setf(ios::unitbuf);
    out << "one" << endl;
    out << "two" << endl;
    abort();
}
```

在执行任何插入操作之前，都有必要先开启单元缓冲。若注释掉对 setf()的调用，某个特定的编译器只在 log.txt 文件中写入了字母'o'。而在开启单元缓冲的情况下，不管 C++的什么实现，都不会出现数据的丢失。[①]

注意，标准错误输出流 cerr 默认开启了单元缓冲。单元缓冲是有代价的，因此如果一个输出流被超负荷使用，那么除非不用考虑效率问题，否则不要随便开启单元缓冲。

4.8.2 格式化标志组

第二种类型的格式化标志是成组工作的。一次只能设置一个这样的标志，就像老式汽车收音机上的按钮——按下其中一个，其他的就会弹起来。但令人遗憾的是，这并不会自动发生。必须注意自己设置了哪些标志，以免不小心调用了错误的 setf()函数。例如，数值的每种基数（十六进制、十进制和八进制）都有一个对应的标志。跟基数有关的这些标志都在名为 ios::basefield 的标志组内。如果设置了 ios::dec 标志，然后又调用 setf(ios::hex)，那么会设置 ios::hex 标志，但不会自动清除 ios::dec，导致出现未定义的行为。相反，此时应该调用 setf()的第二种形式，如下所示：setf(ios::hex, ios::basefield)。该函数首先清除 ios::basefield 中的所有位，然后再设置 ios::hex。因此，这种形式的 setf()确保了当设置了一个标志位时，组内的其他标志位会自动“弹起来”。ios::hex 操纵元能自动完成所有这些工作，因此不需要关心这个类的内部实现细节，甚至不必知道它是一组二进制标志。稍后就会讲到，在所有能使用 setf()的地方，都可以使用具有相同功能的操纵元。[②]

表 4.5~到 4.7 总结了不同的标志组以及其中各个标志的作用。

[①] 译注：测试了几个现代 C++编译器，均没有出现数据丢失的情况。

[②] 译注：例如，以下语句就使用了 hex 操纵元：std::cout << std::hex << 255 << std::endl; 。该语句输出 ff。这样写更方便，没必要先执行 cout.setf(ios::hex, ios::basefield);。

表 4.5　ios::basefield 标志组

ios::basefield	作用
ios::dec	以十进制（base 10）格式化整数值（默认，无前缀）。
ios::hex	以十六进制（base 16）格式化整数值（显示前缀 0x）。
ios::oct	以八进制（base 8）格式化整数值（显示前缀 0）。

表 4.6　ios::floatfield 标志组

ios::floatfield	作用
ios::scientific	以科学记数格式显示浮点数。精度字段指定了小数点后的位数。
ios::fixed	以固定小数点（定点）格式显示浮点数。精度字段指定了小数点后的位数。
“自动”（上述两个位都没有设置）	精度字段指定了有效数位总数。[①]

表 4.7 ios::adjustfield 标志组

ios::adjustfield	作用
ios::left	左对齐值；右侧用填充字符填充。
ios::right	右对齐值。左侧用填充字符填充。这是默认对齐方式。
ios::internal	在任何前置符号或基数指示符之后、但在值之前添加填充字符。换言之，若显示正负号，那么正负号会左对齐，数值则右对齐。[②]

[①] 译注：例如，cout << setprecision(5) << 12345.6789 << endl;会显示 12345.67890。

[②] 译注：例如，cout << setw(10) << setfill('0') << internal << -123 << endl;会显示-00000123。

4.8.3 宽度、填充和精度

一些内部变量控制着输出域的宽度、输出域的填充字符以及打印浮点数时的精度，可以通过同名的成员函数来读写这些成员变量。表 4.8 对它们进行了总结。

表 4.8 设置域宽、填充字符和精度

函数	作用
`int ios::width()`	返回当前宽度，默认为 0。插入和提取都可以使用。
`int ios::width(int n)`	设置宽度，返回之前的宽度。
`int ios::fill()`	返回当前填充字符，默认为空格。
`int ios::fill(int n)`	设置填充字符，返回之前的填充字符。
`int ios::precision()`	返回当前浮点数精度（小数点后的位数），默认为 6。
`int ios::precision(int n)`	设置浮点数精度，返回之前的精度。参见 `ios::floatfield` 表格（表 4.6）以了解“精度”的含义。

`fill` 和 `precision` 值很容易理解，但 `width` 需要进一步解释。当 `width` 为零时，插入一个值会产生表示该值所需的最少字符数。若 `width` 为正数，意味着插入一个值会产生至少 `width` 个字符；若插入的这个值的字符数少于 `width`，就会用填充字符填充它的输出域。但是，值无论如何都不会被截断（不会删除任何字符）。因此，如果将 `width` 设为 2，并尝试显示 123，那么仍然会得到 123。域宽只能指定最小字符数（输出域的最小宽度）；它没有办法指定最大字符数（输出域的最大宽度）。

宽度还颇为独特，因为它会被每一个可能受其影响的插入符或提取符重置为零。它实际上并不是一个状态变量，而是插入符和提取符的一个隐式参数。如果想要一个恒定的宽度，请在每次插入或提取后调用 `width()`。

4.8.4 一个完整的示例

下例运用了前面讨论的所有函数。搞懂了这个示例，就掌握了它们的用法。

```
//: C04:Format.cpp
// 演示各种格式化函数
#include <fstream>
```

```
#include <iostream>
#include "../require.h"
using namespace std;
#define D(A) T << #A << endl; A

int main() {
    ofstream T("format.out");
    assure(T);

    D(int i = 47;)
    D(float f = 2300114.414159;)
    const char* s = "还有更多吗？";

    D(T.setf(ios::unitbuf);)
    D(T.setf(ios::showbase);)
    D(T.setf(ios::uppercase | ios::showpos);)
    D(T << i << endl;) // 默认是十进制
    D(T.setf(ios::hex, ios::basefield);) // 十六进制
    D(T << i << endl;)
    D(T.setf(ios::oct, ios::basefield);) // 八进制
    D(T << i << endl;)
    D(T.unsetf(ios::showbase);)
    D(T.setf(ios::dec, ios::basefield);)
    D(T.setf(ios::left, ios::adjustfield);)
    D(T.fill('0');)
    D(T << "填充字符: " << T.fill() << endl;)
    D(T.width(10);)
    T << i << endl;
    D(T.setf(ios::right, ios::adjustfield);)
    D(T.width(10);)
    T << i << endl;
    D(T.setf(ios::internal, ios::adjustfield);)
    D(T.width(10);)
    T << i << endl;
    D(T << i << endl;) // 没有 width(10)

    D(T.unsetf(ios::showpos);)
    D(T.setf(ios::showpoint);)
    D(T << "精度 = " << T.precision() << endl;)
    D(T.setf(ios::scientific, ios::floatfield);)
    D(T << endl << f << endl;)
    D(T.unsetf(ios::uppercase);)
    D(T << endl << f << endl;)
    D(T.setf(ios::fixed, ios::floatfield);)
    D(T << f << endl;)
    D(T.precision(20);)
    D(T << "精度 = " << T.precision() << endl;)
    D(T << endl << f << endl;)
    D(T.setf(ios::scientific, ios::floatfield);)
    D(T << endl << f << endl;)
```

```
    D(T.setf(ios::fixed, ios::floatfield);)
    D(T << f << endl;)

    D(T.width(10);)
    T << s << endl;
    D(T.width(40);)
    T << s << endl;
    D(T.setf(ios::left, ios::adjustfield);)
    D(T.width(40);)
    T << s << endl;
} ///:~
```

本例利用了一个技巧来创建跟踪文件，以便监控正在发生的事情。其中，D(A)宏使用预处理器的“字符串化”（stringizing）功能将 A 转换成一个字符串来显示。然后，它重复 A 使语句得以实际执行。宏将所有信息发送到一个叫做 T 的跟踪文件中。[1]程序的输出结果是：

```
int i = 47;
float f = 2300114.414159;
T.setf(ios::unitbuf);
T.setf(ios::showbase);
T.setf(ios::uppercase | ios::showpos);
T << i << endl;
+47
T.setf(ios::hex, ios::basefield);
T << i << endl;
0X2F
T.setf(ios::oct, ios::basefield);
T << i << endl;
057
T.unsetf(ios::showbase);
T.setf(ios::dec, ios::basefield);
T.setf(ios::left, ios::adjustfield);
T.fill('0');
T << "fill char: " << T.fill() << endl;
填充字符: 0
T.width(10);
+470000000
T.setf(ios::right, ios::adjustfield);
T.width(10);
0000000+47
T.setf(ios::internal, ios::adjustfield);
```

---

[1] 译注：在#define D(A) T << #A << endl; A 这个宏定义中，#A 是一个预处理操作符，它将 A 转换为一个字符串。例如，如果 A 是 int i = 47;，那么#A 将被转换为"int i = 47;"。注意，T << #A << endl;会将字符串化的 A 和一个换行符输出到流 T 中，即文件 format.out。最后一个 A 作为宏参数则会原封不动地执行。

```
T.width(10);
+000000047
T << i << endl;
+47
T.unsetf(ios::showpos);
T.setf(ios::showpoint);
T << "精度 = " << T.precision() << endl;
精度 = 6
T.setf(ios::scientific, ios::floatfield);
T << endl << f << endl;

2.300114E+06
T.unsetf(ios::uppercase);
T << endl << f << endl;

2.300114e+06
T.setf(ios::fixed, ios::floatfield);
T << f << endl;
2300114.500000
T.precision(20);
T << "精度 = " << T.precision() << endl;
精度 = 20
T << endl << f << endl;

2300114.50000000000000000000
T.setf(ios::scientific, ios::floatfield);
T << endl << f << endl;

2.30011450000000000000e+06
T.setf(ios::fixed, ios::floatfield);
T << f << endl;
2300114.50000000000000000000
T.width(10);
还有更多吗？
T.width(40);
0000000000000000000000000000 还有更多吗？
T.setf(ios::left, ios::adjustfield);
T.width(40);
还有更多吗？0000000000000000000000000000
```

通过研究上述输出，可以更好地理解 iostream 的各种格式化成员函数。

4.9 操纵元

从上一个程序可以看出，为流的各种格式化操作调用成员函数显得颇为繁琐。为了简化编码并提升可读性，标准库提供了一组**操纵元**（manipulator）来复制成员函数提供的操作。

操纵元之所以便利，是因为它们直接插入表达式就可以生效；不需要单独写函数调用语句。

操纵元的作用是改变流的状态而不是处理数据；即使会处理数据，也只是“顺带而为”。例如，在输出表达式中插入 endl 时，它不仅会插入一个换行符，还会对流进行 flush（也就是说，输出已经存储在内部流缓冲区但尚未输出的所有待处理字符）。也可以像这样单纯地 flush 一个流：

```
cout << flush;
```

这会导致一个对 flush()成员函数的调用，即：

```
cout.flush();
```

注意，这个成员函数调用是插入符<<的一个“副作用”（没有任何实质性的东西被插入到流中）。可以利用其他基本操纵元将数值的基数更改为八进制（oct）、十进制（dec）或十六进制（hex）。例如：

```
cout << hex << "0x" << i << endl;
```

在这种情况下，数值输出将以十六进制模式继续，直至在输出流中插入 dec 或 oct 来重新更改为止。

上面这些都是插入操纵元，还有一个提取操纵元可以“吃掉”空白字符（包括空格等）：

```
cin >> ws;
```

无参的操纵元在<iostream>中提供，其中包括 dec、oct 和 hex，它们分别执行与 setf(ios::dec, ios::basefield)、setf(ios::oct, ios::basefield)和 setf(ios::hex, ios::basefield)相同的操作，只是更简洁。<iostream>头文件还提供了 ws、endl 和 flush 以及表 4.9 总结的一组额外的操纵元。

表 4.9 其他无参操纵元

操纵元	**作用**
showbase noshowbase	控制打印整数值时是否显示数值基数（dec，oct 或 hex）。
showpos noshowpos	控制是否为正数显示正号（+）。
uppercase	控制是否为十六进制值显示大写的 A-F，以及是否为科学记数显示大写的 E。

nouppercase	
showpoint noshowpoint	控制是否为浮点值显示小数点和尾随零。
skipws noskipws	控制是否跳过输入中的空白字符。ws 是“空白”的意思。
left right internal	左对齐，右侧填充。 右对齐，左侧填充。 在前置正负号/基数指示符与值之间填充。
scientific fixed	控制浮点数输出的显示偏好（科学记数 vs.固定小数点）。

4.9.1 有参操纵元

有 6 个标准操纵元（例如，setw()）需要提供实参。它们在头文件<iomanip>中定义。表 4.10 对它们进行了总结：

表 4.10 有参操纵元

操纵元	作用
setiosflags(fmtflags n)	等价于调用 setf(n)。设置会一直生效，直到下一次更改，例如执行 ios::setf()。
resetiosflags(fmtflags n)	只清除由 n 指定的格式标志。设置会一直生效，直到下一次更改，例如执行 ios::unsetf()。[①]
setbase(base n)	将数值基数（进制）更改为 n，其中 n 可以是 10、8 或 16（其他任何值都会导致结果 0）。如果 n 为零，则输出时使用基数 10，但输入使用 C 语言的约定：10 代表十进制的 10，010 代表十进制的 8，而 0xf 代表十进制的 15。

[①] 译注：之前说过，将 fmtflags 理解为长整型 long。

	也可以直接使用 dec、oct 和 hex 来进行输出的进制控制。
setfill(char n)	将填充字符更改为 n，类似于 ios::fill()。
setprecision(int n)	将精度更改为 n，类似于 ios::precision()。
setw(int n)	将字段宽度更改为 n，类似于 ios::width()。

如果需要做大量格式化工作，那么使用操纵元而不是调用流成员函数，可以使代码变得更整洁。下例使用操纵元重写了上一节的程序（删除了 D()宏以提升可读性）。

```
//: C04:Manips.cpp
// 使用操纵元（manipulator）重写 Format.cpp
#include <fstream>
#include <iomanip>
#include <iostream>

using namespace std;

int main() {
    ofstream trc("trace.out");
    int i = 47;
    float f = 2300114.414159;
    char* s = "还有更多吗？";

    trc << setiosflags(ios::unitbuf | ios::showbase | ios::uppercase | ios::showpos);

    trc << i << endl;
    trc << hex << i << endl
        << oct << i << endl;

    trc.setf(ios::left, ios::adjustfield);
    trc << resetiosflags(ios::showbase)
        << dec << setfill('0');

    trc << "填充字符: " << trc.fill() << endl;
    trc << setw(10) << i << endl;

    trc.setf(ios::right, ios::adjustfield);
    trc << setw(10) << i << endl;

    trc.setf(ios::internal, ios::adjustfield);
    trc << setw(10) << i << endl;

    trc << i << endl; // Without setw(10)

    trc << resetiosflags(ios::showpos)
```

```
            << setiosflags(ios::showpoint)
            << "精度 = " << trc.precision() << endl;

        trc.setf(ios::scientific, ios::floatfield);
        trc << f << resetiosflags(ios::uppercase) << endl;

        trc.setf(ios::fixed, ios::floatfield);
        trc << f << endl;

        trc << f << endl;

        trc << setprecision(20);
        trc << "精度 = " << trc.precision() << endl;
        trc << f << endl;

        trc.setf(ios::scientific, ios::floatfield);
        trc << f << endl;

        trc.setf(ios::fixed, ios::floatfield);
        trc << f << endl;

        trc << setw(10) << s << endl;
        trc << setw(40) << s << endl;

        trc.setf(ios::left, ios::adjustfield);
        trc << setw(40) << s << endl;
    } ///:~
```

可以看出，许多多行语句已被压缩为单一的链式插入。注意程序中的 setiosflags()调用，我们向函数传递了标志位的“按位 OR”运算结果。这也可以通过上一个例子的 setf()和 unsetf()来实现。

为输出流使用 setw()时，输出表达式会被格式化成一个临时字符串。格式化结果的长度会与 setw()的实参进行比较。如有必要，会使用当前的填充字符进行填充。换言之，setw()影响的是格式化输出操作的**结果字符串**。[1]类似地，只有在读取**字符串**的时候，为输入流使用 setw()才有意义，如下例所示：

```
//: C04:InputWidth.cpp
// 演示将 setw 应用于输入时的限制
#include <cassert>
#include <cmath>
#include <iomanip>
#include <limits>
#include <sstream>
```

[1] 译注：也就是说，输出到指定输出流中的是填充后的临时字符串。

```
#include <string>

using namespace std;

int main() {
    istringstream is("one 2.34 five");
    string temp;
    is >> setw(2) >> temp;
    assert(temp == "on");

    is >> setw(2) >> temp;
    assert(temp == "e");

    double x;
    is >> setw(2) >> x;
    double relerr = fabs(x - 2.34) / x;
    assert(relerr <= numeric_limits<double>::epsilon());
} ///:~
```

在尝试读取一个字符串时，setw()能很好地控制提取的字符数量……但也只是在某种程度上“很好”。第一次提取到两个字符，但是第二次只提取到一个，即使我们要求两个。这是因为 operator>>()使用空白字符作为分隔符（除非关闭 skipws 标志）。然而，当尝试读取一个数字，例如 x 时，使用 setw()来限制读取的字符数量是没有作用的。因此，对于输入流，我们仅在提取字符串时才使用 setw()。

4.9.2 创建操纵元

我们有时想要创建自己的操纵元，这其实很容易。无参操纵元（例如，endl）本质上是一个接收 ostream 引用并返回 ostream 引用的函数。endl 的声明如下所示：

```
ostream& endl(ostream&);
```

现在，当我们这样说的时候：

```
cout << "howdy" << endl;
```

endl 会生成那个函数的**地址**。因此编译器就会问，“是否可以在这里应用一个实参为函数地址的函数？”在<iostream>中预定义的函数就是这样做的。我们把这种函数称为**应用器**（即 applicator，因为它们将一个函数**应用**于一个流）。应用器调用其函数实参，并将 ostream 对象作为该函数的实参传递给它。不需要知道应用器的具体工作方式，我们就可以创建自己的操纵元；只需要知道它们的存在就可以了。下面展示了 ostream 应用器经过简化后的代码：

```
ostream& ostream::operator<<(ostream& (*pf)(ostream&)) {
    return pf(*this); // pf 是函数指针的简称
}
```

实际定义要稍微复杂一些，因为它涉及到模板。但是，可以通过上述代码理解到这个技术的要点。当一个函数（例如*pf，它接收一个流参数并返回一个流引用）被插入到一个流中时，就会调用这个应用器函数，后者进而执行 pf 指向的函数。标准 C++库预定义了 ios_base、basic_ios、basic_ostream 和 basic_istream 的“应用器”。

为了更清楚地理解这一过程，下面展示了一个简单的例子。该程序创建一个名为 nl 的操纵元，它的作用是单纯地向流中插入一个换行符（也就是说，不像 endl 那样还会“顺手”对流进行 flush）。

```
//: C04:nl.cpp
// 自定义一个操纵元
#include <iostream>

using namespace std;

// nl 代表 new line
ostream& nl(ostream& os) {
    return os << '\n';
}

int main() {
    cout << "在" << nl << "每" << nl << "个" << nl
         << "字" << nl << "之" << nl << "间" << nl
         << "插" << nl << "入" << nl << "换" << nl
         << "行" << nl << "符" << nl;
} ///:~
```

将 nl 插入输出流（例如，cout）时，会执行以下函数调用序列：

```
cout.operator<<(nl)  →  nl(cout)
```

nl()内部的表达式 os << '\n'会调用 ostream::operator(char)，后者会返回流，并最终由 nl()返回。[①]

4.9.3 效果器

如前所述，无参操纵元很容易创建。但是，如果想要创建有参的操纵元呢？检查一下<iomanip>头文件，会发现一个名为 smanip 的类型，这就是那些有参操纵元返回的东西。一些人或许会想利用这种类型定义自己的操纵元，但千万不要这样做。smanip 类型依赖于实现，因此不具可移植性。但好消息是，可以基于 Jerry Schwarz 提出的一种称为**效果器**

[①] 如果想把 nl 放到头文件中，请先把它修改成一个内联函数。

（effector）的技术，无需特殊机制就能直接定义这样的操纵元。[1]效果器是一个简单的类，其构造函数格式化一个表示了所需操作的字符串。还提供了一个重载的operator<<，可以将该字符串插入流中。下例包含了两个效果器。第一个输出截断的字符串，第二个以二进制形式打印数字。

```
//: C04:Effector.cpp
// Jerry Schwarz 的“效果器”（effector）
#include <cassert>
#include <limits>  // 为了使用 max()
#include <sstream>
#include <iostream>
#include <string>
using namespace std;

// 截断并输出字符串的前缀部分（指定宽度）：
class Fixw {
    string str;
public:
    Fixw(const string& s, int width) : str(s, 0, width) {}
    friend ostream& operator<<(ostream& os, const Fixw& fw) {
        return os << fw.str;
    }
};

// 以二进制形式打印一个数字：
typedef unsigned long ulong;

class Bin {
    ulong n;
public:
    Bin(ulong nn) { n = nn; }
    friend ostream& operator<<(ostream& os, const Bin& b) {
        const ulong ULMAX = numeric_limits<ulong>::max();
        ulong bit = ~(ULMAX >> 1); // 设置最高位
        while (bit) {
            os << (b.n & bit ? '1' : '0');
            bit >>= 1;
        }
        return os;
    }
};

int main() {
    string words = "Things that make us happy, make us wise";
    // 以下循环每次都从字符串中截去一个字符，输出原始字符串不断缩短的“前缀”
```

[1] Jerry Schwarz 是 iostream 的设计者。

```
    // 如果字符串包含的全是汉字和全角标点，请将--i 修改成(i = i - 2)
    for (int i = words.size(); --i >= 0;) {
        ostringstream s;
        s << Fixw(words, i);
        assert(s.str() == words.substr(0, i));
        cout << s.str() << endl;
    }

    ostringstream xs, ys;
    xs << Bin(0xCAFEBABEUL);
    assert(xs.str() == "1100""1010""1111""1110""1011""1010""1011""1110");
    cout << xs.str() << endl;

    ys << Bin(0x76543210UL);
    assert(ys.str() == "0111""0110""0101""0100""0011""0010""0001""0000");
} ///:~
```

Fixw 的构造函数为其 char*实参创建一个缩短的拷贝，析构函数会释放为这个拷贝分配的内存。重载的 operator<<获取其第二个实参的内容（即 Fixw 对象），将其插入第一个实参 ostream 中，然后返回该 ostream，以便在链式表达式中使用。在一个表达式中使用 Fixw 时，如下所示：

```
cout << Fixw(string, i) << endl;
```

会通过调用 Fixw 构造函数创建一个**临时对象**，并将该临时对象传递给 operator<<。最终的效果等同于有参操纵元。临时 Fixw 对象的生命期会持续到该语句结束。

至于 Bin 效果器，它则依赖于这样一个事实：对无符号数进行右移位时，移出的高位会被补 0。我们使用 numeric_limits<unsigned long>::max()（即最大的无符号长整型值，来自标准头文件<limits>）来生成一个最高位被设置的值。[①]然后，每次都使这个值右移一位，并与目标数字进行“逻辑 AND”运算。

注意，代码中并列了这些字符串字面值以提升可读性。但是，这些独立的字符串会被编译器拼接成单一的字符串。

历史上，这个技术的问题在于，一旦为 char*创建了名为 Fixw 的类，或者为 unsigned long 创建了 Bin 类，其他人就不能为他们的类型创建一个不同的 Fixw 类或 Bin 类了。不

[①] 译注：这相当于得到了一个掩码。由于代码中将常量 ULMAX 定义无符号长整型的最大值，所以它的二进制表示是一个全是 1 的数。比如对于 64 位系统，就是 64 个 1。右移一位，最高位变成 0。再取反，所有位反转，变成了 100000...。这就是我们想要的掩码。这个掩码每次右移一位，并和目标数字执行按位 AND 运算，从而判断目标数字中的相应 bit 是 1 还是 0，这个 bit 的值（字符）会被送入名为 os 的输出流。当掩码不断右移位变成全零之后，处理结束。

过，可以通过命名空间来避免这个问题。效果器和操纵元并不等价，尽管经常可以用它们来解决相同的问题。如果发现效果器无法满足需求，那么就需要克服操纵元的复杂性了。

4.10 输入输出流示例

本节将通过一系列示例，展示如何将本章所学知识融会贯通。虽然市面上不乏字节操作工具（如 UNIX 下的流编辑器 sed 和 awk，甚至是文本编辑器），但这些工具往往存在一些局限性。sed 和 awk 的处理速度可能较慢，且只能按顺序处理文本行；而文本编辑器则需要用户手动操作或编写复杂的宏。相比之下，基于输入输出流编写的程序更加灵活高效，不仅运行速度快，可移植性强，而且能满足各种复杂的处理需求。

4.10.1 维护类库源代码

在创建类时，我们通常会从库的角度进行考虑。具体来说，我们会创建一个头文件 Name.h 来包含类的声明，再创建一个 Name.cpp 文件来包含成员函数的实现。这些文件要遵循一定的规范。首先，它们要符合特定的编码标准（本例将使用本书一直在使用的编码格式）。其次，头文件中要围绕代码写一些预处理器语句，以防止类的重复声明（重复声明会使编译器感到困惑——它无法确定应该使用哪一个声明。这些声明可能不同，因此编译器会报错）。

本例的作用是新建一对头文件和实现文件，或者修改已存在的文件。如果文件已经存在，它会检查并可能修改这些文件。但是，如果不存在，就会使用正确的格式创建它们。

```
//: C04:Cppcheck.cpp
// 配置.h 和.cpp 文件，使之符合样式标准。
// 测试现有文件以确保符合标准。
#include <fstream>
#include <sstream>
#include <string>
#include <cstddef>
#include "../require.h"
using namespace std;

bool startsWith(const string& base, const string& key) {
    return base.compare(0, key.size(), key) == 0;
}

void cppCheck(string fileName) {
    enum bufs { BASE, HEADER, IMPLEMENT, HLINE1, GUARD1,
        GUARD2, GUARD3, CPPLINE1, INCLUDE, BUFNUM };
    string part[BUFNUM];

    part[BASE] = fileName;
```

```
// 查找字符串中的'.'
size_t loc = part[BASE].find('.');
if(loc != string::npos)
    part[BASE].erase(loc); // 去掉扩展名

// 强制转换为大写：
for(size_t i = 0; i < part[BASE].size(); i++)
    part[BASE][i] = toupper(part[BASE][i]);

// 创建文件名和文件内部的代码行：
part[HEADER] = part[BASE] + ".h";
part[IMPLEMENT] = part[BASE] + ".cpp";
part[HLINE1] = "//" ": " + part[HEADER];
part[GUARD1] = "#ifndef " + part[BASE] + "_H";
part[GUARD2] = "#define " + part[BASE] + "_H";
part[GUARD3] = "#endif // " + part[BASE] +"_H";
part[CPPLINE1] = string("//") + ": " + part[IMPLEMENT];
part[INCLUDE] = "#include \"" + part[HEADER] + "\"";

// 首先，尝试打开现有的文件：
ifstream existh(part[HEADER].c_str()),
         existcpp(part[IMPLEMENT].c_str());

if(!existh) { // 文件不存在；创建它
    ofstream newheader(part[HEADER].c_str());
    assure(newheader, part[HEADER].c_str());
    newheader << part[HLINE1] << endl
              << part[GUARD1] << endl
              << part[GUARD2] << endl << endl
              << part[GUARD3] << endl;
} else { // 文件已存在；验证它
    stringstream hfile; // 写入和读取
    ostringstream newheader; // 写入
    hfile << existh.rdbuf(); // 检查前三行是否符合标准：
    bool changed = false;
    string s;
    hfile.seekg(0);
    getline(hfile, s);
    bool lineUsed = false;

    // 对good()的调用是为了兼容Microsoft的编译器（后同）：
    for(int line = HLINE1; hfile.good() && line <= GUARD2; ++line) {
        if(startsWith(s, part[line])) {
            newheader << s << endl;
            lineUsed = true;
            if(getline(hfile, s))
                lineUsed = false;
        } else {
            newheader << part[line] << endl;
            changed = true;
```

```
            lineUsed = false;
        }
    }

    // 拷贝文件剩余内容
    if(!lineUsed)
        newheader << s << endl;
    newheader << hfile.rdbuf();

    // 检查 GUARD3
    string head = hfile.str();
    if(head.find(part[GUARD3]) == string::npos) {
        newheader << part[GUARD3] << endl;
        changed = true;
    }

    // 如果有变更，则覆盖文件：
    if(changed) {
        existh.close();
        ofstream newH(part[HEADER].c_str());
        assure(newH, part[HEADER].c_str());
        newH << "//@//\n" // 变更标记
             << newheader.str();
    }
}

if(!existcpp) { // 创建 .cpp 文件
    ofstream newcpp(part[IMPLEMENT].c_str());
    assure(newcpp, part[IMPLEMENT].c_str());
    newcpp << part[CPPLINE1] << endl
           << part[INCLUDE] << endl;
} else { // 文件已存在；验证它
    stringstream cppfile;
    ostringstream newcpp;
    cppfile << existcpp.rdbuf(); // 检查前两行是否符合标准：
    bool changed = false;
    string s;
    cppfile.seekg(0);
    getline(cppfile, s);
    bool lineUsed = false;

    for(int line = CPPLINE1; cppfile.good() && line <= INCLUDE; ++line) {
        if(startsWith(s, part[line])) {
            newcpp << s << endl;
            lineUsed = true;
            if(getline(cppfile, s))
                lineUsed = false;
        } else {
            newcpp << part[line] << endl;
            changed = true;
```

```
                    lineUsed = false;
                }
            }

            // 拷贝文件剩余内容
            if(!lineUsed)
                newcpp << s << endl;
            newcpp << cppfile.rdbuf();

            // 如果有更改，则覆盖文件：
            if(changed) {
                existcpp.close();
                ofstream newCPP(part[IMPLEMENT].c_str());
                assure(newCPP, part[IMPLEMENT].c_str());
                newCPP << "//@//\n" // 变更标记
                       << newcpp.str();
            }
        }
    }

    int main(int argc, char* argv[]) {
        if(argc > 1)
            cppCheck(argv[1]);
        else
            cppCheck("cppCheckTest.h");
    } ///:~
```

首先注意 startsWith()这个有用的函数。顾名思义，如果第一个字符串实参以第二个实参开头，那么该函数返回 true。查找预期的注释和与 include 相关的语句时，我们会用到该函数。有了字符串数组 part 后，我们就可以方便地遍历源代码中预期的一系列语句。如果源文件不存在，我们只需将这些语句写入给定名称的新文件中。如果文件已经存在，则会逐行搜索，验证存在预期的行。如果不存在，就将其插入。需要注意的是，必须确保不会误删已经有的行（注意布尔变量 lineUsed 在代码中的使用位置）。注意，我们使用 stringstream 来处理现有文件，这样可以先将文件的内容写入其中，再在其中读取和搜索。

下面是对枚举中的各个名称的解释：BASE 是去掉扩展名的大写基础文件名；HEADER 是头文件名（.h）；IMPLEMENT 是实现文件名（.cpp）；HLINE1 规定了头文件第一行的格式；GUARD1、GUARD2 和 GUARD3 是头文件中的“保护”行（用于防止重复包含）；CPPLINE1 规定了.cpp 文件第一行的格式；INCLUDE 则是.cpp 文件中用于包含.h 文件的那一行。

如果不带任何参数运行这个程序，将会创建两个文件。一个是头文件：

```
// CPPCHECKTEST.H
#ifndef CPPCHECKTEST_H
#define CPPCHECKTEST_H
```

```
#endif // CPPCHECKTEST_H
```

还有一个是实现文件：

```
// CPPCHECKTEST.CPP
#include "CPPCHECKTEST.H"
```

注意，这里去除了首行注释中的冒号，以免混淆为本书开发的代码提取器（参见 3.5 节）。在 Cppcheck 程序实际生成的输出中，这些冒号是存在的。

可以自行实验从这些测试文件中删除一些行，再重新运行程序。每次都会看到正确的行被重新添加回去。若文件发生修改，文件的第一行会变成字符串“//@//”来提醒您注意。再次处理文件之前，需要先删除这一行（否则 Cppcheck 程序会认为初始的注释行缺失）。

4.10.2 检测编译器错误

本书所有代码都设计为能够直接编译且无错误。[1]代码中会引起编译错误的行，将用特殊标记“//!”进行注释。以下程序可以删除这种特殊注释，并在行末追加一条带有编号的注释。编译所有文件时，如果编译器检测到这些标记为错误的代码行，它应该会生成相应的错误消息。在错误消息中，应该能看到我们添加的带有编号的注释。此外，该程序还会将修改后的行追加到一个特殊文件中，以便我们轻松定位这些存在错误的行。

```
//: C04:Showerr.cpp {RunByHand}
// 取消对有错误的行的注释
#include <cstddef>
#include <cstdlib>
#include <cstdio>
#include <fstream>
#include <iostream>
#include <sstream>
#include <string>
#include "../require.h"
using namespace std;

const string USAGE =
    "用法: showerr filename chapnum\n"
    "其中，filename 是 C++源文件，\n"
    "chapnum 是该文件所在章的编号。\n"
    "本程序查找//!形式的注释行，删除这个标记，\n"
    "并在行末追加一条带编号的注释，即//(#)。其中，# \n"
    "在所有文件中都是唯一的，以便判断\n"
    "编译器是否在指定文件中发现了错误。\n"
    "执行 showerr /r\n"
```

[1] 译注：中文版的配套代码同样经过了全面测试。

```
    "将重置这个全局唯一的计数器。";

class Showerr {
    const int CHAP;
    const string MARKER, FNAME;
    // 包含错误编号计数器的文件：
    const string ERRNUM;
    // 包含错误行的文件：
    const string ERRFILE;
    stringstream edited; // 已编辑的文件
    int counter;

public:
    Showerr(const string& f, const string& en, const string& ef, int c)
        : CHAP(c), MARKER("//!"), FNAME(f), ERRNUM(en), ERRFILE(ef), counter(0) {}

    void replaceErrors() {
        ifstream infile(FNAME.c_str());
        assure(infile, FNAME.c_str());
        ifstream count(ERRNUM.c_str());
        if (count) count >> counter;
        int linecount = 1;
        string buf;
        ofstream errlines(ERRFILE.c_str(), ios::app);
        assure(errlines, ERRFILE.c_str());
        while (getline(infile, buf)) {
            // 查找行首的标记：
            size_t pos = buf.find(MARKER);
            if (pos != string::npos) {
                // 删除标记：
                buf.erase(pos, MARKER.size() + 1);
                // 追加计数器和错误信息：
                ostringstream out;
                out << buf << " // (" << ++counter << ") "
                    << " 章号：" << CHAP
                    << " 文件：" << FNAME
                    << " 行：" << linecount << endl;
                edited << out.str();
                errlines << out.str(); // 在错误文件中追加数据
            } else {
                edited << buf << "\n"; // 仅拷贝
            }
            ++linecount;
        }
    }

    void saveFiles() {
        ofstream outfile(FNAME.c_str()); // 覆盖
        assure(outfile, FNAME.c_str());
        outfile << edited.rdbuf();
```

```
        ofstream count(ERRNUM.c_str()); // 覆盖
        assure(count, ERRNUM.c_str());
        count << counter; // 保存新的计数器
    }
};

int main(int argc, char* argv[]) {
    const string ERRCOUNT("../errnum.txt"), ERRFILE("../errlines.txt");
    requireMinArgs(argc, 1, USAGE.c_str());
    if (argv[1][0] == '/' || argv[1][0] == '-') {
        // 可以自行添加更多开关选项:
        switch (argv[1][1]) {
            case 'r': case 'R':
                cout << "重置计数器" << endl;
                remove(ERRCOUNT.c_str()); // 删除文件
                remove(ERRFILE.c_str());
                return EXIT_SUCCESS;
            default:
                cerr << USAGE << endl;
                return EXIT_FAILURE;
        }
    }

    if (argc == 3) {
        Showerr s(argv[1], ERRCOUNT, ERRFILE, atoi(argv[2]));
        s.replaceErrors();
        s.saveFiles();
    }
} ///:~
```

注意，可以将错误标记（“//!”）替换为自己选择的任何一个标记。

每个文件都会逐行读取，并且检查每一行的行首是否有特定的标记。如果有，该行会被修改，放入错误行列表中，再放入字符串流 `edited` 中。整个文件处理完毕后，它会在作用域结束时关闭。然后，它作为一个输出文件重新打开并接收 `edited` 的内容。另外，请注意计数器的值保存在一个外部文件中。再次调用这个程序时，计数器会继续递增。[①]

4.10.3 简单的数据记录程序

本例展示了一种将数据记录到磁盘，并在以后检索以进行处理的方法。它的目的是生成海洋不同地点的温度-深度 profile。具体数据由 `DataPoint` 类来保存。

```
//: C04:DataLogger.h
// Datalogger 程序的记录布局
```

[①] 译注：无论用于存储错误行的文件，还是存储错误编号的文件，都位于 Showerr 程序的上一级目录中。

```
#ifndef DATALOG_H
#define DATALOG_H
#include <ctime>
#include <iosfwd>
#include <string>
using std::ostream;

struct Coord {
    int deg, min, sec;
    Coord(int d = 0, int m = 0, int s = 0) : deg(d), min(m), sec(s) {}
    std::string toString() const;
};

ostream& operator<<(ostream&, const Coord&);

class DataPoint {
    std::time_t timestamp;     // 时间和日期
    Coord latitude, longitude; // 经纬度
    double depth, temperature; // 深度和温度

public:
    DataPoint(std::time_t ts, const Coord& lat,
              const Coord& lon, double dep, double temp)
        : timestamp(ts), latitude(lat), longitude(lon),
          depth(dep), temperature(temp) {}

    DataPoint() : timestamp(0), depth(0), temperature(0) {}
    friend ostream& operator<<(ostream&, const DataPoint&);
};
#endif // DATALOG_H ///:~
```

DataPoint 包含以下数据：作为 time_t 值存储的时间戳（该类型在<ctime>中定义）、经纬度坐标以及深度和温度值。我们使用插入符（<<）来进行格式化。以下是实现文件：

```
//: C04:DataLogger.cpp {O}
// Datapoint 类的实现
#include "DataLogger.h"
#include <iomanip>
#include <iostream>
#include <sstream>
#include <string>
using namespace std;

ostream& operator<<(ostream& os, const Coord& c) {
    return os << c.deg << '*' << c.min << '\''
              << c.sec << '"';
}
```

```
string Coord::toString() const {
    ostringstream os;
    os << *this;
    return os.str();
}

ostream& operator<<(ostream& os, const DataPoint& d) {
    os.setf(ios::fixed, ios::floatfield);
    char fillc = os.fill('0'); // 左侧用'0'填充
    tm* tdata = localtime(&d.timestamp);
    os << setw(2) << tdata->tm_mon + 1 << '\\'
       << setw(2) << tdata->tm_mday << '\\'
       << setw(2) << tdata->tm_year+1900 << ' '
       << setw(2) << tdata->tm_hour << ':'
       << setw(2) << tdata->tm_min << ':'
       << setw(2) << tdata->tm_sec;
    os.fill(' '); // 左侧用' '填充
    streamsize prec = os.precision(4);
    os << " Lat:"    << setw(9) << d.latitude.toString()
       << ", Long:"  << setw(9) << d.longitude.toString()
       << ", depth:" << setw(9) << d.depth
       << ", temp:"  << setw(9) << d.temperature;
    os.fill(fillc);
    os.precision(prec);
    return os;
} ///:~
```

Coord::toString()函数是必须的，因为 DataPoint 插入符在打印纬度和经度之前调用了 setw()。相反，如果改为使用流插入符来输出 Coord，那么宽度设置只对 Coord 结构体的第一个成员（即 Coord::deg）生效，因为宽度设置总是立即重置。[1]程序调用 setf()使得浮点数以固定精度（固定小数点格式）输出，并调用 precision()将小数位数设为 4 位。注意，在输出完 DataPoint 对象后，我们恢复了填充字符和精度设置，以确保后续的输出不受影响。

为了从 DataPoint::timestamp 存储的时间编码中获取值，我们调用了 std::localtime()函数，它返回指向 tm 对象的一个静态指针。tm 结构体的布局如下所示：

```
struct tm {
    int tm_sec;   // 0-59 秒
    int tm_min;   // 0-59 分钟
    int tm_hour;  // 0-23 小时
    int tm_mday;  // 一月中的天数，范围是 1-31
    int tm_mon;   // 0-11 月
    int tm_year;  // 自 1900 年以来的年数（因此实际年份需要加上 1900）
```

[1] 译注：setw()的设置是针对当前输出流的，而输出流的宽度设置在输出完一个对象后会自动重置。

```
    int tm_wday;  // 星期日==0，以此类推
    int tm_yday;  // 一年中的天数，范围是 0-365
    int tm_isdst; // 是否为夏令时？
};
```

生成测试数据

以下程序创建了一个二进制形式的测试数据文件（使用 write()）和一个 ASCII 形式的文件（使用 DataPoint 重载的插入符）。当然也可以将数据显示在屏幕上，但以文件形式更容易查看。[①]

```
// 测试数据生成程序
#include <cstdlib>
#include <ctime>
#include <cstring>
#include <fstream>
#include "DataLogger.h"
#include "../require.h"
using namespace std;

int main() {
  time_t timer;
  srand(time(&timer)); // 初始化随机数生成器种子
  ofstream data("data.txt");
  assure(data, "data.txt");
  ofstream bindata("data.bin", ios::binary);
  assure(bindata, "data.bin");
  for(int i = 0; i < 100; i++, timer += 55) {
    // 0 到 199 米：
    double newdepth  = rand() % 200;
    double fraction = rand() % 100 + 1;
    newdepth += 1.0 / fraction;
    double newtemp = 150 + rand() % 200; // 开尔文温度
    fraction = rand() % 100 + 1;
    newtemp += 1.0 / fraction;
    const DataPoint d(timer, Coord(45,20,31),
                      Coord(22,34,18), newdepth,
                      newtemp);
    data << d << endl;
    bindata.write(reinterpret_cast<const char*>(&d),
                  sizeof(d));
  }
} ///:~
```

[①] 译注：编译命令是 cl /EHsc Datagen.cpp DataLogger.cpp。

data.txt 创建为普通的 ASCII 文件，但 data.bin 使用 ios::binary 标志来告诉构造函数将其创建为二进制文件。为了理解文本文件中的数据格式，下面展示了 data.txt 的第一行（每次运行程序，结果都可能有所不同）：

```
12\17\2024 11:46:15 Lat:45*20'31", Long:22*34'18", depth: 149.0714, temp: 312.0294
```

标准 C 库函数 time()将它的实参指向的 time_t 值更新为当前时间的编码，这个时间在大多数平台上都是自 1970 年 1 月 1 日 00:00:00 GMT 以来经过的秒数。如本例所示，执行标准 C 库函数 srand()时，我们经常使用当前时间来初始化随机数生成器。

然后，在模拟中，每次循环都会使定时器的时间戳递增 55 秒，人为地营造一个读数间隔。

本例使用了固定经纬度，表明这是同一个采集点的一组读数。深度和温度都是通过标准 C 库函数 rand()生成的，该函数返回一个从零到常量 RAND_MAX 的伪随机数。注意，由<cstdlib>定义的 RAND_MAX 具体有多大要取决于具体的平台。但是，通常都是平台的最大无符号整数值。为了确保随机数处于目标范围内，我们使用了模除操作符%和范围的上限。但是，得到的数字是整数。为了添加小数部分，我们再次调用 rand()，并在加 1.0 之后取倒数（以防“除以零”错误）。[①]

本例中，二进制文件 data.bin 文件充当了程序生成的那些数据的存储容器，尽管这个容器位于磁盘上而非内存中。write()将数据以二进制形式写入磁盘。第一个实参是源数据块块（source block）的起始地址。它需要强制转换为 char*类型，因为 write()函数要求窄字节流。第二个实参是写入的字符数，本例就是 DataPoint 对象的大小（同样地，这是因为我们使用的是窄字节流[②]）。由于 DataPoint 中不含任何指针，因此将其直接写入磁盘是没有任何问题的。但是，如果对象更复杂，就应该考虑实现一个**序列化**方案。这种方案应该写入指针所引用的数据，并在以后读回时为其定义新的指针。本书暂不讨论序列化，大多数厂商提供的类库都内置了某种序列化结构。

验证和查看数据

为了验证存储二进制数据的有效性，可以使用输入流的 read()成员函数将其读入内存，并将其与早先由 Datagen.cpp 创建的文本文件进行比较。以下示例直接将格式化的结果写入 cout，但完全可以将其重定向到文件中，然后使用某个文件比较工具来验证它是否与原始

[①] 译注：以深度的生成为例。首先通过 rand() % 200 生成一个 0 到 199 的随机整数，作为深度的整数部分。然后，再生成一个 0 到 99 的随机整数，加 1.0 后取倒数，得到一个小数部分，与整数部分相加得到最终的深度值。这种方法可以生成 0 到 199.99 之间的随机深度值。温度的生成与此相似。

[②] 译注：由于使用的是窄字节流，因此 DataPoint 对象的大小（以字节为单位）就代表了需要写入的字节数。

文件一致。[①]

```
//: C04:Datascan.cpp
//{L} DataLogger
#include <fstream>
#include <iostream>
#include "DataLogger.h"
#include "../require.h"
using namespace std;

int main() {
    ifstream bindata("data.bin", ios::binary);
    assure(bindata, "data.bin");
    DataPoint d;
    while(bindata.read(reinterpret_cast<char*>(&d), sizeof d))
        cout << d << endl;
} ///:~
```

4.11 国际化

软件行业如今已成为一个健康的全球性经济市场，经常要求应用程序能在不同的语言和文化环境中运行。早在 20 世纪 80 年代末期，C 标准委员会就通过其**区域设置**（locale）[②]机制增加了对非美国格式约定的支持。“区域设置”是一组用于显示某些实体（如日期和货币金额）的偏好设置。到了 90 年代，C 标准委员会批准了一个对标准 C 的增补，规范了一些处理**宽字符**（由 `wchar_t` 类型表示）的函数，除了支持 ASCII 及其常见的西欧扩展之外，还能够支持其他字符集。尽管标准没有规定宽字符的具体大小，但在一些平台上，宽字符被实现为 32 位数据类型，因此可以容纳由 Unicode 联盟规定的编码以及由各种亚洲标准机构定义的多字节字符集。C++已在其 `iostream` 库中整合了对宽字符（wide character）和区域设置的支持。

4.11.1 宽流

宽流（wide stream）是一种处理宽字符的流类。本书到目前为止的所有示例（除了第 3 章最后一个 trait 示例之外）使用的都是容纳了 `char` 实例的**窄流**（narrow stream），或者称为“窄字节流”。由于流操作基本上都与底层字符类型无关，因此它们被泛型为模板。例如，基于这一大背景，所有输入流都由 `basic_istream` 类模板来定义。

```
template<class charT, class traits = char_traits<charT> > class basic_istream {...};
```

[①] 译注：编译命令是 `cl /EHsc Datascan.cpp DataLogger.cpp`。

[②] 译注：locale 有多种翻译，除了“区域设置”，还可以说成“国家和地区设置”等。

事实上，所有的输入流类型都是该模板的“特化”，这从它们的定义就可以看出：

```
// 有 w 前缀的处理的都是宽流（宽字符流、宽字节流）
typedef basic_istream<char> istream;
typedef basic_istream<wchar_t> wistream;
typedef basic_ifstream<char> ifstream;
typedef basic_ifstream<wchar_t> wifstream;
typedef basic_istringstream<char> istringstream;
typedef basic_istringstream<wchar_t> wistringstream;
```

其他所有流类型的定义都与此相似。

在一个理想的世界中，为了创建不同字符类型的流，我们只需要掌握这些知识就足够了。但现实给我们上了深刻的一课。事实上，为 char 和 wchar_t 提供的字符处理函数具有不同的名称。例如，为了比较两个窄字符串，我们使用 strcmp()函数。但对于宽字符，该函数的名称变成了 wcscmp()。记住，这并非 C++故意设计如此。造成这一切的根源都在于 C 语言。C 没有函数重载的概念，因此为了区分不同的函数，只能通过修改函数名来实现。出于这个原因，一个泛型的流无法在响应比较操作符时“无脑”地调用 strcmp()。必须有一种机制能自动调用正确的底层函数。

解决方案是化繁为简，对差异进行抽象。最终的结果就是，我们可以对字符执行的操作被抽象成了 char_traits 模板，该模板为 char 和 wchar_t 提供了预定义的特化（参见 3.4.5 节的讨论）。因此，要比较两个字符串，basic_string 只需调用 traits::compare()（记住，traits 是第二个模板参数），后者会根据当前使用的特化来调用 strcmp()或 wcscmp()（basic_string 不关心这个，这一步对它来说是透明的）。

除非需要访问底层字符处理函数，否则不用关心 char_traits；我们大多数时候都不在意这一点。不过，可以考虑将自己的插入符和提取符定义为模板，以防有人想把它们应用于宽流。像这样做可以使它们更“健壮”。

为了体会这一点，下面重新检视一下 4.2.1 节为 Date 类重载的插入符。当时它是这样声明的：

```
ostream& operator<<(ostream&, const Date&);
```

但这仅适用于窄流。为了使其更通用，我们简单地把它变成基于 basic_ostream 的模板：

```
template<class charT, class traits>
std::basic_ostream<charT, traits>& operator<<(
    std::basic_ostream<charT, traits>& os,
    const Date& d
) {
    charT fillc = os.fill(os.widen('0'));
    charT dash = os.widen('-');

    os << setw(2) << d.month << dash
```

```
            << setw(2) << d.day << dash
            << setw(4) << d.year;
        os.fill(fillc);
        return os;
    }
```

注意，还需要在 fillc 的声明中将 char 替换成模板参数 charT，因为它既可能是 char，也可能是 wchar_t，具体取决于所使用的模板“特化”。

由于在编写模板时并不知道具体会使用哪种类型的流，因此需要一种方法将字符字面值自动转换为适合当前流的正确大小。这正是 widen()成员函数的作用。例如，如果流是宽流，那么表达式 widen('-')会将其实参转换为 L'-'（相当于 wchar_t('-')，只不过表示成字面值[①]）；否则保持不变。除此之外，还有一个对应的 narrow()函数，它能在必要时将实参转换为 char。

可以利用 widen()为 4.9.2 节介绍的 nl（new line）操作符编写一个泛型版本，如下所示：

```
template<class charT, class traits>
basic_ostream<charT, traits>& nl(basic_ostream<charT, traits>& os) {
    return os << charT(os.widen('\n'));
}
```

4.11.2 区域设置

在全球范围内，不同地区在计算机数值输出的格式上存在显著差异，最典型的就是在一个实数中用于分隔整数和小数部分的符号。例如，美国习惯使用句点（.）表示小数点，而欧洲许多国家使用逗号（,）。这种地域差异给程序的本地化带来了不小的挑战。如果开发者试图手动处理所有可能的格式化情况，无疑会增加工作量且容易出错。幸好，C++标准委员会通过引入抽象层，有效地解决了这一问题，使得程序能适应这种差异。

这个抽象就是**区域设置**（locale）。所有流都有一个与之关联的区域设置对象，它们用来指导如何为不同的语言文化环境显示特定的数量值。区域设置管理着与语言文化相关的各种显示规则。表 4.11 总结了不同类别的规则。

表 4.11 不同类别的区域设置规则

规则类别	作用

[①] 译注：直接在字符前面加上 L 前缀，就表明这是一个宽字符。不需要像 wchar_t('-')那样执行显式的类型转换。

collate	不同语言的字符排序规则不同，collate 会根据不同的语言文化来决定如何比较两个字符串的大小。
cctype	对<cctype>中的字符分类和转换方式进行抽象。例如，判断一个字符是字母、数字还是标点符号。
monetary	以本地化的方式显示货币金额，其中包括货币符号。
numeric	支持实数的本地化显示格式，包括基数（小数点）和分组（千位）分隔符。
time	支持以本地化的格式显示日期和时间。
messages	提供一个框架来管理上下文相关的消息目录。例如，以不同的语言文化显示错误消息。

以下程序演示了基本的区域设置。

```
//: C04:Locale.cpp {-g++} {-bor} {-edg} {RunByHand}
// 演示区域设置（locale）的效果
#include <iostream>
#include <locale>
using namespace std;

int main() {
    locale def;

    // 看看当前 locale 什么
    cout << def.name() << endl;
    locale current = cout.getloc();
    cout << current.name() << endl;
    float val = 12.34;
    cout << val << endl;

    // 将 locale 更改为法语/法国
    cout.imbue(locale("fr-FR"));
    current = cout.getloc();
    cout << current.name() << endl;
    cout << val << endl;

    cout << "请输入字面值 123,45: " ;
    cin.imbue(cout.getloc());
    cin >> val;
    cout << val << endl;
    cout.imbue(def);
```

```
    cout << val << endl;
} ///:~
```

下面展示了该程序的一次示例运行结果：

```
C
C
12.34
fr-FR
12,34
请输入字面值 123,45: 123,45
123,45
123.45
```

默认区域设置是“C”，C 和 C++程序员对此一定不会感到陌生（它主要对应于英语语言和美国文化）。所有流最初都“注入”（imbue）了“C”区域设置。可以使用成员函数 imbue()来更改一个流的区域设置。注意，fr-FR 分为两部分，前者代表法语，后者代表法国（还有其他法语地区）。这个示例演示了在这个区域设置中，数值在显示时使用逗号作为小数点。如果希望按照该区域设置的规则进行输入，那么需要将 cin 也更改为相同的区域设置。

每一类区域设置都被细分为多个**方面**（facet），这些方面本质上是类，其中封装了与该类别相关的功能。例如，time 类别有 time_put 和 time_get 方面，其中包含用于时间和日期输入输出的功能。monetary 类别有 money_get、money_put 和 moneypunct 方面（最后那个方面决定了货币符号）。以下程序对 moneypunct 这个“方面”进行了演示。（跟时间有关的方面涉及复杂的迭代器操作，这超出了本章的范围。）

```
//: C04:Facets.cpp {-bor}{-g++}{-mwcc}{-edg}
// 更改货币符号
#include <iostream>
#include <locale>
#include <string>
using namespace std;

int main() {
    // 更改为法语/法国区域设置
    locale loc("fr-FR");
    cout.imbue(loc);
    string currency = use_facet<moneypunct<char> >(loc).curr_symbol();
    char point = use_facet<moneypunct<char> >(loc).decimal_point();
    cout << "我今天赚了" << currency << 123.45 << "欧！" << endl;
} ///:~
```

输出显示了欧元符号和逗号小数点：

```
我今天赚了€123,45 欧！
```

还可以定义自己的“方面”来构建自定义的区域设置。[①]需要注意的是，区域设置的开销相当大。事实上，一些库供应商提供了不同“口味”的标准 C++库以适应空间有限的环境。[②]

4.12 小结

本章全面介绍了 iostream 类库。或许仅凭本章所学的知识，就足以创建使用“输入输出流”的程序。需要注意的是，虽然 iostream 中的一些额外功能并不常用，但在需要时，可以通过查阅各种输入输出流类的头文件以及阅读编译器提供的 iostream 文档，或者参考本章及附录中提到的资料来快速学习。

4.13 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. 通过创建 ifstream 对象来打开一个英文文本文件。创建一个 ostringstream 对象，并使用 rdbuf()成员函数将文件中的所有内容读取到 ostringstream 中。提取底层缓冲区的一个 string 拷贝，并使用标准 C 在<cctype>中定义的 toupper()宏将文件中的每个字符都转换为大写。将结果写入一个新文件中。

2. 编写程序来打开一个文件（由命令行的第一个实参指定），在其中查找指定的一组字/词（由命令行的其余实参指定）。逐行读取输入，并将找到匹配项的行（带行号）写入一个新文件中。

3. 编写程序，在由程序命令行实参指定的所有源码文件的开头添加版权声明。

4. 使用自己熟悉的文本查找程序（例如，grep）在指定的所有文件中查找一个特定的模式。[③]。输出包含该模式的所有文件的文件名（只输出这个）。将输出重定向到一个文件中。编写一个程序，利用该文件的内容生成一个批处理文件，该批处理文件会在每个由程序找到的文件上调用编辑器。

[①] 可以参考 Langer & Kreft 的 *Standard C++ iostreams and Locales* 一书（Addison–Wesley，1999）。

[②] 例如，Dinkumware 的 Abridged 库（www.dinkumware.com）就省略了“区域设置”支持，而且异常支持也是可选的。

[③] 译注：例如，grep 'ho*'将查找以'h'开头,以任意多个'o'结尾的字符串。

5. 我们知道，setw()允许指定最小读取字符数。但是，如果想要指定最大读取字符数呢？编写一个效果器（参见 4.9 节），让用户能够指定提取的最大字符数。使这个效果器也能用于输出，以便在必要时截断输出字段，以保持在宽度限制内。

6. 写代码来证明如果设置了 failbit 或 badbit，然后为流启用异常，那么流会立即抛出异常（参见 4.3.2 节）。

7. 字符串流支持简单的转换，但这有一定的代价。编写一个程序，对比 atoi()和 stringstream 的转换系统，体会 stringstream 所产生的开销。[1]

8. 创建一个 Person 结构体，其中包含姓名、年龄和地址等字段。使用固定大小的数组来定义所有字符串字段。身份证号（或社保号）是每条记录的“键”。实现以下 DataBase 类：

```
class DataBase {
public:
    // 查找记录在磁盘上的位置
    size_t query(size_t ssn);

    // 返回 rn（记录编号）所代表的人
    Person retrieve(size_t rn);

    // 将一条记录记录到磁盘（文件）上
    void add(const Person& p);
};
```

将一些 Person 记录写入磁盘（不要将它们都保留在内存中）。当用户请求一条记录时，从磁盘上读取它并返回。DataBase 类中的 I/O 操作使用 read()和 write()来处理所有 Person 记录。

9. 为 Person 结构体重载 operator<<插入符，以便用一种易于阅读的格式显示记录。通过向一个文件中写入数据来演示它。

10. 假设存储了 Person 结构体的数据库丢失了，但在上个练习中写入的文件幸存下来了。使用该文件重建数据库。一定要进行错误检查。

11. 将 size_t(-1)（当前平台最大的无符号整数）写入一个文本文件 1 000 000 次。重复这一过程，但这次把它写入一个二进制文件。比较这两个文件的大小，看看使用二进制格式能节省多少空间。（最好先估算一下自己的平台上能节省多少空间，再和程序的实际运行

[1] 译注：这是一道很有意思的题。本书中文版在本章答案目录中提供了一个名为 ex07_CPP20.cpp 的程序，作为对原始答案 ex07.cpp 的补充。

结果对比。）[①]

12. 针对当前的 iostream 实现，调查精度最多可以有多少位。方法是在打印一个超越数（例如，sqrt(2.0)）[②]时，不断递增传递给 precision()的实参。

13. 编写一个程序，从文件中读取实数并打印它们的总和、平均值、最小值和最大值。

14. 先自行判断以下程序的输出，再实际执行来做对比：

```
//: C04:Exercise14.cpp
#include <fstream>
#include <iostream>
#include <sstream>
#include "../require.h"
using namespace std;

#define d(a) cout << #a " ==\t" << a << endl;

void tellPointers(fstream& s) {
    d(s.tellp());
    d(s.tellg());
    cout << endl;
}

void tellPointers(stringstream& s) {
    d(s.tellp());
    d(s.tellg());
    cout << endl;
}

int main() {
    fstream in("Exercise14.cpp");
    assure(in, "Exercise14.cpp");
    in.seekg(10);
    tellPointers(in);
```

[①] 译注：4 字节 unsigned int 的最大值为 4294967295，该数字的字符串长度为 10 个字符。若改为以二进制存储，每个数字占用 4 字节。基于这些信息，应该就可以估算存储 100 万个数字要占据多大空间了。本书中文版提供的 ex11_CPP20.cpp 程序对原始答案进行了完善。

[②] 译注：超越数是不能作为有理系数多项式方程的根的数，即不是代数数的数。这个名字来源于欧拉的话：“它们超越代数方法所及的范围之外。”（1748 年）更通俗地说，超越数就是那些不能通过有限次的加、减、乘、除、开方运算得到的数。它们“超越”了代数运算所能表达的范围。注意，超越数和代数数统称为“无理数”。另外，数学上的 √2 实际并不是超越数，但在计算机程序中，sqrt(2.0)或 sqrt(2)就成了超越数。原因请自行体会。

```
    in.seekp(20);
    tellPointers(in);
    stringstream memStream("Here is a sentence.");
    memStream.seekg(10);
    tellPointers(memStream);
    memStream.seekp(5);
    tellPointers(memStream);
}
```

15. 假设一个文本文件中包含按行组织的数据，如下所示：

```
//: C04:Exercise15.txt
Australia
5E56,7667230284,Langler,Tyson,31.2147,0.00042117361
2B97,7586701,Oneill,Zeke,553.429,0.0074673053156065
4D75,7907252710,Nickerson,Kelly,761.612,0.010276276
9F2,6882945012,Hartenbach,Neil,47.9637,0.0006471644
Austria
480F,7187262472,Oneill,Dee,264.012,0.00356226040013
1B65,4754732628,Haney,Kim,7.33843,0.000099015948475
DA1,1954960784,Pascente,Lester,56.5452,0.0007629529
3F18,1839715659,Elsea,Chelsy,801.901,0.010819887645
Belgium
BDF,5993489554,Oneill,Meredith,283.404,0.0038239127
5AC6,6612945602,Parisienne,Biff,557.74,0.0075254727
6AD,6477082,Pennington,Lizanne,31.0807,0.0004193544
4D0E,7861652688,Sisca,Francis,704.751,0.00950906238
Bahamas
37D8,6837424208,Parisienne,Samson,396.104,0.0053445
5E98,6384069,Willis,Pam,90.4257,0.00122009564059246
1462,1288616408,Stover,Hazal,583.939,0.007878970561
5FF3,8028775718,Stromstedt,Bunk,39.8712,0.000537974
1095,3737212,Stover,Denny,3.05387,0.000041205248883
7428,2019381883,Parisienne,Shane,363.272,0.00490155
```

每一节的标题都代表一个地区，该标题下的每一行都代表该地区的一个销售商。每个逗号分隔的字段都代表当前销售商的数据。数据行中的第一个字段是 SELLER_ID，注意它不巧使用了十六进制格式。第二个字段是 PHONE_NUMBER（注意，有些电话号码缺少区号）。接着是 LAST_NAME 和 FIRST_NAME。倒数第二列是 TOTAL_SALES。最后一个字段是该销售代表占公司总销售额的比例（一个小数）。请在终端窗口上格式化数据，使高管能轻松解读销售情况。下面给出一个示例输出：

```
                                    Australia
                                -------------------------
*Last Name*     *First Name*    *ID*    *Phone*         *Sales*         *Percent*
Langler         Tyson           24150    766-723-0284      31.21          4.21e-04
Oneill          Zeke            11159    XXX-758-6701     553.43          7.47e-03
(等等...)
```

第 5 章 深入理解模板

有的人或许以为 C++模板就是“T 的容器”，但它能做的事情远不止于此。尽管最初的动机是实现一个类型安全的、泛型的容器，但在现代 C++中，模板还可以用来生成自定义代码，并通过编译时的编程构造来优化程序执行。

本章将从实用的角度探讨现代 C++“模板编程”的强大功能（以及一些陷阱）。如果想更全面地了解与模板相关的语言问题和陷阱，推荐阅读 Daveed Vandevoorde 和 Nico Josuttis 的 C++ *Templates: The Complete Guide* 一书（Addison Wesley，2003）。

5.1 模板参数

如本书第 1 卷所述，模板主要分为两种：**函数模板**和**类模板**。它们的设计完全取决于其参数的定义。每个模板参数都可以属于以下类别之一：

1. 类型（内置类型或用户自定义类型）。
2. 编译时常量值（例如，整数以及指向静态实体的指针和引用；通常称为**非类型参数**）。
3. 其他模板。

本书第 1 卷的所有示例都属于第一类，也是最常见的。现在，许多人都喜欢用 `Stack` 类来举例说明简单的容器模板。作为容器，`Stack` 对象不关心它存储的是什么类型的对象。换言之，对象的容纳逻辑是一回事，具体容纳的是什么对象类型则是另一回事。因此，可以使用一个**类型参数**来表示所容纳的类型。

```
template<class T> class Stack {
    T* data;
    size_t count;
public:
    void push(const T& t);
    // 等等...
};
```

对于一个特定的 `Stack` 实例，其实际使用的类型是由参数（形参）`T` 的实参确定的：

```
Stack<int> myStack; //一个用于存储 int 的 Stack，类型实参是 int
```

当编译器遇到这个语句，会将 `T` 替换为 `int`，并生成相应的代码来提供一个 `int` 版本的 `Stack`。在本例中，从模板生成的类实例名称是 `Stack<int>`。

5.1.1 非类型模板参数

模板的非类型参数（non-type template parameter）必须是在编译时已知的一个常量值。例

如，可以指定一个非类型参数来作为底层数组的大小，从而创建一个固定大小的 Stack，如下所示：

```
template<class T, size_t N> class Stack {
    T data[N]; // 将容量固定为 N
    size_t count;
public:
    void push(const T& t);
    // 等等...
};
```

请求创建该模板的实例时，必须为参数 N 提供一个能在编译时确定的常量值，例如：

```
Stack<int, 100> myFixedStack;
```

由于 N 的值在编译时已知，所以可以将底层数组（data）放到运行时栈上，而不必放到自由存储区（堆）上。这样一来，由于避免了动态内存分配所带来的开销，因此运行时性能有一定的提高。遵循这一模式，最终的类名应该是 Stack<int, 100>。这意味着 N 的每个不同值都会产生唯一的**类类型**（class type）。例如，Stack<int, 99>就是一个与 Stack<int, 100>不同的类。

在标准 C++库中，唯一使用了非类型参数的类模板是 bitset（将在第 7 章详述）。该参数指定了 bitset 对象可以容纳的二进制位数。下面是一个随机数生成器的例子，它使用 bitset 来跟踪数字，使得在重新开始之前，范围内的所有数字都在不重复的情况下按随机顺序返回。这个例子还重载了 operator()，以支持我们熟悉的函数调用语法。

```
//: C05:Urand.h {-bor}
// 该随机数生成器生成不重复的随机数
#ifndef URAND_H
#define URAND_H

#include <bitset>
#include <cstddef>
#include <cstdlib>
#include <ctime>

using std::size_t;
using std::bitset;

template<size_t UpperBound> class Urand {
    bitset<UpperBound> used;

public:
    Urand() { srand(time(0)); } // 初始化随机数
    size_t operator()();        // “生成器”函数
};

template<size_t UpperBound>
```

```
inline size_t Urand<UpperBound>::operator()() {
    if (used.count() == UpperBound)
        used.reset(); // 重新开始（清空 bitset）

    size_t newval;

    while (used[newval = rand() % UpperBound])
        ; // 直到找到不重复的值

    used[newval] = true;
    return newval;
}

#endif // URAND_H ///:~
```

Urand 生成的数字是唯一的，因为 used 这个 bitset 跟踪了随机空间中所有可能的数字（上限由模板实参设置），并通过设置相应位置的二进制位来记录每个已使用的数字。当所有数字都被使用完后，bitset 将被清除以重新开始。下面这个简单的程序演示了如何使用 Urand 对象：

```
//: C05:UrandTest.cpp {-bor}
#include <iostream>
#include "Urand.h"
using namespace std;

int main() {
    Urand<10> u;
    for(int i = 0; i < 20; ++i)
        cout << u() << ' ';
}
///:~
```

本章稍后会讲到，非类型模板参数还能在在数值计算的优化中发挥重要作用。

5.1.2 默认模板实参

可以在**类模板**中为模板参数（形参）提供默认实参，但在**函数模板**中不允许这样做。和默认函数实参一样，它们只能定义一次。具体来说，是当模板声明或定义第一次被编译器“看到”时定义的。一旦引入了默认实参，后续所有模板参数也必须有默认值。例如，为了使前面展示的固定大小 Stack 模板更友好，可以像下面这样添加一个默认实参：

```
template<class T, size_t N = 100> class Stack {
    T data[N]; // 将容量固定为 N
    size_t count;
public:
    void push(const T& t); // Etc.
    // 等等...
};
```

现在，如果在声明 Stack 对象时省略了第二个模板实参，那么 N 的值将默认为 100。

可以选择为所有参数提供默认值，但在声明实例时必须使用一对空的尖括号，使编译器知道当前处理的是一个类模板，如下所示：

```
template<class T = int, size_t N = 100> // 两个类型参数都有默认值
class Stack {
    T data[N]; // 将容量固定为 N
    size_t count;
public:
    void push(const T& t);
    // 等等...
};

Stack<> myStack; // 等同于 Stack<int, 100>
```

标准 C++库广泛使用了默认参数。例如，vector 类模板是这样声明的：

```
template<class T, class Allocator = allocator<T> >
class vector;
```

注意最后两个右尖括号之间的空格，这是为了防止编译器将>>其解释为右移位操作符。[①]

上述声明表明 vector 接受两个实参。一个是它所容纳的对象的类型，另一个是代表 vector 所用的分配器的类型。如果省略第二个实参，那么会默认使用一个标准 allocator 模板[②]，该模板由 vector 模板类的第一个模板实参进行“参数化”[③]。从这个声明还可以看出，模板参数是可以重复使用的，就像本例的 T 那样。

虽然不能在函数模板中使用默认模板实参，但可以将模板参数作为普通函数的默认实参来

[①] 译注：从 C++11 开始，在闭合嵌套模板的时候，已经不再需要在右尖括号（>和>）号之间添加空格了。具体请参见维基百科“C++0x ”条目中的“Right angle bracket”一节，网址是 https://en.wikipedia.org/wiki/C++0x#Right_angle_bracket。正是由于这个原因，所以本章的中文版代码并不一定在两个>之间插入空格。

[②] 译注：即 allocator<T>，该模板类能根据具体的类型来进行内存分配，用于在 C++中进行内存管理。

[③] 译注：用实参替换形参，即为“参数化”。

传递。以下函数模板用于累加一个序列中的所有元素：[1]

```
//: C05:FuncDef.cpp
#include <iostream>
using namespace std;

template<class T>
T sum(T* b, T* e, T init = T()) {
    while(b != e)
        init += *b++;
    return init;
}

int main() {
    int a[] = { 1, 2, 13 };
    cout << sum(a, a + sizeof a / sizeof a[0]) << endl; // 16
}
///:~
```

传给 sum()的第三个实参是初始的元素累加值。由于这里省略了它，因此该参数默认为 T()。对于 int 和其他内置类型，T()会调用一个**伪构造函数**来执行“零初始化”（初始化为零或其他等价的初始值）。

5.1.3 将模板作为模板参数

模板可以接受的第三种类型的参数是其他类模板。如果打算在代码中将模板类型参数本身作为模板使用，那么编译器需要知道该参数是模板。下例演示了如何将模板用作模板参数：

```
//: C05:TempTemp.cpp
// 将模板作为模板参数
#include <cstddef>
#include <iostream>
using namespace std;

template<class T>
```

[1] 这个例子值得稍微解释一下。模板函数的第一个参数（b）和第二个参数（e）分别代表数组的开始和结束地址。在 main 中，为什么 a + sizeof a / sizeof a[0]是数组的结束地址？原因是 sizeof a 代表整型数组的总字节数，即 3 * sizeof(int)，而 sizeof a[0]代表单个 int 元素的字节数，即 sizeof(int)。因此，sizeof a / sizeof a[0] = 3，表示数组的长度为 3。最后，a + sizeof a / sizeof a[0]代表从数组的起始地址 a 开始向前移动 3 个位置，也就是数组最后一个元素之后的位置，即整个数组的结束地址。有一个基础知识不能忘记：在 C++ 中，指针可以进行算术运算。当一个指针加上一个整数时，指针会向后移动相应的字节数，每个位置的偏移量等于指针所指向的数据类型的字节数。

```
class Array { // 一个简单的、可扩展的序列
    enum { INIT = 10 };
    T* data;
    size_t capacity;
    size_t count;
public:
    Array() {
        count = 0;
        data = new T[capacity = INIT];
    }
    ~Array() {
        delete [] data;
    }
    void push_back(const T& t) {
        if(count == capacity) {
            // 扩展底层数组
            size_t newCap = 2 * capacity;
            T* newData = new T[newCap];
            for(size_t i = 0; i < count; ++i)
                newData[i] = data[i];
            delete [] data;
            data = newData;
            capacity = newCap;
        }
        data[count++] = t;
    }
    void pop_back() {
        if(count > 0)
            --count;
    }
    T* begin() { return data; }
    T* end() { return data + count; }
};

template<class T, template<class> class Seq>
class Container {
    Seq<T> seq;
public:
    void append(const T& t) {
        seq.push_back(t);
    }
    T* begin() {
        return seq.begin();
    }
    T* end() {
        return seq.end();
    }
};

int main() {
```

```
    Container<int, Array> container;
    container.append(1);
    container.append(2);
    int* p = container.begin();
    while(p != container.end())
        cout << *p++ << endl;
} ///:~
```

Array 类模板是一个简单的序列容器。Container 模板接受两个参数：容纳的数据类型，以及用于实际容纳数据的一个序列数据结构。在 Container 类的实现中，下面这一行告诉编译器 Seq 是模板：

```
Seq<T> seq;
```

如果没有将 Seq 声明为一个传给模板的模板参数[①]，编译器会在这里抱怨 Seq 不是模板，这是因为我们正在将其作为模板使用。在 main()中，我们实例化 Container，指定用 Array 来容纳整数值。因此，本例中的 Seq 就代表 Array。

注意，在这种情况下，就不需要在 Container 的声明中为 Seq 指定参数（形参）[②]名称了，也就是下面这一行：

```
template<class T, template<class> class Seq>
```

虽然也可以写成：

```
template<class T, template<class U> class Seq>
```

但参数 U 在任何地方都用不上。这里的重点在于，Seq 是一个获取单个类型参数的类模板。这类似于在不需要时省略函数的参数名，比如在重载后缀递增（后递增）操作符的时候：

```
T operator++(int);
```

这里的 int 只是一个占位符，因此无需命名。

以下程序改为使用固定大小的数组，它有一个额外的模板参数来表示数组长度：

```
//: C05:TempTemp2.cpp
// 将多变量模板参数传给模板
#include <cstddef>
#include <iostream>
using namespace std;
```

[①] 译注：原书称为“模板模板参数”（template template parameter），显得非常“拗口”。这也正是示例程序叫做 temptemp.cpp 的原因。

[②] 译注：本书中文版只在必要的时候才区分形参和实参。如果不影响理解，会直接说成“参数”。

```
template<class T, size_t N>
class Array {
    T data[N];
    size_t count;
public:
    Array() { count = 0; }
    void push_back(const T& t) {
        if(count < N)
            data[count++] = t;
    }
    void pop_back() {
        if(count > 0)
            --count;
    }
    T* begin() { return data; }
    T* end() { return data + count; }
};

template<class T, size_t N, template<class, size_t> class Seq>
class Container {
    Seq<T, N> seq;
public:
    void append(const T& t) { seq.push_back(t); }
    T* begin() { return seq.begin(); }
    T* end() { return seq.end(); }
};

int main() {
    const size_t N = 10;
    Container<int, N, Array> container;
    container.append(1);
    container.append(2);
    int* p = container.begin();
    while(p != container.end())
        cout << *p++ << endl;
} ///:~
```

再次强调，在 Container 的声明中，声明 Seq 时不需要指定其参数名，但我们需要两个参数来声明 Seq 类型的数据成员 seq，这正是在最外层出现了非类型参数 N 的原因。[①]

若将默认实参与作为模板传递的模板参数（模板模板参数）结合使用，情况会稍显复杂。当编译器匹配“模板模板参数”的内层参数时，不会自动考虑其默认实参。因此，为了实

[①] 译注：Container 类就是最外层的模板。Seq 则是 Container 类中的一个“模板模板参数”，它嵌套在 Container 内部。

现精确匹配，我们需要显式地重复这些默认实参。以下示例展示了如何为固定大小的 Array 模板设置默认实参，并说明如何在使用时处理这一情况。

```
//: C05:TempTemp3.cpp {-bor}{-msc}
// 作为模板传递的模板参数及其默认实参
#include <cstddef>
#include <iostream>
using namespace std;

template<class T, size_t N = 10> // 设置了一个默认实参
class Array {
    T data[N];
    size_t count;
public:
    Array() { count = 0; }
    void push_back(const T& t) {
        if(count < N)
            data[count++] = t;
    }
    void pop_back() {
        if(count > 0)
            --count;
    }
    T* begin() { return data; }
    T* end() { return data + count; }
};

// 必须重复默认实参
template<class T, template<class, size_t = 10> class Seq>
class Container {
    Seq<T> seq; // 这里使用了默认实参
public:
    void append(const T& t) { seq.push_back(t); }
    T* begin() { return seq.begin(); }
    T* end() { return seq.end(); }
};

int main() {
    Container<int, Array> container;
    container.append(1);
    container.append(2);
    int* p = container.begin();
    while(p != container.end())
        cout << *p++ << endl;
} ///:~
```

在下面这行代码中，必须重复指定默认大小 10：

```
template<class T, template<class, size_t = 10> class Seq>
```

无论 Container 中的 seq 定义，还是 main()中的 container 定义，它们都使用了默认实参。这是之前所描述之规则（即默认实参在一个编译单元中只应出现一次）的唯一例外。[①]注意，要想使用非默认实参，唯一的办法就是像之前的 TempTemp2.cpp 所展示的那样做。[②]

由于标准序列容器（vector、list 和 deque，所有这些都会在第 7 章更详细地讨论）设置了一个默认的 allocator（分配器）实参，因此在将其中某个序列作为模板实参传递时，上述技术非常有用。以下程序将 vector 和 list 分别传递给 Container 的两个实例：

```
//: C05:TempTemp4.cpp {-bor} {-msc}
// 将标准序列作为模板实参传递
#include <iostream>
#include <list>
#include <memory> // allocator<T>在这个头文件中声明
#include <vector>
using namespace std;

// 注意>和>之间从 C++11 起不用插入空格
template<class T, template<class U, class = allocator<U>> class Seq>
class Container {
    Seq<T> seq; // 隐式应用默认的 allocator<T>分配器
public:
    void push_back(const T& t) { seq.push_back(t); }
    typename Seq<T>::iterator begin() { return seq.begin(); }
    typename Seq<T>::iterator end() { return seq.end(); }
};

int main() {
    // 使用 vector
    Container<int, vector> vContainer;
    vContainer.push_back(1);
    vContainer.push_back(2);
    for(vector<int>::iterator p = vContainer.begin();
        p != vContainer.end(); ++p) {
        cout << *p << endl;
    }
    // 使用 list
```

[①] 译注：通常，一个默认实参只能在一个地方定义。但模板比较特殊，因为对于模板来说，模板实例化时会产生新的实例，而每个实例都可以有自己的默认实参。这个设计使我们在使用模板时有更大的灵活性。

[②] 译注：在那个程序中，我们显式地将一个自定义的序列类型作为 Seq 的模板实参传递给 Container，从而使用了非默认的序列类型。

```
    Container<int, list> lContainer;
    lContainer.push_back(3);
    lContainer.push_back(4);
    for(list<int>::iterator p2 = lContainer.begin();
        p2 != lContainer.end(); ++p2) {
        cout << *p2 << endl;
    }
} ///:~
```

在本例中，我们命名了内层模板 Seq 的第一个参数（命名为 U），因为标准序列中的分配器本身必须使用与序列中包含的对象相同的类型进行“参数化”。另外，由于默认 allocator 参数是已知的，因此可以在后续引用 Seq<T>时省略它，就像在之前的程序中所做的那样。然而，为了充分解释这个例子，我们必须理解 typename 关键字的语义。

5.1.4 typename 关键字

来看看下面这个示例程序：

```
//: C05:TypenamedID.cpp {-bor}
// 使用 typename 作为嵌套类型的前缀。

template<class T> class X {
    // 没有 typename 的话，这里会报错：
    typename T::id i;
public:
    void f() { i.g(); }
};

class Y {
public:
    class id {
        public:
        void g() {};
    };
};

int main() {
    X<Y> xy;
    xy.f();
} ///:~
```

模板定义假设传递给它的类 T 肯定有某种形式的一个嵌套标识符，这里命名为 id。然而，id 也可以是 T 的一个静态数据成员。在这种情况下，我们能直接对 id 进行操作，但不能“创建 id 类型的对象”。在本例中，标识符 id 被视为 T 内的一个嵌套类型。对于类 Y 来说，id 实际上是一个嵌套类型，但是，如果没有 typename 关键字的话，编译器在编译类

X 时是无法确定这一点的。[①]

当编译器在模板中看到一个标识符时，如果可以选择将其视为类型或其他非类型的东西，那么它会默认该标识符代表的是非类型的东西。换言之，它假定该标识符代表一个对象（包括基元类型[②]的变量）、枚举或其他类似的东西。但是，它不会（也不能）假定它是一个类型。

编译器默认会将符合上述条件的标识符视为非类型。因此，为了明确告诉编译器嵌套名称是一个类型，我们必须在它前面加上 typename 关键字（除非是在构造函数的初始化列表中，因为这里既不需要也不允许 typename）。在上例中，typename T::id 中的 typename 告诉编译器，id 是 T 的一个嵌套类型，因此允许创建该类型的对象。

该规则有一个简短的版本：如果模板代码内部引用的类型由一个模板类型参数限定[③]，那么必须使用 typename 关键字作为前缀，除非它出现在同一作用域内的基类规范[④]或初始化列表中（在这种情况下，不应使用 typename）。

现在，我们总算搞清楚了为什么 TempTemp4.cpp 程序要使用 typename 关键字。如果没有 typename，编译器将认为表达式 Seq<T>::iterator 不是类型。但是，由于这个位置必须是类型，因此会造成一个语法错误。

下例定义了一个可以打印任意标准 C++ 序列的函数模板，可以通过它巩固对 typename 的理解：

```
//: C05:PrintSeq.cpp {-msc} {-mwcc}
// printSeq 函数可以打印任意标准 C++序列
#include <iostream>
#include <list>
#include <memory>
#include <vector>
```

---

① 译注：如果没有 typename 的话，编译器可能将 T::id 解释为一个静态数据成员。更具体地说，模板参数 T 在编译时是未知的。编译器无法在编译模板本身时确定 T::id 的具体类型。因此，需要 typename 来告诉编译器，无论 T 是什么类型，T::id 都是一个类型，::不是域解析操作符。

② 译注：编译器直接支持的数据类型就是基元类型（primitive type），例如 int 和 bool 等。相反，auto 就不是基元类型，因为编译时需要分析初始化表达式，然后将 auto 替换为具体的类型。也可以将基元类型称为语言的“内置类型”。

③ 译注：换言之，类型的名称取决于模板参数的值。

④ 译注：例如，像 class Derived : public Base1, public Base2 { ... };这样的形式就是“基类规范”。

```
using namespace std;

template<class T, template<class U, class = allocator<U>> class Seq>
void printSeq(Seq<T>& seq) {
    for(typename Seq<T>::iterator b = seq.begin(); b != seq.end();)
        cout << *b++ << endl;
}

int main() {
    // 处理 vector
    vector<int> v;
    v.push_back(1);
    v.push_back(2);
    printSeq(v);

    // 处理 list
    list<int> lst;
    lst.push_back(3);
    lst.push_back(4);
    printSeq(lst);
} ///:~
```

同样地，如果没有 typename 关键字，编译器将会将 iterator 解释为 Seq<T>的静态数据成员，这是语法错误，因为此处需要类型。

用 typedef typename 定义新类型名称

请记住一个重点，不要假设 typename 关键字创建的是一个新的类型名。它并没有这样做。它的目的是告诉编译器限定的标识符应该被解释为类型。以下代码：

```
typename Seq<T>::iterator It;
```

会导致声明名为 It 的变量，其类型为 Seq<T>::iterator。如果想创建一个新的类型名，那么应该像平时那样使用 typedef，如下所示：

```
typedef typename Seq<It>::iterator It;
```

使用 typename 代替 class

typename 关键字的另一个作用是在模板定义的“模板实参列表”中提供用 typename 来代替 class 的选项：

```
//: C05:UsingTypename.cpp
// 在模板参数列表中使用'typename'。

template<typename T> class X {};
```

```
int main() {
    X<int> x;
} ///:~
```

一些人认为，这样写显得更清晰。

5.1.5 使用 template 关键字向编译器提示

编译器默认期望的不是类型标识符。为此，我们可以使用 typename 关键字来帮助进行解析。类似地，有时也会遇到不是标识符的标记，例如<和>字符。它们有时代表小于或大于符号，有时则用于界定模板实参列表。下例再次使用了 bitset 类：

```
//: C05:DotTemplate.cpp
// 演示.template 构造
#include <bitset>
#include <cstddef>
#include <iostream>
#include <string>
using namespace std;

template<class charT, size_t N>
basic_string<charT> bitsetToString(const bitset<N>& bs) {
    return bs. template to_string<charT, char_traits<charT>,
                                  allocator<charT> >();
}

int main() {
    bitset<10> bs;
    bs.set(1);
    bs.set(5);
    cout << bs << endl; // 0000100010
    string s = bitsetToString<char>(bs);
    cout << s << endl;  // 0000100010
} ///:~
```

bitset 类的 to_string 成员函数支持将该类的对象转换为 string。为了支持多种字符串类，to_string 本身也是一个模板，遵循由第 3 章讨论的 basic_string 模板所建立的模式。to_string 在 bitset 内部的声明如下所示：

```
template<class charT, class traits, class Allocator>
basic_string<charT, traits, Allocator> to_string() const;
```

上例的 bitsetToString()函数模板请求 bitset 的不同类型的字符串表示形式。例如，为了获取宽字符串，可以像下面这样修改调用：

```
wstring s = bitsetToString<wchar_t>(bs);
```

注意，basic_string 使用了默认模板实参，因此不需要在返回值中重复 char_traits 和

allocator实参了。但令人遗憾的是，bitset::to_string没有使用默认实参。如果能使用bitsetToString<char>(bs)，那么肯定比每次都输入完全限定的调用 bs.template to_string<char, char_traits, allocator<char> >()方便多了。

为此，我们在 bitsetToString()的返回语句的一个奇怪的位置使用了 template 关键字，它紧接在应用于 bitset 对象 bs 的点操作符之后。这是因为当解析模板时，to_string 后面的<字符会被解释为小于操作符而不是模板实参列表的开始。在这种上下文中使用 template 关键字，就可以告诉编译器后面跟着的是一个模板的名称，使<字符能被正确地解释。同样的推导也适用于模板的->和::操作符。和 typename 关键字一样，这种模板歧义消除技术只能在模板代码内使用。[①②]

5.1.6 成员模板

bitset::to_string()函数模板称为**成员模板**，也就是另一个类或类模板内部声明的模板。这使独立的模板实参可以有多种组合。标准 C++库中的 complex（复数）类模板展示了一个非常有用的例子。complex 模板的一个类型参数代表底层用什么浮点类型来存储复数的实部和虚部。以下是从标准库中摘录的代码片段，展示了 complex 类模板中的一个成员模板构造函数：

```
template<typename T> class complex {
public:
    template<class X> complex(const complex<X>&);
};
```

标准 complex 模板已经准备好了为 T 使用 float、double 和 long double 的特化。上述成员模板构造函数则可以创建一个使用不同浮点类型作为基类型的新复数，如下代码所示：

```
complex<float> z(1, 2);
complex<double> w(z);
```

在 w 的声明中，complex 模板参数 T 是 double，而 X 是 float。有了成员模板，我们可以非常灵活地进行这样的转换。

---

① C++标准委员会正在考虑放宽规则，不再限制这些用于消除歧义的提示只能在模板内部使用。有的编译器甚至已经在非模板代码中支持这些提示。

② 译注：C++20 引入的 Concepts（概念）使得模板的约束更加明确，减少了对 typename 和各种“提示”的依赖。详情请参见清华大学出版社出版的《学习 C++20（中文版）》一书或者 Bjarne Stroustrup 撰写的“Concepts: The Future of Generic Programming”一文（https://wg21.link/p0557r0）。本书中文版在配套代码的第 5 章目录中提供了一个名为 DotTemplate_CPP20.cpp 的程序，它演示了如何用 C++20 的 Concepts 特性修改原示例程序，编译命令是 cl /EHsc DotTemplate_CPP20.cpp /std:c++20。

在一个模板内定义另一个模板是一种嵌套操作，因此如果成员模板是在外层类定义的外部定义的，那么用于引入模板的前缀必须反映出这种嵌套。例如，在实现 complex 类模板时，如果成员模板构造函数是在 complex 模板类定义的外部定义的，那么要像下面这样写：

```
template<typename T>
template<typename X>
complex<T>::complex(const complex<X>& c) {/* 这里放函数体… */}
```

对于标准库中的“成员函数模板”来说，其另一种用途是对容器进行初始化。假设有一个容纳了 int 值的 vector，我们想用它来初始化一个新的 vector，并在其中容纳 double 值，如下所示：

```
int data[5] = { 1, 2, 3, 4, 5 };
vector<int> v1(data, data+5);
vector<double> v2(v1.begin(), v1.end());
```

只要 v1 中的元素与 v2 中的元素具有“赋值兼容性”（就像这里的 double 和 int 一样），那么一切都会很 OK。这是因为 vector 类模板提供了如下所示的成员模板构造函数：

```
template<class InputIterator>
vector(InputIterator first, InputIterator last, const Allocator& = Allocator());
```

在前面的向量声明中，该构造函数总共使用了两次。用 int 数组初始化 v1 时，类型 InputIterator 直接代表 int*。用 v1 初始化 v2 时，在所用的成员模板构造函数实例中，InputIterator 代表的则是 vector<int>::iterator。[①]

成员模板还可以是类。换言之，它们并不一定是函数。下例展示了外层类模板内的一个**成员类模板**：

```
//: C05:MemberClass.cpp
// 演示成员类模板
#include <iostream>
#include <typeinfo>
using namespace std;

template<class T>
class Outer {
public:
    template<class R>
    class Inner {
    public:
        void f();
```

[①] 译注：换言之，InputIterator 这个模板参数代表任意“输入迭代器”类型，可以是数组指针，也可以是 vector 迭代器等。

```
    };
};

template<class T>
template<class R>
void Outer<T>::Inner<R>::f() {
    cout << "外层类 == " << typeid(T).name() << endl;
    cout << "内层类 == " << typeid(R).name() << endl;
    cout << "当前对象的完整类型名称 == " << typeid(*this).name() << endl;
}

int main() {
    Outer<int>::Inner<bool> inner;
    inner.f();
} ///:~
```

typeid 操作符将在第 8 章介绍，它获取单一实参并返回一个 type_info 对象，该对象的 name()函数返回代表实参类型的字符串。例如，typeid(int).name()可能返回字符串"int"（实际返回什么字符串要取决于平台）。typeid 操作符也可以接收一个表达式，并返回代表该表达式（求值后）类型的type_info对象。例如，对于int i，typeid(i).name()将返回类似于"int"的字符串，而 typeid(&i).name()返回类似于"int*"的字符串。

上述程序的输出如下所示：

```
外层类 == int
内层类 == bool
当前对象的完整类型名称 == class Outer<int>::Inner<bool>
```

注意，在这个程序中，main 中的 inner 变量声明会同时实例化 Inner<bool>和 Outer<int>。

成员模板函数不能声明为 virtual（虚函数）。对类进行解析的时候，当前的编译器技术预期能够确定类的**虚函数表**大小。[1]若允许虚成员模板函数，就必须提前知道程序中所有对该成员函数的调用情况。而这是不现实的，特别是在多文件项目中。

5.2 函数模板的问题

类模板描述了一组类。类似地，函数模板描述了一组函数。创建这两种模板所用的语法几乎完全一样，但在用法上有所不同。实例化类模板时，必须使用尖括号，而且必须提供所有非默认的模板实参。然而，在函数模板中，经常都可以省略模板实参，而且甚至根本不

[1] 译注：现代 C++编译器仍然存在这个限制。

允许默认模板实参。[1]来考虑在<algorithm>头文件中声明的 min()函数模板的一个典型实现，如下所示：

```
template<typename T>
const T& min(const T& a, const T& b) {
    return (a < b) ? a : b;
}
```

为了调用这个模板，可以在尖括号内提供实参的类型，这和类模板是一样的，例如：

```
int z = min<int>(i, j);
```

这种语法告诉编译器需要 min()模板的一个特化，用 int 来替换参数 T。然后，编译器会生成相应的代码。按照由类模板生成的类的命名习惯，我们可以认为这个实例化的函数的名称是 min<int>()。

5.2.1 自动推断函数模板实参的类型

虽然可以始终像上一个例子那样，显式地为函数模板提供模板实参，但为了提高代码的可读性和简洁性，在实际编程中，我们通常会省略模板实参，让编译器根据传给函数的参数值来推断其类型，例如：

```
int z = min(i, j);
```

如果 i 和 j 都是 int 类型，那么编译器知道我们需要的是 min<int>()，因此会自动实例化它。要想成功推断，类型必须相同，因为模板为两个函数参数使用的是同一个模板类型实参。注意，如果传给函数的实参的类型是由模板参数指定的，那么不会执行标准转换。例如，如果想要找出一个 int 和一个 double 之间的最小值，以下 min()调用将会失败：

```
int z = min(x, j); // x 是一个 double
```

x 和 j 是不同的类型，无法用单一的类型来匹配 min()定义中的模板参数 T。因此，该调用和模板声明是不匹配的。为了解决这个问题，可以将其中一个实参强制转换为另一个实参的类型，或者使用完整调用语法，例如：

```
int z = min<double>(x, j);
```

这会告诉编译器生成 double 版本的 min()。然后，j 就可以借助标准转换规则提升为 double（因为此时存在匹配的函数 min<double>(const double&, const double&)）。

你可能会想，为什么不定义两个模板参数，让 min()接受两个类型相互独立的实参呢？例

[1] 译注：函数模板的重点在于，它能根据调用时提供的实参自动推导出形参 T 的实际类型。如果提供默认模板实参（比如 int），可能会影响这种推导过程。

如：

```
template<typename T, typename U>
const T& min(const T& a, const U& b) {
    return (a < b) ? a : b;
}
```

这在其他地方或许是一个好办法，但在这里会出问题，因为 min()必须返回一个值，但没有令人满意的方法来确定该值应该是什么类型（T 还是 U？）。

如果函数模板的返回类型是一个独立的模板参数，那么在调用函数时，必须总是显式地指定该参数的具体类型，因为没有可供从其中推断出该类型的实参。以下 fromString 模板就属于这种情况：[①]

```
//: C05:StringConv.h
// 以下两个函数模板支持 T 与 string 两种类型之间的相互转换
#ifndef STRINGCONV_H
#define STRINGCONV_H
#include <string>
#include <sstream>

template<typename T>
T fromString(const std::string& s) {
    std::istringstream is(s);
    T t;
    is >> t;
    return t;
}

template<typename T>
std::string toString(const T& t) {
    std::ostringstream s;
    s << t;
    return s.str();
}
#endif // STRINGCONV_H ///:~
```

这两个函数模板可以用于任何提供了流插入符（<<）或提取符（>>）的类型，从而实现与 std::string 之间的相互转换。以下测试程序用到了标准库的 complex（复数）类型：

```
//: C05:StringConvTest.cpp
#include <complex>
```

[①] 译注：在这个例子中，函数的返回类型 T 是一个独立的模板参数，与传入的字符串参数 s 之间没有直接的对应关系。编译器无法通过 s 的类型来推断出 T 的类型。因此，调用时必须显式指定 T 具体是什么类型。

```
#include <iostream>
#include "StringConv.h"
using namespace std;

int main() {
    int i = 1234;
    cout << "i == \"" << toString(i) << "\"" << endl;
    float x = 567.89;
    cout << "x == \"" << toString(x) << "\"" << endl;
    complex<float> c(1.0, 2.0);
    cout << "c == \"" << toString(c) << "\"" << endl;
    cout << endl;

    i = fromString<int>(string("1234"));
    cout << "i == " << i << endl;
    x = fromString<float>(string("567.89"));
    cout << "x == " << x << endl;
    c = fromString<complex<float> >(string("(1.0,2.0)"));
    cout << "c == " << c << endl;
} ///:~
```

程序的输出符合我们的预期：

```
i == "1234"
x == "567.89"
c == "(1,2)"

i == 1234
x == 567.89
c == (1,2)
```

注意，每次实例化 fromString()函数模板时，都要在调用中指定模板参数。如果函数模板的模板参数既用于参数类型也用于返回类型，那么一定要先声明返回类型参数，否则就无法省略函数参数的类型参数。下面这个著名的函数模板展示了一个例子：

```
//: C05:ImplicitCast.cpp
template<typename R, typename P>
R implicit_cast(const P& p) {
    return p;
}

int main() {
    int i = 1;
    float x = implicit_cast<float>(i);
    int j = implicit_cast<int>(x);
    // ! char* p = implicit_cast<char*>(i);
} ///:~
```

在文件顶部的模板参数列表中，如果将 R 和 P 交换一个位置，这个程序将无法通过编译，

因为返回类型将保持未指定状态（第一个模板参数成了函数的参数类型）。[1]另外，最后一行（注释掉的那一行）是非法的，因为不存在从 int 到 char*的标准转换。implicit_cast 只能表示那些天然就允许的转换。

小心一点，我们甚至能推断出数组的维度。下例用一个数组初始化函数模板（init2）来执行这种推断：

```
//: C05:ArraySize.cpp
#include <cstddef>
using std::size_t;

template<size_t R, size_t C, typename T>
void init1(T a[R][C]) {
    for(size_t i = 0; i < R; ++i)
        for(size_t j = 0; j < C; ++j)
            a[i][j] = T();
}

template<size_t R, size_t C, class T>
void init2(T (&a)[R][C]) { // 引用参数
    for(size_t i = 0; i < R; ++i)
        for(size_t j = 0; j < C; ++j)
            a[i][j] = T();
}

int main() {
    int a[10][20];
    init1<10,20>(a); // 必须指定大小
    init2(a);        // 自动推断大小
} ///:~
```

除非参数通过指针或引用传递，否则数组维度（每一维的大小）不会作为函数参数的类型的一部分传递。在本例中，函数模板 init2 将 a 声明为对一个二维数组的引用。因此，其维度 R 和 C（即行列数）可以由模板机制推断出来。采用这种方式，我们可以用 init2 来方便地初始化任意大小的二维数组。相反，由于模板 init1 不通过引用来传递数组，因此它的大小必须显式指定，尽管类型参数仍然可以推断。

5.2.2 函数模板重载

和普通函数一样，我们可以重载同名的函数模板。当编译器处理程序中的函数调用时，它

---

[1] 译注：模板参数的位置交换后，当调用 implicit_cast 函数时，编译器首先看到的是 P，这将作为实际传递给函数的参数值的类型。但是，返回类型 R 成了未指定的类型，这会导致编译错误，因为编译器需要一个明确的返回类型来进行类型检查。

必须决定哪个模板函数或普通函数是“最佳”匹配。[1]

在以下程序中，除了早先介绍的 min()函数模板，我们还添加了一些同名的普通函数：

```
//: C05:MinTest.cpp
#include <cstring>
#include <iostream>
using std::strcmp;
using std::cout;
using std::endl;

template<typename T> const T& min(const T& a, const T& b) {
  return (a < b) ? a : b;
}

const char* min(const char* a, const char* b) {
  return (strcmp(a, b) < 0) ? a : b;
}

double min(double x, double y) {
  return (x < y) ? x : y;
}

int main() {
  const char *s2 = "say \"Ni-!\"", *s1 = "knights who";
  cout << min(1, 2) << endl;      // 1: 1 (自动调用模板版本的 min)
  cout << min(1.0, 2.0) << endl;  // 2: 1 (double)
  cout << min(1, 2.0) << endl;    // 3: 1 (double)
  cout << min(s1, s2) << endl;    // 4: knights who (const char*)
  cout << min<>(s1, s2) << endl;  // 5: say "Ni-!"
                                  //    (显式调用模板版本的 min)
} ///:~
```

除了函数模板，本程序还定义了两个非模板函数：一个 C 风格字符串版本的 min()和一个 double 版本。如果没有模板存在，第 1 行（参见注释中的编号）会调用 double 版本的 min()，因为标准转换会将 int 转换为 double。但是，模板可以生成一个 int 版本，因为这被认为是更好的匹配。第 2 行的调用与 double 版本完全匹配，第 3 行也调用了相同的函数，隐式地将 1 转换为 1.0。第 4 行则直接调用了 const char*版本的 min()。在第 5 行，我们通过在函数名后加上一对空的尖括号来强制编译器使用模板机制，这迫使它从模板生成一个 const char*版本并使用它（这一点可以通过错误的答案得到验证——这个版本只

[1] 译注：“函数模板”和“模板函数”有细微的区别。前者是模板，后者是基于该模板实例化的函数。

会比较地址!)。[1][2]

如果好奇什么本例单独使用了一些 using 声明而不是直接使用一个 using namespace std 指令，那是因为一些编译器会自动包含一些头文件，这些头文件引入了 std::min()，会与我们的 min()声明冲突。

如前所述，只要编译器能够区分，就可以重载同名的模板。例如，可以声明一个处理三个实参的 min()函数模板：

```
template<typename T> const T& min(const T& a, const T& b, const T& c);
```

只有使用同类型三个实参调用 min()时，才会基于该模板来实例化函数。

5.2.3 获取所生成的函数模板的地址

我们有时需要获取函数的地址。例如，假定有一个函数需要接收指向另一个函数的指针作为参数，而后者可能是从模板函数生成的。因此，需要以某种方式来获取这种地址，如下例所示：[3]

```
//: C05:TemplateFunctionAddress.cpp {-mwcc}
// 获取从模板生成的函数的地址
template<typename T> void f(T*) {}

void h(void (*pf)(int*)) {}

template<typename T> void g(void (*pf)(T*)) {}

int main() {
    h(&f<int>);      // 完全类型限定
    h(&f);           // 类型推断
    g<int>(&f<int>); // 完整类型限定
    g(&f<int>);      // 类型推断
    g<int>(&f);      // 部分（但足够的）限定
} ///:~
```

可以从本例看出不少问题。首先，即使使用了模板，签名也必须匹配。h()函数接收一个指针，它指向获取 int*并返回 void 的一个函数。模板 f()生成的正是这样的函数。其次，

---

[1] 从技术角度来说，比较两个不在同一个数组内的指针是未定义行为，但现代编译器通常不会对此发出警告。这更强调了正确执行此操作的重要性。

[2] 译注：模板版本通过指针来比较指向的内存地址。相反，普通的 const char*版本在内部使用 strcmp()函数来逐个字符地比较，这样获得的才是正确的结果。

[3] 感谢 Nathan Myers 提供这个例子。

想要接收函数指针作为实参的函数本身也可以是一个模板，模板 g()对此进行了演示。

在 main()中，可以看到类型推断在这里也起作用。第一次调用 h()时，为 f()模板显式指定了类型实参 int。但是，由于 h()声明它只会接受指向“获取 int*的一个函数”的地址，因此这部分可由编译器推断。[①]g()的情况则更有趣，因为它涉及到两个模板。编译器没有任何信息来推断类型，但如果为 f()或 g()给定了 int，那么其余部分都可以被推断出来。

尝试将<cctype>中声明的 tolower 或 toupper 函数作为参数传递时，会出现一个较为复杂的问题。例如，可以将这些函数与 transform 算法一起使用（下一章会详细讨论该算法），从而将字符串转换为小写或大写。但这样做的时候必须非常小心，因为这些函数有多个声明。一个简单的尝试可能是这样的：

```
// 变量 s 是一个 std::string
transform(s.begin(), s.end(), s.begin(), tolower);
```

transform 算法将其第 4 个实参（此处为 tolower()）应用到字符串 s 中的每个字符，并将结果放回 s 本身中，从而将 s 中的每个字符覆盖为其小写形式。但如果真的这样做，该语句可能成功，也可能失败！下例展示了它可能失败的一种情况：

```
//: C05:FailedTransform.cpp {-xo}
#include <algorithm>
#include <cctype>
#include <iostream>
#include <string>
using namespace std;

int main() {
    string s("LOWER");
    transform(s.begin(), s.end(), s.begin(), tolower);
    cout << s << endl;
} ///:~
```

即使编译器允许这样写，但这也是不合理的。[②]原因是<iostream>头文件还提供了 tolower()和 toupper()的双参数版本：

```
template<class charT> charT toupper(charT c, const locale& loc);
template<class charT> charT tolower(charT c, const locale& loc);
```

这些函数模板接收 locale 类型的第二个参数。编译器没有办法知道是应该使用<cctype>

[①] 译注：f 被隐式地特化为 f<int>。

[②] 译注：现代 C++编译器不会对这个程序报错，也能生成正确的结果。但是，确实可以利用现代 C++的特性来重写该程序，完全排队歧义。稍后的“译注”会提供一个更符合现代 C++标准的程序。

中定义的单参数版本的 tolower()，还是应该使用上面提到的那个版本。为此，可以在对 transform 的调用中执行一次强制类型转换。这在大多数时候能解决问题。

```
transform(s.begin(), s.end(), s.begin(), static_cast<int (*)(int)>(tolower));
```

记住，tolower()和 toupper()是基于 int 而不是 char 来操作的。上述强制类型转换清楚地表明希望使用单参数版本的 tolower()。这在某些编译器上可行，但并非所有编译器都支持。原因在于，库的实现可能为从 C 语言继承的函数提供“C 链接性”（即 C linkage，意味着函数名不包含像普通 C++函数那样多的辅助信息[①]），如果属于这种情况，那么类型转换会失败，因为 transform 是一个 C++函数模板，期望其第 4 个参数具有“C++链接性”，而且类型转换不允许改变这个链接性。真是棘手！

解决方案是将 tolower()调用放到一个无歧义的上下文中。例如，可以编写一个名为 strTolower()的函数，并把它放到自己的文件中而不包含<iostream>，如下所示：

```
//: C05:StrTolower.cpp {O} {-mwcc}
#include <algorithm>
#include <cctype>
#include <string>
using namespace std;

string strTolower(string s) {
    transform(s.begin(), s.end(), s.begin(), tolower);
    return s;
} ///:~
```

这里没有涉及<iostream>，并且我们使用的编译器在此上下文中不会引入 tolower()的双参数版本，[②]因此不会出任何问题。然后，我们就可以正常地使用这个函数了，如下例所示：[③]

```
//: C05:Tolower.cpp {-mwcc} //{L} StrTolower
```

[①] 译注：这些“辅助信息”指的是编译器为了区分重载函数而在函数名中加入的类型信息等。而在 C 链接的情况下，这些信息不会被包含在函数名中。例如，void foo(int x);和 void foo(double x);这两个函数声明，它们可能会被装饰成不同的名字，比如_foo_int 和_foo_double，这样编译器就可以通过名字来区分它们。相反，如果使用了“C 链接性”，比如 extern "C" { void c_function(int x); }，那么 c_function 在 C++中就会有与 C 语言相同的链接名称，不会被装饰，从而能够在 C++中正确调用 C 库中的函数。但是，代价就是无法区分同名的不同重载版本了。记住，C 语言不支持函数重载的概念，所有函数都必须有不同的名字。

[②] 然而，C++编译器可以在它们想要的任何地方引入名称。幸好，大多数都不会声明它们不需要的名称。

[③] 译注：编译命令是 cl /EHsc Tolower.cpp StrTolower.cpp。

```
#include <algorithm>
#include <cctype>
#include <iostream>
#include <string>
using namespace std;
string strTolower(string);

int main() {
    string s("LOWER");
    cout << strTolower(s) << endl;
} ///:~
```

另一个解决方案是编写一个显式调用正确版本 tolower()的包装函数模板：

```
//: C05:ToLower2.cpp {-mwcc}
#include <algorithm>
#include <cctype>
#include <iostream>
#include <string>
using namespace std;

template<class charT> charT strTolower(charT c) {
    return tolower(c); // 调用单参数版本
}

int main() {
    string s("LOWER");
    transform(s.begin(), s.end(), s.begin(), &strTolower<char>);
    cout << s << endl;
} ///:~
```

这个版本的优势在于，它可以同时处理宽字符串和窄字符串，因为底层字符类型是一个模板参数。C++标准委员会正在修改语言，使得第一个示例（不带强制类型转换）也能正常工作，未来这些“临时性的变通方案”应该可以忽略了。[①]

[①] 译注：如前所述，现代 C++确实已经解决了这个问题。现在，甚至可以使用 C++11 引入的 lambda 表达式、自定义函数对象以及 C++20 引入的 std::ranges 来重写第一版的程序，从而简化编码。本书中文版配套代码的第 5 章目录提供了一个名为 StrTolower_CPP20.cpp 的程序，它对这些新技术进行了演示。注意，为了使用 C++20 特性，请使用以下编译命令：cl /EHsc StrTolower_CPP20.cpp /std:c++20。

5.2.4 将函数应用于 STL 序列

假设要对 STL 序列容器（后续各章会更多地讨论这个主题；目前只需使用熟悉的 vector 即可）中的所有对象应用（该对象的）一个成员函数。vector 可以包含任意类型的对象，因此需要一个支持任何类型的 vector 的函数（模板）。在下例中，该函数（模板）命名为 apply：

```
//: C05:ApplySequence.h
// 将一个函数应用于 STL 序列容器

// const、无参、任意返回值类型的成员函数 f：
template<class Seq, class T, class R>
void apply(Seq& sq, R (T::*f)() const) {
    typename Seq::iterator it = sq.begin();
    while(it != sq.end())
        ((*it++)->*f)();
}

// const、单个参数、任意返回值类型的成员函数 f：
template<class Seq, class T, class R, class A>
void apply(Seq& sq, R(T::*f)(A) const, A a) {
    typename Seq::iterator it = sq.begin();
    while(it != sq.end())
        ((*it++)->*f)(a);
}

// const、两个参数、任意返回值类型的成员函数 f：
template<class Seq, class T, class R, class A1, class A2>
void apply(Seq& sq, R(T::*f)(A1, A2) const, A1 a1, A2 a2) {
    typename Seq::iterator it = sq.begin();
    while(it != sq.end())
        ((*it++)->*f)(a1, a2);
}

// 非 const、无参、任意返回值类型的成员函数 f：
template<class Seq, class T, class R>
void apply(Seq& sq, R (T::*f)()) {
    typename Seq::iterator it = sq.begin();
    while(it != sq.end())
        ((*it++)->*f)();
}

// 非 const、单个参数、任意返回值类型的成员函数 f：
template<class Seq, class T, class R, class A>
void apply(Seq& sq, R(T::*f)(A), A a) {
    typename Seq::iterator it = sq.begin();
    while(it != sq.end())
        ((*it++)->*f)(a);
}
```

```
// 非 const、两个参数、任意返回值类型的成员函数 f：
template<class Seq, class T, class R, class A1, class A2>
void apply(Seq& sq, R(T::*f)(A1, A2), A1 a1, A2 a2) {
    typename Seq::iterator it = sq.begin();
    while(it != sq.end)
        ((*it++)->*f)(a1, a2);
}
// 等等……以支持可能的最大参数数量  ///:~
```

上述 apply()函数模板获取一个容器类引用和一个函数指针，后者指向容器所容纳对象的成员函数。它使用迭代器遍历序列，并将成员函数应用于每个对象。我们已根据成员函数是否为 const 进行了重载，因此 const 和非 const 成员函数都可以使用。

注意，ApplySequence.h 中没有 include 任何 STL 头文件（事实上，没有 include 任何头文件），因此它并非只能用于 STL 容器。不过，它确实做出了一些更适合 STL 序列的假设（主要是 iterator 的名称和行为）。除此之外，它还假设容器中的元素为指针类型。

可以看出，apply()有多个版本，从中可以更深入地体会函数模板的重载。虽然这些模板不限制成员函数 f 的返回值类型（虽然在函数模板中没有用上这个返回值，但为了匹配指向成员函数的指针，返回值的类型信息还是必要的），但 apply()的每个版本都需要提供不同数量的实参。并且，由于它是模板，所以这些实参可以是任何类型。唯一的限制是没有“超级模板”可以帮助我们创建能适应所有情况的模板。相反，必须自行决定程序需要多少个实参，并手动进行相应的定义。

为了测试 apply()的各种重载版本，我们创建了一个名为 Gromit 的类[①]，其中包含的函数具有不同数量的实参，而且成员函数既有 const 的，也有非 const 的。

```
//: C05:Gromit.h
// Gromit（阿高）是一只科技狗（techno-dog），其成员函数能获取不同数量的实参
#include <iostream>
class Gromit {
    int arf;
    int totalBarks;
public:
    // 成员变量 arf 被初始化为 arf + 1
    Gromit(int arf = 1) : arf(arf + 1), totalBarks(0) {}

    // 让阿高叫：根据 arf（汪）的值来决定叫多少声
```

[①] 译注：Gromit 这个名称来自英国著名的黏土定格动画喜剧系列 *Wallace and Gromit*（超级无敌掌门狗）。Gromit（阿高）是业余发明家 Wallace（华莱士）的一只聪明且忠诚的拟人化小猎犬，经常参与 Wallace 的各种奇思妙想的发明。它不仅能理解主人的意图，还能提供建设性的意见，甚至能帮助主人解决发明过程中遇到的问题。

```
    // 每个对象根据其实例化时的 arf 值来输出不同次数的 "汪! "
    void speak(int) {
        for(int i = 0; i < arf; i++) {
            std::cout << "汪! ";
            ++totalBarks;
        }
        std::cout << std::endl;
    }

    // 模拟阿高吃饭
    // const 声明该函数不会修改对象的成员变量
    char eat(float) const {
        std::cout << "我吃吃吃!" << std::endl;
        return 'z';
    }

    // 模拟阿高睡觉
    // const 声明该函数不会修改对象的成员变量
    int sleep(char, double) const {
        std::cout << "zzz..." << std::endl;
        return 0;
    }

    // 模拟阿高坐下
    // const 声明该函数不会修改对象的成员变量
    void sit() const {
        std::cout << "正在坐下..." << std::endl;
    }
}; ///:~
```

然后，在 apply()模板函数的各种版本中，我们可以将 Gromit 的各种成员函数应用于一个 vector<Gromit*>，如下所示：

```
//: C05:ApplyGromit.cpp
// 测试 ApplySequence.h
#include <cstddef>
#include <iostream>
#include <vector>
#include "ApplySequence.h"
#include "Gromit.h"
#include "../purge.h"
using namespace std;

int main() {
    vector<Gromit*> dogs;
    for(size_t i = 0; i < 5; i++)
        dogs.push_back(new Gromit(i));
    apply(dogs, &Gromit::speak, 1);
    apply(dogs, &Gromit::eat, 2.0f);
    apply(dogs, &Gromit::sleep, 'z', 3.0);
```

```
    apply(dogs, &Gromit::sit);
    purge(dogs);
} ///:~
```

purge()函数是一个小的辅助工具，它对序列中的每个元素调用 delete。第 7 章会展示它的定义，本书许多地方都用到了该函数。

虽然 apply()的定义有些复杂，初学者可能不容易理解，但在实际使用中却非常简洁高效。初学者可以直接使用它，只需了解它预期能完成的任务，而无需深入探究其具体实现细节。事实上，所有程序组件都应该这样设计。也就是说，复杂的细节被隔离在设计者这一边，用户只需要关注如何实现自己的目标，而无需看到、知道或依赖底层实现的细节。我们将在下一章探讨如何使用一些更灵活的方法将函数应用于序列。

5.2.5 函数模板的部分排序

之前提到过，min()的普通函数重载比模板更好。如果已经有了与函数调用匹配的函数，为什么还要专门从函数模板生成另一个呢？但是，在没有普通函数的情况下，重载函数模板可能导致歧义。为了避免这种情况，语言的设计者制定了一个函数模板排序规则，它每次都选择“最特化”的模板（如果有的话）。如果一个函数模板能够匹配的每种可能的实参列表也能匹配另一个模板，但反过来则不行，那么前一个模板被认为比后一个更特化。考虑以下函数模板声明，它摘自 C++标准文档中的一个示例：

```
template<class T> void f(T);
template<class T> void f(T*);
template<class T> void f(const T*);
```

第一个模板可以匹配任何类型。第二个模板比第一个更特化，因为只有指针类型才能匹配它。换言之，能匹配第二个模板的所有可能的调用都是第一个模板的子集。第二和第三个模板声明也存在类似的关系：第三个只支持指向常量的指针[①]，但第二个支持任何类型的指针。以下程序说明了这些规则：

```
//: C05:PartialOrder.cpp
// 演示函数模板的排序规则
#include <iostream>
using namespace std;

template<class T> void f(T) {
    cout << "T" << endl;
}
```

[①] 译注：记住，const T*是指向常量的指针，而 T* const 是常量指针。前者不能通过指针修改数据，但可以修改指针的指向；后者不能改变指针的指向，但可以通过指针修改数据。

```
template<class T> void f(T*) {
    cout << "T*" << endl;
}

template<class T> void f(const T*) {
    cout << "const T*" << endl;
}

int main() {
    f(0);            // T
    int i = 0;
    f(&i);           // T*
    const int j = 0;
    f(&j);           // const T*
} ///:~
```

`f(&i)`这个调用肯定也和第一个模板匹配，但由于第二个模板更特化，所以选择了第二个。之所以不能选择第三个模板，是因为该指针不是指向常量（`const`）的。`f(&j)`这个调用和全部三个模板都匹配（例如，在第二个模板中，`T` 将是 `const int`）但是，由于第三个模板更特化，所以选择了第三个。

如果一组重载函数模板中没有“最特化”的模板，就会出现歧义，编译器会报错。这就是为什么这个特性称为“部分排序”的原因——它可能无法解决所有可能性。[①]类似的规则也适用于类模板（参见稍后的 5.3.2 节“偏特化”）。

5.3 模板特化

特化（specialization）这个术语在 C++中具有特定的、与模板相关的含义。模板定义本质上就是一种泛化（generalization），因为它使用通用或者泛化（general）的措辞描述了一组函数或类。一旦提供模板实参，就得到了模板的一种特化，因为它确定了在这一组函数或类中中，众多可能实例中的唯一实例。本章开头展示的 `min()`函数模板就是对“查找最小值”函数的一种泛化，因为在定义时，它的参数的类型没有明确指定（因此称为“形参”）。为模板参数（形参）提供实际的类型时，无论是显式提供，还是通过实参推导机制来隐式地提供，编译器所生成的结果代码（例如`min<int>()`）就是该模板的一种特化。生成的代码也被认为是对模板的一种**实例化**。事实上，所有通过模板机制生成的代码都可以称为对

[①] 译注：部分排序或者偏序（Partial Ordering）是一种数学概念，用于比较元素之间的关系。与全序（Total Ordering）不同，部分排序不要求所有元素都能相互比较。

模板的实例化。[1]

5.3.1 显式特化

在必要时，我们可以自己为给定的模板特化提供代码。特别是类模板，它通常需要显式特化。但是，我们打算从 min()函数模板开始介绍这种语法。

在本章早些时候的 MinTest.cpp 程序中，我们引入了以下普通（非模板）函数：

```
const char* min(const char* a, const char* b) {
    return (strcmp(a, b) < 0) ? a : b;
}
```

这是为了让 min()的调用比较字符串而不是地址。虽然没有什么特别的好处，但完全可以修改上述代码，为 min()定义一个 const char*特化，如以下程序所示：

```
//: C05:MinTest2.cpp
#include <cstring>
#include <iostream>
using std::strcmp;
using std::cout;
using std::endl;

template<class T> const T& min(const T& a, const T& b) {
    return (a < b) ? a : b;
}

// min 模板的一个显式特化
template<>
const char* const& min<const char*> (const char* const& a,
                                     const char* const& b) {
    return (strcmp(a, b) < 0) ? a : b;
}

int main() {
    const char *s2 = "say \"Ni-!\"";
    const char *s1 = "knights who";
    cout << min(s1, s2) << endl;
    cout << min<>(s1, s2) << endl;
} ///:~
```

[1] 译注：如果想拓展这个主题的话，模板的特化还可以进一步分为全特化和偏特化。全特化是对模板的所有参数进行特化，而偏特化是对部分参数进行特化。后者是 5.3.2 节“部分特化”的主题。“部分”和“偏”在这里是同义词，在英语中都是 partial。只是因为历史原因，本书使用了“偏特化”而不是“部分特化”。

template<>前缀告诉编译器后面跟着的是模板的特化。具体的特化类型必须出现在函数名后的尖括号内，这和显式指定了类型的调用是一样的。注意，在显式特化中，我们仔细地将 T 替换成了 const char*。当原始模板指定 const T 时，该 const 修饰的是完整的类型 T 而不是指针。换言之，T 的实例必须是一个常量。整个 const T 则代表一个指针，该指针指向该常量。因此，在特化中，必须写 const char* const[①]以代替 const T。当编译器看到程序中以 const char*为实参的 min()调用时，它就会实例化这个 const char*版本的 min()以便可以调用它。在上述程序中，两个 min()调用都调用了同一特化版本的 min()。

相较于函数模板，类模板更需要显式特化。但是，当我们为类模板提供全特化（full specialization）时，可能需要实现所有成员函数。这是因为此时相当于提供了一个单独的类，而客户端代码很可能期望看到完整的接口实现。

标准库为用于容纳 bool 的 vector 提供了一个显式特化。vector<bool>的目的是允许库的实现将二进制位包装到整数中，以节省空间。[②]

正如本章前面所述，主 vector 类模板是这样声明的：

```
template<class T, class Allocator = allocator<T> >
class vector;
```

要为 bool 类型的对象特化该模板，可以像下面这样声明一个显式特化：

```
template<> class vector<bool, allocator<bool> > { ... };
```

再次强调，当看到 template<>前缀，并且看到类名后的实参列表完全匹配了主模板的所有参数，就很容易识别出这一个“全特化”

但是，vector<bool>实际并不是这样特化的，它比我们描述的更灵活，具体将在下一节讲述。

5.3.2 偏特化

类模板还可以进行**偏特化**（或者称为**部分特化**），这意味着至少有一个模板参数在某种程度上是开放（未具体指定）的。vector<bool>虽然指定了对象类型（bool），但未指定分配器类型。以下是实际的 vector<bool>声明：

```
template<class Allocator> class vector<bool, Allocator>;
```

[①] 译注：整个 const char* const&代表一个指向常量字符串的常量引用。第一个 const 表示指针指向的内容是常量（即字符串内容不可修改）；第二个 const 表示指针本身是常量（指针的值不可修改）。

[②] 第 7 章将深入探讨 vector<bool>。

偏特化很容易识别，因为在 template 关键字后面以及类名后面的尖括号中，出现了非空的参数列表。前者包含的是未指定的参数，后者包含的则是指定的实参。像这样定义 vector<bool>，用户就可以提供自定义的分配器类型，即使包含的类型 bool 是固定的。换言之，特化，特别是偏特化，构成了类模板的一种“重载”。

类模板的部分排序(偏序)

至于具体选择哪个模板进行实例化，其规则类似于函数模板的部分排序——选择“最特化”的那个。以下每个 f()成员函数输出的字符串解释了每个模板定义的作用。

```
//: C05:PartialOrder2.cpp
// 演示类模板的部分排序（偏序）
#include <iostream>
using namespace std;

template<class T, class U> class C {
public:
    void f() { cout << "主模板\n"; }
};

template<class U> class C<int, U> {
public:
    void f() { cout << "T == int\n"; }
};

template<class T> class C<T, double> {
public:
    void f() { cout << "U == double\n"; }
};

template<class T, class U> class C<T*, U> {
public:
    void f() { cout << "使用了 T*\n"; }
};

template<class T, class U> class C<T, U*> {
public:
    void f() { cout << "使用了 U*\n"; }
};

template<class T, class U> class C<T*, U*> {
public:
    void f() { cout << "使用了 T*和 U*\n"; }
};

template<class T> class C<T, T> {
public:
```

```
    void f() { cout << "T == U\n"; }
};

int main() {
    C<float, int>().f();      // 1: 主模板
    C<int, float>().f();      // 2: T == int
    C<float, double>().f();   // 3: U == double
    C<float, float>().f();    // 4: T == U
    C<float*, float>().f();   // 5: 使用了 T* [T 是 float]
    C<float, float*>().f();   // 6: 使用了 U* [U 是 float]
    C<float*, int*>().f();    // 7: 使用了 T*和 U* [float, int]

    // 以下情况存在歧义：
    // 8: C<int, int>().f();
    // 9: C<double, double>().f();
    // 10: C<float*, float*>().f();
    // 11: C<int, int*>().f();
    // 12: C<int*, int*>().f();
} ///:~
```

如你所见，可以根据模板参数是否为指针类型或者它们是否相等来部分指定这些参数。使用 T*特化时，例如在第 5 行（行号参见注释），T 本身并不是传递的顶级指针类型——它是该指针所引用的类型（本例即为 float）[①]。T*更像是一种模式匹配，它告诉编译器：当模板参数是一个指针类型时，就使用这个特化版本。如果使用 int**作为第一个模板实参，T 就变成 int*。第 8 行存在歧义，因为将第一个参数设为 int 与两个参数相等是两个独立的问题——一个并不比另一个更特化[②]。类似的逻辑也适用于第 9 行到第 12 行。

5.3.3 一个实例

我们可以轻松地从类模板派生，并且可以创建一个新的模板，该模板实例化并继承自现有模板。例如，如果 vector 模板满足了自己的大部分需求，但在某个应用程序中希望有一个能“自排序”的版本，那么可以轻松地重用 vector 的代码。下例从 vector<T>派生并添加了排序功能。注意，由于 vector 没有虚析构函数，所以从它派生时，如果需要在析构函数中执行资源清理工作，那么会是相当危险的。

```
//: C05:Sortable.h
// 模板特化
#ifndef SORTABLE_H
#define SORTABLE_H
#include <cstring>
#include <cstddef>
```

[①] 译注：使用 T*作为模板参数时，T 代表的是指针所指向的类型，而不是指针本身。

[②] 译注：换言之，这一行既可以匹配"T == int"的特化，也可以匹配"T == U"的特化。

```
#include <string>
#include <vector>
using std::size_t;

template<class T>
class Sortable : public std::vector<T> {
public:
    void sort();
};

template<class T>
void Sortable<T>::sort() { // 简单排序
    for(size_t i = this->size(); i > 0; --i)
        for(size_t j = 1; j < i; ++j)
            if(this->at(j-1) > this->at(j)) {
                T t = this->at(j-1);
                this->at(j-1) = this->at(j);
                this->at(j) = t;
            }
}

// 指针偏（部分）特化：
template<class T>
class Sortable<T*> : public std::vector<T*> {
public:
    void sort();
};

template<class T> void Sortable<T*>::sort() {
    for(size_t i = this->size(); i > 0; --i)
        for(size_t j = 1; j < i; ++j)
            if(*this->at(j-1) > *this->at(j)) {
                T* t = this->at(j-1);
                this->at(j-1) = this->at(j);
                this->at(j) = t;
            }
}

// char*全特化
//（这里为了方便将其做成内联函数——正常情况下应该将函数体
//  放在单独的文件中，这里只写声明。）
template<> inline void Sortable<char*>::sort() {
    for(size_t i = this->size(); i > 0; --i)
        for(size_t j = 1; j < i; ++j)
            if(std::strcmp(this->at(j-1), this->at(j)) > 0) {
                char* t = this->at(j-1);
                this->at(j-1) = this->at(j);
                this->at(j) = t;
            }
}
```

```
#endif // SORTABLE_H ///:~
```

除了一个特例（char*）之外，Sortable 模板对其他所有用来实例化的类都施加了一个限制，即必须包含（支持）一个>操作符。该模板仅适用于非指针对象（包括内置类型的对象）。在全特化版本中，模板使用 strcmp()来比较元素，以便根据空终止字符串对由 char*构成的 vector 进行排序。在上述代码中，“this->”是必需的[①]，并在本章后面的 5.4 节“名称查找问题”一节中进行了详细解释。[②]

下面是 Sortable.h 的一个驱动程序[③]，它使用了本章之前介绍的随机数生成器。

```
//: C05:Sortable.cpp
//{-bor} (因为 Urand.h 中使用了 bitset)
// 测试模板规范
#include <cstddef>
#include <iostream>
#include "Sortable.h"
#include "Urand.h"
using namespace std;

// 计算数组 a 的元素个数
// asz 是“数组大小”（array size）的意思
#define asz(a) (sizeof a / sizeof a[0])

char* words[] = { "is", "running", "big", "dog", "a", };
char* words2[] = { "this", "that", "theother", };

int main() {
    Sortable<int> is;
    Urand<47> rnd;
    for (size_t i = 0; i < 15; ++i)
        is.push_back(rnd());
    for (size_t i = 0; i < is.size(); ++i)
        cout << is[i] << ' ';
    cout << endl;

    is.sort();
    for (size_t i = 0; i < is.size(); ++i)
        cout << is[i] << ' ';
```

---

① 也可以不使用 this->，而是改为使用其他有效的限定方式，例如 Sortable::at()或 vector<T>::at()。这里的重点在于“限定”。

② 同时参见第 7 章对示例程序 PriorityQueue6.cpp 的解释。

③ 译注：也可以说成“测试程序”。之所以称为驱动程序（driver），是因为它包含了程序入口点，可以“驱动”整个程序的运行。

```
    cout << endl;

    // 使用模板偏特化（部分特化）：
    Sortable<string*> ss;
    for (size_t i = 0; i < asz(words); ++i)
        ss.push_back(new string(words[i]));
    for (size_t i = 0; i < ss.size(); ++i)
        cout << *ss[i] << ' ';
    cout << endl;
    ss.sort();
    for (size_t i = 0; i < ss.size(); ++i) {
        cout << *ss[i] << ' ';
        delete ss[i];
    }
    cout << endl;

    // 使用针对 char*的全特化：
    Sortable<char*> scp;
    for (size_t i = 0; i < asz(words2); ++i)
        scp.push_back(words2[i]);
    for (size_t i = 0; i < scp.size(); ++i)
        cout << scp[i] << ' ';
    cout << endl;
    scp.sort();
    for (size_t i = 0; i < scp.size(); ++i)
        cout << scp[i] << ' ';
    cout << endl;
} ///:~
```

在上述代码中，每个模板实例化都使用了模板类的一个不同的版本。其中，Sortable<int>使用了主模板（primary template）。Sortable<string*>使用了针对指针的“偏特化”。最后，Sortable<char*>使用了针对 char*的“全特化”。如果没有这个全特化版本，您可能会误以为一切工作正常，因为 words 数组仍会按字母顺序排列成“a big dog is running”，因为偏特化会最终比较每个数组的第一个字符。然而，words2 的排序就不正确了。

5.3.4 防止模板代码膨胀

每当一个类模板被实例化时，为会该“特化”生成类定义代码，还会为程序中调用的所有成员函数生成代码。注意，只有实际调用的成员函数才会生成代码。这是一个很好的设计，如以下程序所示：

```
//: C05:DelayedInstantiation.cpp
// 类模板的成员函数在需要时才实例化
class X {
public:
    void f() {}
};
```

```
class Y {
public:
    void g() {}
};

template<typename T> class Z {
    T t;
public:
    void a() { t.f(); }
    void b() { t.g(); }
};

int main() {
    Z<X> zx;
    zx.a(); // 不会创建 Z<X>::b()
    Z<Y> zy;
    zy.b(); // 不会创建 Z<Y>::a()
} ///:~
```

在本例中，虽然在模板 Z 的声明中同时使用了 T 的 f()和 g()成员函数，但既然该程序能成功通过编译，所以必然只有在 zx 上显式调用时才会生成 Z<X>::a()。（如果同时生成 Z<X>::b()，会因为尝试调用不存在的 X::g()而报告编译错误。）类似地，调用 zy.b()不会生成 Z<Y>::a()。因此，模板 Z 可以同时用于 X 和 Y。相反，如果在类首次实例化时便生成所有成员函数，那么许多模板的使用都会受到显著限制。

假设现在有一个模板化的 Stack（栈）容器，并且使用了针对 int、int*和 char*的特化。程序中将会生成并链接这三个版本的 Stack。我们之所以使用模板，一个主要的是动机是避免手动复制代码。但是，代码仍会被复制——只是由编译器而不是我们自己完成这项工作。可以使用“全特化”和“偏特化”的一个组合，将指针类型的存储逻辑集中到一个单独的类中。这里的关键是针对 void*进行全特化。然后，从 void*的实现中派生出其他所有指针类型，以便共享那些通用的代码。以下程序演示了这一技术：

```
//: C05:Nobloat.h
// 共享在栈中存储指针的逻辑
#ifndef NOBLOAT_H
#define NOBLOAT_H
#include <cassert>
#include <cstddef>
#include <cstring>

// 主模板
template<class T> class Stack {
    T* data;
    std::size_t count;
    std::size_t capacity;
    enum { INIT = 5 };
public:
```

```
    Stack() {
        count = 0;
        capacity = INIT;
        data = new T[INIT];
    }

    void push(const T& t) {
        if (count == capacity) { // 扩展数组
            std::size_t newCapacity = 2 * capacity;
            T* newData = new T[newCapacity];
            for (size_t i = 0; i < count; ++i)
                newData[i] = data[i];
            delete[] data;
            data = newData;
            capacity = newCapacity;
        }
        assert(count < capacity);
        data[count++] = t;
    }

    void pop() {
        assert(count > 0);
        --count;
    }

    T top() const {
        assert(count > 0);
        return data[count - 1];
    }

    std::size_t size() const { return count; }
};

// 针对 void*的全特化
template<> class Stack<void*> {
    void** data;
    std::size_t count;
    std::size_t capacity;
    enum { INIT = 5 };
public:
    Stack() {
        count = 0;
        capacity = INIT;
        data = new void*[INIT];
    }

    void push(void* const& t) {
        if (count == capacity) {
            std::size_t newCapacity = 2 * capacity;
            void** newData = new void*[newCapacity];
```

```
            std::memcpy(newData, data, count * sizeof(void*));
            delete[] data;
            data = newData;
            capacity = newCapacity;
        }
        assert(count < capacity);
        data[count++] = t;
    }

    void pop() {
        assert(count > 0);
        --count;
    }

    void* top() const {
        assert(count > 0);
        return data[count - 1];
    }

    std::size_t size() const {
        return count;
    }
};

// 针对其他指针类型的偏特化
template<class T> class Stack<T*> : private Stack<void*> {
    typedef Stack<void*> Base;
public:
    void push(T* const& t) { Base::push(t); }
    void pop() { Base::pop(); }
    T* top() const { return static_cast<T*>(Base::top()); }
    std::size_t size() { return Base::size(); }
};
#endif // NOBLOAT_H ///:~
```

这个简单的栈会在达到容量上限时自动扩展（扩容）。针对void*指针的模板特化之所以被称为“全特化”，是因为它没有指定任何模板参数（这通过 template<>来表示，即模板参数列表为空）。如前所述，在类模板的全特化中，所有成员函数都有必要实现。因此，在这个示例中，void*的特化提供了指针存储的完整实现。但是，在针对其他所有指针类型的特化中，我们可以节省大量代码。对于其他指针类型，我们只需提供“偏特化”，将其私有继承自Stack<void*>。这是因为我们仅仅是出于实现的目的而使用Stack<void*>，并不希望直接向用户公开它的任何接口。[1]在这些特化中，每个成员函数都非常简单，它们

[1] 译注：避免用户直接操作void*指针，减少潜在的类型错误和内存泄漏。用户只需要关心自己使用的指针类型，而不需要了解底层的实现细节。

只是将函数调用转发给基类 Stack<void*>中的对应函数。因此，当我们实例化一个非void*指针类型的栈时，编译器会为它生成一个非常小的类，因为大部分代码都继承自基类。[1]下面展示了一个驱动（测试）程序：

```
//: C05:NobloatTest.cpp
#include <iostream>
#include <string>
#include "Nobloat.h"
using namespace std;

template<class StackType> void emptyTheStack(StackType& stk) {
    while (stk.size() > 0) {
        cout << stk.top() << endl;
        stk.pop();
    }
}

// emptyTheStack 模板函数的一个重载（不是特化！）
template<class T> void emptyTheStack(Stack<T*>& stk) {
    while (stk.size() > 0) {
        cout << *stk.top() << endl;
        stk.pop();
    }
}

int main() {
    Stack<int> s1;
    s1.push(1);
    s1.push(2);
    emptyTheStack(s1);

    Stack<int*> s2;
    int i = 3;
    int j = 4;
    s2.push(&i);
    s2.push(&j);
    emptyTheStack(s2);
} ///:~
```

[1] 译注：由于负责“转发”的函数是内联（inline）的，因此编译器在生成最终的可执行文件时，并不会为 Stack<void*>类单独生成代码。注意，类内定义的成员函数会被隐式地视为内联函数。

为了方便起见，我们提供了两个 emptyTheStack 函数模板。[①]由于函数模板不支持偏特化，因此这里提供了重载模板。第二个版本的 emptyTheStack 比第一个更“特化”。因此，在使用指针类型时会选择它。程序中实例化了三个类模板：Stack<int>、Stack<void*>和Stack<int*>。Stack<void*>是隐式实例化的，因为 Stack<int*>派生于它。当程序需要实例化多种指针类型的栈时，相比仅仅使用单一的 Stack 模板，这种方式可以显著减少编译器生成的代码量。[②]

5.4 名称查找问题

当编译器遇到一个标识符时，它必须确定该标识符所代表的实体的类型与作用域（如果是变量，还包括生命周期）。模板增加了这种情况的复杂性。因为编译器在第一次看到模板定义时，并不能了解关于模板的所有信息。因此，除非“看到”了模板的实例化，否则它无法判断模板是否被正确使用。这种情况导致了模板编译的两阶段过程。

5.4.1 模板中的名称

第一阶段，编译器会解析模板定义，寻找明显的语法错误，并尽可能解析所有名称。它可以使用正常的名称查找机制来解析那些不依赖于模板参数的名称。如有必要，它还可以通过依赖于实参的查找（稍后解释）来解析名称。不能解析的名称就是所谓的**依赖名称**（dependent name），因为这些名称以某种方式依赖于模板参数。这些名称必须等到模板用实参来实例化时才能正确解析。因此，实例化就是模板编译的第二阶段。在这个阶段，编译器决定是否应该使用模板的一个显式特化，而不是使用主模板。

在接触实际的例子之前，需要先理解另外两个术语。第一个是**限定名称**（qualified name），它是带有类名前缀的名称、以对象名称和点操作符为前缀的名称或者以对象指针和箭头操作符为前缀的名称。下面展示了一些示例限定名称：

```
MyClass::f();
x.f();
p->f();
```

[①] 译注：第一个 emptyTheStack 函数模板接受 StackType 类型的栈作为参数（StackType 可以是任意栈类型），并直接输出栈顶元素。第二个函数模板则专门针对 Stack<T*>类型的栈。这种栈中包含的是指针。因此，它进行了重载，输出指针指向的值。之所以设计这样的一个重载，是为了更好地处理指针类型的数据，特别是当希望打印指针所指向的数据而非指针本身时。

[②] 译注：当对代码大小和性能有较高要求时，使用多个指针类型实例化是一个不错的选择。但要注意的是，现代 C++编译器的优化能力非常强大，它们可以自动进行许多优化。因此在实际应用中，代码大小的差异可能并不像理论上那么显著。

本书在多个地方使用了限定名称，最近在与 typename 关键字相关的上下文中也提到了它们。这些之所以被称为限定名称，因为目标名称（例如上述代码中的 f）显式地关联了一个类或命名空间，从而告诉编译器应该去哪里查找这些名称的**声明**。

另一个术语是**依赖于实参的查找**（Argument-Dependent Lookup，简称 ADL）[1]，它最初的设计宗旨是简化命名空间中声明的非成员函数调用（包括操作符函数），如下例所示：

```
#include <iostream>
#include <string>
// ...
std::string s("hello");
std::cout << s << std::endl;
```

注意，遵循头文件中通常的做法，本例没有使用 using namespace std;指令。没有这样的指令，就必须使用 std::限定符来引用 std 命名空间中的项。但是，并不是 std 中的所有项都进行了限定。您能看出哪些项没有限定吗？

本例没有限定的是要使用哪个操作符函数。我们希望存在以下函数声明，但又不想真的写出来：

```
std::operator<<(std::operator<<(std::cout, s), std::endl);
```

为了让原始输出语句按预期工作，ADL 规定，但凡出现未限定的函数调用，并且其声明不在（正常的）作用域中，就搜索其各个实参所在的命名空间，以查找匹配的函数声明。原始语句中的第一个函数调用是：

```
operator<<(std::cout, s);
```

由于原始的作用域内不存在这样的一个函数，因此编译器注意到该函数的第一个实参 std::cout 位于 std 命名空间；因此将该命名空间添加到搜索列表中，以查找与 operator<<(std::ostream&, std::string)这个签名最匹配的唯一函数。通过<string>头文件，它发现这个函数是在 std 命名空间中声明的。

没有 ADL 的帮助，命名空间会非常不便。注意，ADL 通常会从所有符合条件的命名空间中引入有疑问的名称的所有声明。因此，如果没有找到唯一的最佳匹配，将会导致歧义。

要关闭 ADL，可以将函数名放到一对圆括号中：

```
(f)(x, y); // ADL 被抑制
```

[1] 也称为 **Koenig 查找**，以首次向 C++标准委员会提议该技术的 Andrew Koenig 的名字命名。ADL 适用于所有情况，无论是否涉及模板。

现在考虑以下程序：[①]

```
//: C05:Lookup.cpp
// 只有使用某些特定编译器（例如，EDG 和 Metrowerks，并且后者要使用一个特殊选项），
// 程序才会表现出“预期”的行为。
#include <iostream>
using std::cout;
using std::endl;

// 传入 double 值时打印"f(double)"
void f(double) { cout << "f(double)" << endl; }

// 该模板类包含一个成员函数 g。g 函数调用了 f 函数，并传递值 1
// 值 1 是 double 还是 int？这是一个问题！
template<class T> class X {
public:
    void g() { f(1); }
};

// 传入 int 值时打印"f(int)"
void f(int) { cout << "f(int)" << endl; }

// main 创建一个 X<int>类型的对象，并立即调用它的成员函数 g
// 如果编译器认为 1 是 int，那么应该调用 f(int)函数，即输出"f(int)"
// 如果编译器认为 1 是 double，那么应该调用 f(double)函数，即输出"f(double)"
int main() {
    X<int>().g();
} ///:~
```

目前唯一能产生正确行为的编译器是 Edison Design Group 的前端编译器。另外，一些编译器，例如 Metrowerks，可以设置一个选项来启用正确的查找行为。[②]输出应该是：

```
f(double)
```

这是因为 f 是一个非依赖名称，可以在定义模板的当前上下文中提前解析。此时，只有 f(double)在作用域内。遗憾的是，行业中还有大量代码依赖于非标准的行为，也就是将 g()内部的 f(1)调用绑定到后面的 f(int)，因此编译器的开发者一直不愿意做出更改。

[①] 来自 Herb Sutter 的一次演讲。

[②] 译注：显然，作者认为正确的输出是 f(double)。虽然对这一点持保留态度，但译者还是分别使用 MSVC 和 GCC 进行了测试。前者的编译命令是 cl /EHsc Lookup.cpp，后者的编译命令是 g++ -o lookup Lookup.cpp。注意，均未添加额外的参数来指导“查找行为”。结果也很有趣，MSVC 编译器的输出结果是 f(int)，GCC 编译器的输出是 f(double)。

这里还有一个更详细的例子：[1][2]

```
//: C05:Lookup2.cpp {-bor}{-g++}{-dmc}
// MSVC 编译器请使用选项-Za（ANSI 模式）
#include <algorithm>
#include <iostream>
#include <typeinfo>
using std::cout;
using std::endl;

void g() { cout << "全局 g()" << endl; }

template<class T> class Y {
public:
    void g() {
        cout << "Y<" << typeid(T).name() << ">::g()" << endl;
    }

    void h() {
        cout << "Y<" << typeid(T).name() << ">::h()" << endl;
    }
    typedef int E;
};

typedef double E;

template<class T> void swap(T& t1, T& t2) {
    cout << "全局 swap" << endl;
    T temp = t1;
    t1 = t2;
    t2 = temp;
}

template<class T> class X : public Y<T> {
public:
    E f() {
        g();
        this->h();
        T t1 = T(), t2 = T(1);
        cout << t1 << endl;
        swap(t1, t2);
```

[1] 灵感也来自 Herb Sutter。

[2] 译注：编译命令是 `cl /EHsc /Za Lookup2.cpp`。注意，`/Za` 开关的作用是将编译器置于 ANSI 模式。此时会禁用非标准扩展，严格遵循 C++标准，确保代码的可移植性。读者可以试试去掉该开关后的输出结果。

```
        std::swap(t1, t2);
        cout << typeid(E).name() << endl;
        return E(t2);
    }
};

int main() {
    X<int> x;
    cout << x.f() << endl;
} ///:~
```

程序的输出应该是：

```
全局 g()
Y<int>::h()
0
全局 swap
double
1
```

下面重点分析一下 X::f()函数：

- E，这是X::f()的返回类型，它不是一个依赖名称[①]。因此，编译器在解析模板时就能确定 E 的类型，最终找到了全局作用域中将 E 定义为 double 的 typedef（即 typedef double E;）。这看起来有些违反直觉，因为对于非模板类来说，通常会优先在基类中查找成员，首先找到的是基类中的E声明（即typedef int E;）。但对于模板类来说，规则有所不同。（基类 Y 是一个依赖基类，即 dependent base class，因此在模板定义期间搜索不到它。）
- g()这个调用也是非依赖的，因为它没有在任何地方提到 T。如果 g 有在其他命名空间中定义的类类型的参数，那么 ADL 将会介入，因为当前作用域中没有带参数的 g。无论如何，就当前的情况来说，该调用匹配的是全局 g()声明。
- this->h()这个调用是一个限定名称，限定它的对象（this）引用了当前对象，即类型为 X 的对象，后者通过继承依赖于名称 Y<T>。X 内部没有 h()函数，因此会去 X 的基类 Y<T>的作用域中查找 h()函数。由于 Y<T>是一个依赖名称，其具体类型在模板实例化时才能确定，因此当模板实例化为 X<int>时，编译器会查找 Y<int>类中的 h()函数，并调用 Y<int>::h()。
- t1 和 t2 的声明是依赖的。
- operator<<(cout, t1)这个调用是依赖的，因为 t1 的类型是模板参数 T。只有在模板实例化时，T 的具体类型才能确定。当 T 被特化为 int 类型时，编译器会去查找 int

[①] 译注：前面讲过，所谓“依赖名称”，就是要依赖于模板参数的名称。这些名称必须等到模板用实参来实例化时才能正确解析。

类型的插入符（函数）。最终，它在 std 中找到了用于插入 int 类型数据的 operator<<函数。

- 对 swap()的无限定调用是依赖的，因其实参的类型为 T。这最终导致对全局 swap(int&, int&)的实例化。
- 对 std::swap()的限定调用是非依赖的，因为 std 是一个固定的命名空间，编译器知道在 std 中查找适当的声明。（只有在::左侧的限定符包含了一个模板参数的前提下，才认为被限定的名称是依赖的。）std::swap()函数模板稍后会实例化为 std::swap(int&, int&)。在 X<T>::f()中，没有更多的依赖名称了。

最后澄清并总结一下：如果名称是依赖的（依赖于模板参数），那么它的查找会推迟到模板实例化时进行。但是，对于无限定的依赖名称，编译器会在模板定义阶段就尝试进行一次普通的名称查找。模板中所有的非依赖名称都会在模板定义阶段查找。（模板实例化时，当模板实参的类型确定后，ADL 机制可能再次发挥作用，查找在模板定义阶段未能确定的函数。[①]）

如果仔细研究了这个例子，并真正做到了理解，那么请准备好迎接下一节所带来的关于友元声明的又一个“惊喜”。

5.4.2 模板和友元

如果在类的内部声明一个友元函数，那么相当于允许一个非成员函数访问该类的非 public 成员。如果友元函数名称带有限定符（即一个命名空间或类名，相当于指定了友元的作用域），那么编译器会直接在相应的命名空间或类中寻找该函数。相反，如果友元函数的名称不带限定符，那么编译器必须对友元函数的定义位置做出一个假设，因为**所有标识符都必须有唯一的作用域**。通常的假设是，该函数将在包含授予了友元关系的那个类的最近的外围命名空间（而非类的作用域）中定义。通常，这就是全局作用域。以下非模板示例阐明了这个问题：

```
//: C05:FriendScope.cpp
#include <iostream>
using namespace std;

class Friendly {
    int i;
public:
    Friendly(int theInt) { i = theInt; }
```

[①] 译注：例如，在嵌套的命名空间或类中，可能存在同名的实体。在模板定义阶段，编译器可能无法确定应该使用哪个实体，而在实例化阶段，根据模板实参的类型，可以确定应该使用哪个实体。另外，如果在模板中使用了依赖于模板参数的类型别名或 using 声明，那么在实例化时，这些别名或 using 声明会展开，从而引入新的名称。因篇幅有限，这里不再举例说明了。

```
    friend void f(const Friendly&); // 需要全局定义
    void g() { f(*this); }
};

void h() { f(Friendly(1)); } // ADL 机制介入

void f(const Friendly& fo) { // 友元定义
    cout << fo.i << endl;
}

int main() {
    h(); // 输出 1
    Friendly(2).g(); // 输出 2
} ///:~
```

在 Friendly 类内部，f()函数的声明没有限定符。因此，编译器期望最终能将该声明“链接”到文件作用域（本例就是包含 Friendly 的命名空间作用域）中的一个定义。该定义出现在 h()函数定义之后。而 h()内部对 f()的调用与同一函数的“链接”是两码事。这通过 ADL 来解决。由于 h()中 f()的实参是一个 Friendly 对象，因此会搜索 Friendly 类中的 f()声明，这成功了。相反，如果调用 f(1)（由于 1 可以隐式转换为 Friendly(1)，因此这样做无可厚非），那么调用就会失败，因为编译器不知道应该去哪里查找 f()的声明。编译器在这种情况下应该正确地报告 f 未定义。[①]

现在假设 Friendly 和 f 都是模板，如以下程序所示：

```
//: C05:FriendScope2.cpp
#include <iostream>
using namespace std;

// 必要的前置（前向）声明:
template<class T> class Friendly;
template<class T> void f(const Friendly<T>&);

template<class T> class Friendly {
    T t;
public:
    Friendly(const T& theT) : t(theT) {}
    friend void f<>(const Friendly<T>&);
    void g() { f(*this); }
};

void h() {
    f(Friendly<int>(1));
}
```

[①] 译注：有趣的是，GCC 编译器确实会报错，但 MSVC 不会。

```
template<class T> void f(const Friendly<T>& fo) {
    cout << fo.t << endl;
}

int main() {
    h();
    Friendly<int>(2).g();
} ///:~
```

首先注意 Friendly 内部 f 声明中的尖括号。为了告诉编译器 f 是模板，这对尖括号是必需的。否则，编译器会寻找一个名为 f 的普通函数（不存在这样的函数）。虽然可以将模板参数插入尖括号中（<T>），但这根据声明很容易就能推断出来。

类定义之前对函数模板 f 的前置声明是必要的，尽管在 f 不是模板的前一个例子中不需要；但语言规定了友元函数模板必须预先声明。而为了正确声明 f，Friendly 也必须声明，因为 f 要接受一个 Friendly 参数。因此，程序开头处了也对 Friendly 进行了前置声明。虽然可以在 Friendly 最初的声明之后放置 f 的完整定义，而不是把它的定义和声明分开，但我们选择保留更接近前一个例子的形式。

为了在模板内使用友元，最后一个选项是在宿主类模板定义本身中完整地定义它。下例对上一个例子进行了修改：

```
//: C05:FriendScope3.cpp
#include <iostream>
using namespace std;

template<class T> class Friendly {
    T t;
public:
    Friendly(const T& theT) : t(theT) {}
    friend void f(const Friendly<T>& fo) {
        cout << fo.t << endl;
    }
    void g() { f(*this); }
};

void h() {
    f(Friendly<int>(1));
}

int main() {
    h();
    Friendly<int>(2).g();
} ///:~
```

本例和上一个例子的重要区别在于：f 不是模板，而是普通的函数。每次 Friendly 类模板

实例化时，都会创建一个新的接受当前 Friendly 特化作为实参的普通函数重载。这就是 Dan Saks 所说的“交新朋友”。[1]这是为模板定义友元函数的最简便的方法。

可以再澄清一下，假设要向类模板添加非成员友元操作符。以下类模板的作用很简单，就是容纳泛型值：

```
template<class T> class Box {
    T t; // 容纳了一个 T 类型的值
public:
    Box(const T& theT) : t(theT) {} // 构造函数，初始化容纳的值
};
```

如果不理解本节前面的例子，新手会感到沮丧，因为他们搞不定简单的流输出插入符（<<）。如果不在 Box 的定义中定义操作符，那么必须提供之前展示的前置声明：

```
//: C05:Box1.cpp
// 定义模板操作符
#include <iostream>
using namespace std;

// 前置声明
template<class T> class Box;
template<class T> Box<T> operator+(const Box<T>&, const Box<T>&);
template<class T> ostream& operator<<(ostream&, const Box<T>&);

template<class T>
class Box {
    T t;
public:
    Box(const T& theT) : t(theT) {}
    friend Box operator+<>(const Box<T>&, const Box<T>&);
    friend ostream& operator<< <>(ostream&, const Box<T>&);
};

template<class T>
Box<T> operator+(const Box<T>& b1, const Box<T>& b2) {
    return Box<T>(b1.t + b2.t);
}

template<class T>
ostream& operator<<(ostream& os, const Box<T>& b) {
    return os << '[' << b.t << ']';
}

int main() {
```

[1] 在 2001 年 9 月俄勒冈州波特兰市举行的 C++研讨会上的一次演讲中。

```
    Box<int> b1(1), b2(2);
    cout << b1 + b2 << endl;   // [3]
    // cout << b1 + 2 << endl; // 没有隐式转换!
} ///:~
```

本例定义了一个加法操作符和一个输出流操作符。main 程序揭示了这种方法的一个缺点：我们依赖不了隐式转换（表达式 b1 + 2），因为模板没有提供。在这种情况下，在类内使用非模板函数，显得更加简洁和健壮：

```
//: C05:Box2.cpp
// 定义非模板操作符
#include <iostream>
using namespace std;

template<class T> class Box {
    T t;
public:
    Box(const T& theT) : t(theT) {}
    friend Box<T> operator+(const Box<T>& b1, const Box<T>& b2) {
        return Box<T>(b1.t + b2.t);
    }
    friend ostream& operator<<(ostream& os, const Box<T>& b) {
        return os << '[' << b.t << ']';
    }
};

int main() {
    Box<int> b1(1), b2(2);
    cout << b1 + b2 << endl; // [3]
    cout << b1 + 2 << endl;  // [3]
} ///:~
```

由于操作符是普通函数（会为 Box 的每个特化而重载——本例只有一个 int 特化版本），因此会正常应用隐式转换，表达式 b1 + 2 是有效的。

注意，有一种类型不能成为 Box 或其他任何类模板的友元，那就是 T——或者说，对类模板进行参数化的那个类型。据我们所知，真的没有好的理由不允许这样做，但目前来看，声明 friend class T 是非法的，编译时会报错。[①]

友元模板

可以精确指定一个模板的哪些特化是某个类的友元。在之前的例子中，对于 f 函数模板的特化，只有当特化它的类型与特化 Friendly 的类型是一样的时候，f 函数的这个特化才会

[①] 译注：截止最新的 C++标准，friend class T 仍然是非法的。

成为友元。例如，只有 f<int>(const Friendly<int>&)才是 Friendly<int>类的友元。为此，需要使用 Friendly 的模板参数在友元声明中特化 f。但是，如果愿意，也可以让 f 的一个固定的特化成为所有 Friendly 实例的友元，如下所示：

```
// 在 Friendly 类内部声明：
friend void f<>(const Friendly<double>&);
```

通过使用 double 而不是 T，double 特化版的 f 可以访问 Friendly 类的任何特化中的非 public 成员。注意，除非显式调用，否则 f<double>()不会自动实例化，这和之前是一样的。

类似地，如果声明一个非模板函数，它没有任何参数依赖于 T，那么该函数就是 Friendly 的所有实例的友元：

```
// 在 Friendly 类内部声明：
friend void g(int); // g(int)成为所有 Friendly 的友元
```

由于 g(int)没有进行限定，因此它必须在文件作用域（包含 Friendly 的那个命名空间作用域）中定义。

也可以安排 f 的所有特化成为所有 Friendly 特化的友元，这是通过所谓的**友元模板**（friend template）来实现的，如下所示：

```
template<class T> class Friendly {
    template<class U> friend void f<>(const Friendly<U>&);
}
```

由于友元声明中的模板实参独立于 T，因此 T 和 U 的任何组合都是允许的，从而实现了“交朋友”的目标。类似于成员模板，非模板类中也可以使用友元模板。

5.5 模板编程惯用法

既然语言是思维的工具，所以新的语言特性往往会孵化出新的编程技术。本节将介绍自模板引入 C++语言以来，这些年来间发展起来的一些常见的模板编程惯用法（idioms）。[①]

[①] 第 9 章详细介绍将另一种模板惯用法：mixin 继承。（译注：mixin 继承是一种特殊的继承机制，它允许一个类从多个类中继承特性，但又不形成传统意义上的继承层次。可以想象它“混入”了其他类的特性，从而创建出一个新的类。注意，它和传统的“多重继承”是有区别的，不会形成传统意义上的继承层次结构。

5.5.1 traits

由 Nathan Myers 开创的 traits[①]模板技术是一种将依赖于类型的声明捆绑在一起的方法。从本质上说，利用 traits，我们可以在具体的使用上下文中，以一种灵活的方式对某些类型和值进行“混合并匹配”，同时保持代码的可读性和可维护性。

traits 模板最简单的例子是在<limits>中定义的 numeric_limits 类模板。主模板定义如下所示：

```
template<class T> class numeric_limits {
public:
    static const bool is_specialized = false;
    static T min() throw();
    static T max() throw();
    static const int digits = 0;
    static const int digits10 = 0;
    static const bool is_signed = false;
    static const bool is_integer = false;
    static const bool is_exact = false;
    static const int radix = 0;
    static T epsilon() throw();
    static T round_error() throw();
    static const int min_exponent = 0;
    static const int min_exponent10 = 0;
    static const int max_exponent = 0;
    static const int max_exponent10 = 0;
    static const bool has_infinity = false;
    static const bool has_quiet_NaN = false;
    static const bool has_signaling_NaN = false;
    static const float_denorm_style has_denorm = denorm_absent;
    static const bool has_denorm_loss = false;
    static T infinity() throw();
    static T quiet_NaN() throw();
    static T signaling_NaN() throw();
    static T denorm_min() throw();
    static const bool is_iec559 = false;
    static const bool is_bounded = false;
    static const bool is_modulo = false;
    static const bool traps = false;
    static const bool tinyness_before = false;
    static const float_round_style round_style = round_toward_zero;
};
```

<limits>头文件为所有基本数值类型都定义了特化（当成员 is_specialized 设置为 true

[①] 译注：一些人将 traits 翻译为“特性”，本书选择保留原文。

的时候）。例如，为了在自己的浮点数系统中获取 double 的基数，可以使用表达式 numeric_limits<double>::radix 。[1] 要获取最小整数值，可以使用 numeric_limits<int>::min()。并不是说 numeric_limits 的所有成员都适用于所有基本类型。（例如，epsilon()仅对浮点类型有意义。请参见 4.7.1 节对于 epsilon 的解释）

那些始终为整型的值是 numeric_limits 的静态数据成员。那些可能不是整型的值，例如 float 的最小值，是作为静态内联成员函数实现的。这是因为 C++只允许在类定义内初始化**整型**静态数据成员常量。

第 3 章已经讲述了如何用 traits 来控制字符串类使用的字符处理功能。std::string 和 std::wstring 是 std::basic_string 模板的特化，后者的定义如下所示：

```
template<class charT,
    class traits = char_traits<charT>,
    class allocator = allocator<charT> >
class basic_string;
```

模板参数 charT 表示底层的字符类型，通常是 char 或者 wchar_t。主 char_traits 模板通常是空的，而且针对 char 和 wchar_t 的特化已由标准库提供。下面展示了 C++标准提供的 char_traits<char>特化：

```
template<> struct char_traits<char> {
    typedef char char_type;
    typedef int int_type;
    typedef streamoff off_type;
    typedef streampos pos_type;
    typedef mbstate_t state_type;

    static void assign(char_type& c1, const char_type& c2);
    static bool eq(const char_type& c1, const char_type& c2);
    static bool lt(const char_type& c1, const char_type& c2);
    static int compare(const char_type* s1, const char_type* s2, size_t n);
    static size_t length(const char_type* s);
    static const char_type* find(const char_type* s, size_t n, const char_type& a);
    static char_type* move(char_type* s1, const char_type* s2, size_t n);
    static char_type* copy(char_type* s1, const char_type* s2, size_t n);
    static char_type* assign(char_type* s, size_t n, char_type a);
    static int_type not_eof(const int_type& c);
    static char_type to_char_type(const int_type& c);
    static int_type to_int_type(const char_type& c);
    static bool eq_int_type(const int_type& c1, const int_type& c2);
    static int_type eof();
```

---

[1] 译注：在大多数系统中，double 类型的浮点数都是二进制表示的，因此该表达式返回 10。但在极少数系统中，double 值用十进制表示。

```
};
```

使用这些函数，basic_string 类模板执行字符串处理中常见的一些基于字符的操作。例如，当声明一个 string 变量时：

```
std::string s;
```

s 实际是这样声明的（注意 basic_string 所指定的默认模板实参）：

```
std::basic_string<char, std::char_traits<char>, std::allocator<char>> s;
```

由于字符 traits 已经从 basic_string 类模板中分离出来，因此可以提供一个自定义的 traits 类来替代 std::char_traits。下例演示了这种灵活性：

```
//: C05:BearCorner.h
#ifndef BEARCORNER_H
#define BEARCORNER_H
#include <iostream>
using std::ostream;

// 表示各种物品的类（也就是客人的 traits，可以理解成客人的“偏好”或“特征”）：
class Milk {
public:
    friend ostream& operator<<(ostream& os, const Milk&) { return os << "牛奶"; }
};

class CondensedMilk {
public:
    friend ostream& operator<<(ostream& os, const CondensedMilk &) {
        return os << "炼乳";
    }
};

class Honey {
public:
    friend ostream& operator<<(ostream& os, const Honey&) { return os << "蜂蜜"; }
};

class Cookies {
public:
    friend ostream& operator<<(ostream& os, const Cookies&) { return os << "饼干"; }
};

// 表示各种客人的类：
class Bear {
public:
    friend ostream& operator<<(ostream& os, const Bear&) { return os << "泰迪"; }
};

class Boy {
```

```
public:
    friend ostream& operator<<(ostream& os, const Boy&) { return os << "帕特里克"; }
};

// 主 traits 模板（现在是空的，可以容纳通用类型）
template<class Guest> class GuestTraits;

// 针对客人类型的 traits 特化
// 熊喜欢炼乳和蜂蜜
template<> class GuestTraits<Bear> {
public:
    typedef CondensedMilk beverage_type;
    typedef Honey snack_type;
};

// 男孩喜欢牛奶和饼干
template<> class GuestTraits<Boy> {
public:
    typedef Milk beverage_type;
    typedef Cookies snack_type;
};
#endif // BEARCORNER_H ///:~
```

以下程序演示了如何使用 traits 类：

```
//: C05:BearCorner.cpp
// 演示如何使用 traits 类
#include <iostream>
#include "BearCorner.h"
using namespace std;

// 一个自定义的 traits 类
class MixedUpTraits {
public:
    typedef Milk beverage_type;
    typedef Honey snack_type;
};

// Guest 模板（使用一个 traits 类）
template<class Guest, class traits = GuestTraits<Guest>> class BearCorner {
    Guest theGuest;
    typedef typename traits::beverage_type beverage_type;
    typedef typename traits::snack_type snack_type;
    beverage_type bev; // 喝的
    snack_type snack;  // 吃的
public:
    BearCorner(const Guest& g) : theGuest(g) {}
    void entertain() {
```

```
            cout << "向" << theGuest << "提供" << bev << "和" << snack << endl;
        }
    };

    int main() {
        Boy cr;
        BearCorner<Boy> pc1(cr);
        pc1.entertain();
        Bear pb;
        BearCorner<Bear> pc2(pb);
        pc2.entertain();
        BearCorner<Bear, MixedUpTraits> pc3(pb);
        pc3.entertain();
    } ///:~
```

在这个程序中，Boy 和 Bear 类的实例会得到适合他们口味的食物。男孩喜欢牛奶和饼干，而熊喜欢炼乳和蜂蜜。这种客人与物品之间的关联是通过特化一个主（空）traits 类模板来实现的。传递给 BearCorner 的默认实参确保了不同类型的客人得到他们最适合的东西，但也可以通过简单地提供一个满足 traits 类要求的类来覆盖这一默认设置，就像本例中的 MixedUpTraits 类那样。该程序的输出为：

```
向帕特里克提供牛奶和饼干
向泰迪提供炼乳和蜂蜜
向泰迪提供牛奶和蜂蜜
```

使用 traits 具有两方面的关键优势：（1）为对象与其相关属性或功能的配对提供了灵活性和扩展性；（2）保持了模板参数列表的小巧和可读性。想想看，假如一个客人关联了 30 种类型，那么在每个 BearCorner 声明中都直接指定所有 30 个实参显得极为不便。将这些类型纳入一个单独的 traits 类中，可以极大地简化编程。

traits 技术也被广泛用于实现流和区域设置（locale），具体可以参见第 4 章。第 6 章的示例头文件 PrintSequence.h 展示了迭代器 traits 的一个例子。

5.5.2 策略

仔细检查针对 wchar_t 的 char_traits 模板特化，会发现它实际上与 char 版本几乎完全相同：

```
template<> struct char_traits<wchar_t> {
    typedef wchar_t char_type;
    typedef wint_t int_type;
    typedef streamoff off_type;
    typedef wstreampos pos_type;
    typedef mbstate_t state_type;

    static void assign(char_type& c1, const char_type& c2);
    static bool eq(const char_type& c1, const char_type& c2);
```

```
    static bool lt(const char_type& c1, const char_type& c2);
    static int compare(const char_type* s1, const char_type* s2, size_t n);
    static size_t length(const char_type* s);
    static const char_type* find(const char_type* s,
                                 size_t n, const char_type& a);
    static char_type* move(char_type* s1, const char_type* s2, size_t n);
    static char_type* copy(char_type* s1, const char_type* s2, size_t n);
    static char_type* assign(char_type* s, size_t n, char_type a);
    static int_type not_eof(const int_type& c);
    static char_type to_char_type(const int_type& c);
    static int_type to_int_type(const char_type& c);
    static bool eq_int_type(const int_type& c1, const int_type& c2);
    static int_type eof();
};
```

在两个版本之间，唯一的真正区别在于涉及的类型集（char/int 与 wchar_t/wint_t）。但是，两个特化提供的功能性是一样的。这突显了traits（特性、特征）一词的含义。而在相关的各种 traits 类之间，发生变化的通常是指针类型和常量值，或者是使用了与类型相关的模板参数的固定算法。traits类往往本身就是模板，因其包含的类型和常量被视为主模板参数的特征（例如，char 和 wchar_t）。

另外，将**功能性**（functionality）与模板实参关联起来也很有用，这使客户端程序员能在编码时轻松地定制行为。例如，下面这个版本的 BearCorner 程序支持不同类型的娱乐活动：

```
//: C05:BearCorner2.cpp
// 演示策略类
#include <iostream>
#include "BearCorner.h"
using namespace std;

// 策略类（要求一个静态 doAction()函数）:
class Feed {
public:
    static const char* doAction() { return "喂"; }
};

class Stuff {
public:
    static const char* doAction() { return "填充"; }
};

// Guest 模板（使用了一个策略和一个 traits 类）
template<class Guest, class Action, class traits = GuestTraits<Guest> >
class BearCorner {
    Guest theGuest;
    typedef typename traits::beverage_type beverage_type;
    typedef typename traits::snack_type snack_type;
    beverage_type bev; // 喝的
    snack_type snack;  // 吃的
```

```
public:
    BearCorner(const Guest& g) : theGuest(g) {}
    void entertain() {
        cout << "用" << bev << "和" << snack
             << Action::doAction() << theGuest << endl;
    }
};

int main() {
    Boy cr;
    BearCorner<Boy, Feed> pc1(cr);
    pc1.entertain();
    Bear pb;
    BearCorner<Bear, Stuff> pc2(pb);
    pc2.entertain();
} ///:~
```

BearCorner 类的 Action 模板参数期望有一个名为 doAction()的静态成员函数，该函数在 BearCorner<>::entertain()中使用。用户可以根据需要选择 Feed（喂）或 Stuff（填充）这两种行动，两者都提供了相应的 doAction()函数。以这种方式封装功能性的类称为**策略类**（policy class）。上述娱乐“策略”是通过 Feed::doAction()和 Stuff::doAction()来提供的。这些策略类在本例中只是正好为普通类。但是，它们完全可以是模板，并且可与继承结合使用以获得更大的优势。要想更深入地了解“基于策略的设计”，请参见 Andrei Alexandrescu 的书[①]，这是该主题的权威参考书。

5.5.3 奇特的递归模板模式

任何 C++新手都很容易搞清楚如何修改一个类，以跟踪该类现有的对象数量。所要做的就是添加静态成员，并修改构造和析构函数逻辑，如下所示：

```
//: C05:CountedClass.cpp
// 利用静态成员进行对象数量计数
#include <iostream>
using namespace std;

class CountedClass {
    static int count;
public:
    CountedClass() { ++count; }
    CountedClass(const CountedClass&) { ++count; }
    ~CountedClass() { --count; }
    static int getCount() { return count; }
};
```

[①] *Modern C++ Design: Generic Programming and Design Patterns Applied*，Addison Wesley, 2001。

```
int CountedClass::count = 0;

int main() {
    CountedClass a;
    cout << CountedClass::getCount() << endl;     // 1
    CountedClass b;
    cout << CountedClass::getCount() << endl;     // 2
    { // 随便定义一个作用域：
        CountedClass c(b);
        cout << CountedClass::getCount() << endl; // 3
        a = c;
        cout << CountedClass::getCount() << endl; // 3
    }
    // 离开作用域，其中实例化的对象就自动销毁了
    cout << CountedClass::getCount() << endl;     // 2
} ///:~
```

每创建一个 CountedClass 实例，其构造函数都会递增静态数据成员 count，而析构函数则会递减它。静态成员函数 getCount()返回当前对象数量。

每次希望为一个类添加对象计数时，如果都手动添加这些成员，那么会显得十分繁琐。在“面向对象编程”理念中，为了重复或共享代码，我们一般会使用继承。但就目前这种情况来说，“继承”只能解决一半的问题。来看看将计数逻辑放到基类中之后会发生什么情况。

```
//: C05:CountedClass2.cpp
// 演示对象计数的一次错误尝试
#include <iostream>
using namespace std;

class Counted {
    static int count;
public:
    Counted() { ++count; }
    Counted(const Counted&) { ++count; }
    ~Counted() { --count; }
    static int getCount() { return count; }
};

int Counted::count = 0;

// 两个类都从 Counted 继承
class CountedClass : public Counted {};
class CountedClass2 : public Counted {};

int main() {
    CountedClass a;
    cout << CountedClass::getCount() << endl; // 1
```

```
    CountedClass b;
    cout << CountedClass::getCount() << endl; // 2
    CountedClass2 c;
    cout << CountedClass2::getCount() << endl; // 3（这个是错的）
} ///:~
```

从 Counted 继承的所有类都共享同一个静态数据成员。因此，整个 Counted 继承层次结构中的对象数量是作为一个整体来跟踪的。因此，我们需要一种方法为每个派生类自动生成一个**不同**的基类。这可以通过下面这个奇特的模板构造来实现：

```
//: C05:CountedClass3.cpp
#include <iostream>
using namespace std;

template<class T> class Counted {
    static int count;
public:
    Counted() { ++count; }
    Counted(const Counted<T>&) { ++count; }
    ~Counted() { --count; }
    static int getCount() { return count; }
};

template<class T> int Counted<T>::count = 0;

// 奇特的类定义
class CountedClass : public Counted<CountedClass> {};
class CountedClass2 : public Counted<CountedClass2> {};

int main() {
    CountedClass a;
    cout << CountedClass::getCount() << endl; // 1
    CountedClass b;
    cout << CountedClass::getCount() << endl; // 2
    CountedClass2 c;
    cout << CountedClass2::getCount() << endl; // 1 (这次对了)
} ///:~
```

每个派生类都从一个由自身（即派生类）作为模板参数来确定的唯一基类派生！这看起来有点像是循环定义。如果任何一个基类成员在计算中使用了模板实参的话，那么确实就是如此。由于 Counted 的任何数据成员都不依赖于 T，因此 Counted 的大小（零!）在模板被解析时就已经知道了。因此，随便用哪个实参来实例化 Counted 都没关系，因为大小总是

相同的。[1]从Counted实例派生的任何类都可以在其被解析时完成，不会发生递归。由于每个基类都是唯一的，因此它有自己的静态数据，可以方便地向任何类添加实例计数功能。Jim Coplien 是第一个在文献中提到这种有趣的派生方式的人，具体请参见他的“奇特重复模板模式”（Curiously Recurring Template Patterns，CRTP）一文。[2]

5.6 模板元编程

1993 年左右，编译器开始支持简单的模板构造，使得用户可以定义泛型容器和函数。差不多在同一时间，当 STL 正在考虑纳入 C++标准期间，像下面这样巧妙且令人惊讶的例子在 C++标准委员会成员之间流传：[3]

```
//: C05:Factorial.cpp
// 使用模板在编译时计算阶乘
#include <iostream>
using namespace std;

template<int n> struct Factorial {
    enum { val = Factorial<n-1>::val * n };
};

template<> struct Factorial<0> {
    enum { val = 1 };
};

int main() {
    cout << Factorial<12>::val << endl; // 479001600
} ///:~
```

这个程序能输出阶乘 12!的正确结果。但并不令人惊讶。令人惊讶的是，在程序甚至还没有开始运行之前，就已经完成计算了！

当编译器尝试实例化 Factorial<12>时，它发现还必须实例化 Factorial<11>，这又需要 Factorial<10>，以此类推。最终递归终止于 Factorial<1>这个特化，计算展开（unwind）。

---

[1] 译注：Counted 类的大小在编译期就已经确定，因为它的成员变量 count 是静态的，不占用对象的空间。因此，无论用什么类型 T 来实例化 Counted，其大小都是相同的（都为零）。这就实现了一种所谓的“静态多态”效果，即可以实现类似于虚函数的多态性，但发生在编译期，效率更高。

[2] 收录于 *C++ Gems* 一书，本书由 Stan Lippman 主编，SIGS，1996。

[3] 本例的使用的 val 从技术上说是编译时常量，所以你可以会觉得，这种标识符应该全大写以遵循惯例。但是，我们保留了小写形式，因为它们是对变量的模拟。

最终，Factorial<12>::val 被替换为整型常量 479001600，编译结束。由于所有计算都由编译器完成，因此涉及的值必须是编译时常量，这就是为什么要使用 enum 的原因。真正运行程序时，唯一剩下的工作就是输出那个常量，并后跟一个换行符。为了证明是 Factorial 的一个特化导致了正确的编译时值，可以在程序中把它作为数组维数使用，例如：

```
double nums[Factorial<5>::val];
assert(sizeof nums == sizeof(double)*120); // 5! = 120
```

5.6.1 编译时编程

模板最初的设计是为了方便地替换类型参数，但它却发展成支持编译时编程的一种机制。人们将这种技术称为**模板元编程**（因为它相当于在编写一个程序），并将用这种技术编写的程序称为**模板元程序**，其应用范围非常广泛。事实上，模板元编程是**图灵完备**的[①]，因为它支持选择（类似于 if-else）和循环（通过递归）。我们理论上可以用它执行任何计算。[②]上面的阶乘示例展示了如何实现重复：编写一个递归模板，并通过一个特化来提供停止条件。下例展示了如何使用这种技术在编译时计算斐波那契数列：

```
//: C05:Fibonacci.cpp
#include <iostream>
using namespace std;

template<int n> struct Fib {
    enum { val = Fib<n-1>::val + Fib<n-2>::val };
};

template<> struct Fib<1> {
    enum { val = 1 };
};

template<> struct Fib<0> {
    enum { val = 0 };
};

int main() {
    cout << Fib<5>::val << endl;  // 6
    cout << Fib<20>::val << endl; // 6765
```

[①] 译注：图灵完备（Turing complete）是一个计算机科学的概念，用于描述一种计算模型或编程语言的能力。它指的是，这种模型或语言能够模拟任何其他图灵机（一种抽象的计算模型）所能进行的计算。换句话说，如果一个系统是图灵完备的，那么理论上它就能执行任何可计算的任务。

[②] 1966 年，Böhm 和 Jacopini 证明了只要语言支持选择和重复，并且能使用任意数量的变量，那么就等价于图灵机，而图灵机被认为能够表达任何算法。

```
} ///:~
```

斐波那契数列在数学上定义为：

$$f_n = \begin{cases} 0, & n = 0 \\ 1, & n = 1 \\ f_{n-2} + f_{n-1}, & n > 1 \end{cases}$$

前两种情况导致了代码中的两个模板特化，而第三行的规则成为了主模板。

编译时循环

为了在模板元程序中计算任何循环，首先必须将其重新表述为递归形式。假定要计算整数 n 的 p 次幂，但不想使用如下所示的循环：

```
int val = 1;
while(p--)
    val *= n;
```

那么可以将其表示为一个递归过程：

```
int power(int n, int p) {
    return (p == 0) ? 1 : n*power(n, p - 1);
}
```

这就能很轻松地转换为模板元编程的形式：

```
//: C05:Power.cpp
#include <iostream>
using namespace std;

template<int N, int P> struct Power {
    enum { val = N * Power<N, P-1>::val };
};

// 一个部分特化的模板
template<int N> struct Power<N, 0> {
    enum { val = 1 };
};

int main() {
    cout << Power<2, 5>::val << endl; // 32
} ///:~
```

我们需要使用一个部分特化的模板来作为停止条件，其中值 N 仍然是一个自由模板参数。注意，该程序仅适用于非负次幂。

以下模板元程序改编自 Czarnecki 和 Eisenecker。[1]有趣之处在于，它使用了一个“模板模板参数”（将模板作为模板参数传递），并模拟了将一个函数作为参数传递给另一个函数，该函数“遍历”数字 0 到 n：

```
//: C05:Accumulate.cpp
// 编译时将一个“函数”作为参数传递
#include <iostream>
using namespace std;

// 累加 F(0)到 F(n)的结果
template<int n, template<int> class F> struct Accumulate {
    enum { val = Accumulate<n-1, F>::val + F<n>::val };
};

// 停止条件（返回 F(0)的值）
template<template<int> class F> struct Accumulate<0, F> {
    enum { val = F<0>::val };
};

// 各种“函数”：
template<int n> struct Identity {
    enum { val = n };
};

template<int n> struct Square { // 平方
    enum { val = n*n };
};

template<int n> struct Cube {   // 立方
    enum { val = n*n*n };
};

int main() {
    cout << Accumulate<4, Identity>::val << endl; // 1-4 累加：10
    cout << Accumulate<4, Square>::val << endl;   // 1-4 分别求平方后相加：30
    cout << Accumulate<4, Cube>::val << endl;     // 1-4 分别求立方后相加：100
} ///:~
```

主 Accumulate 模板试图计算 F(n)+F(n-1)...F(0)这个累加和。停止条件是通过一个部分特化获得的，后者“返回”F(0)的值。参数 F 本身就是模板，其工作方式和本节前一个例子展示的函数类似。Identity、Square 和 Cube 模板对其名称所暗示的函数进行求值。main 函数中的第一个 Accumulate 实例化计算了累加和 4+3+2+1+0，因为 Identity 函数简

[1] Czarnecki and Eisenecker，*Generative Programming: Methods, Tools, and Applications*，Addison Wesley，2000，p. 417。

单地“返回”其模板参数。[1]main 函数中的第二行对那些数的平方进行求和（16+9+4+1+0），最后一行则计算立方和（64+27+8+1+0）。

循环展开

算法设计者们一直致力于优化他们的程序。一种久经考验的优化技术是**循环展开**（loop unrolling）。它特别适合数值编程，旨在最小化循环开销。循环展开最经典的例子就是矩阵乘法。以下函数乘以一个矩阵和一个向量（假设代表矩阵行列数的常量 ROWS 和 COLS 已预先定义）：[2]

```
// a 是矩阵，x 是向量，y 是结果向量
// i 是矩阵的行索引，j 是矩阵的列索引
void mult(int a[ROWS][COLS], int x[COLS], int y[COLS]) {
    for(int i = 0; i < ROWS; ++i) {
        y[i] = 0;
        for(int j = 0; j < COLS; ++j)
            y[i] += a[i][j]*x[j];
    }
}
```

如果 COLS 是偶数，那么在内层循环中将计算“展开”成对，就可以将递增和比较循环控制变量 j 的开销减半：[3]

```
void mult(int a[ROWS][COLS], int x[COLS], int y[COLS]) {
    for(int i = 0; i < ROWS; ++i) {
        y[i] = 0;
        for(int j = 0; j < COLS; j += 2)
            y[i] += a[i][j]*x[j] + a[i][j+1]*x[j+1];
    }
}
```

一般来说，如果 COLS 是 k 的倍数，那么内层循环每次迭代就可以执行 k 次操作，从而大

---

[1] 译注：这正是为什么取名为“Identity”的原因。在英语中，它表示“同一性”和“身份”。在数学上，它经表示“恒等”。恒等函数 f(x)=x 表示函数的输出值恒等于输入值。形象地说，Identity 就相当于一个“传送带”，它原封不动地将输入的值传送到输出端。而 Accumulate 模板则是一个“累加器”，它可以对经过“传送带”的值进行累加，这些值可以是恒等值、平方值和立方值。

[2] 译注：在这段代码中，外层循环遍历矩阵的每一行。内层循环遍历矩阵的每一列，计算矩阵 a 的第 i 行与向量 x 的点积，并将结果累加到 y[i]中。

[3] 译注：在优化版本中，外层循环仍然遍历矩阵的每一行。内层循环则每次使 j 递增 2，而不是 1。在每次迭代中，同时计算两个元素的乘积并累加到 y[i]中。这就是计算“展开”成对的含义。

幅减少开销。虽然这种时间上的节省在处理大数组时才明显，但这正是工业级数学计算的常见情况。

函数内联也是循环展开的一种形式。考虑以下计算整数幂的方式：[①]

```
//: C05:Unroll.cpp
// 通过内联展开一个隐式的循环
#include <iostream>
using namespace std;

// 当 n 大于 1 时，递归调用自身，计算 m 的 n-1 次幂，然后乘以 m
template<int n> inline int power(int m) {
    return power<n-1>(m) * m;
}

// 模板特化：任何数的 1 次幂都是它本身
template<> inline int power<1>(int m) {
    return m;
}

// 模板特化：任何数的 0 次幂都为 1
template<> inline int power<0>(int m) {
    return 1;
}

int main() {
    int m = 4;
    cout << power<3>(m) << endl; // 64
} ///:~
```

从概念上讲，编译器必须生成 power<>模板函数的三个特化，分别对应模板实参 3、2 和 1。但是，因为每个函数的代码都可以内联，因此实际在 main 函数中实际插入的代码是单一表达式 m*m*m。[②]因此，简单的模板特例化加上内联，提供了一种完全避免循环控制开销的方法。不过，这种方法受限于当前使用的编译器的内联深度。

[①] 译注：本例通过递归隐式地实现一个本来该写成 for 或 while 的循环，内联则将一个函数的代码直接嵌入到调用它的函数中，减少函数调用的开销。power 函数虽然没有显式的循环，但是通过递归调用实现了幂运算。编译器在优化时，可能会将这个递归调用展开成一系列的乘法操作，从而达到类似于循环展开的效果。

[②] 俄氏乘法（Russian Peasant Algorithm）是一种计算整数幂的更有效的方法。

编译时选择

为了在编译时模拟条件语句，可以在 enum 声明中使用条件三元操作符。以下程序使用这种技术来计算两个整数的最大值：

```
//: C05:Max.cpp
#include <iostream>
using namespace std;

template<int n1, int n2> struct Max {
    enum { val = n1 > n2 ? n1 : n2 };
};

int main() {
    cout << Max<10, 20>::val << endl; // 20
} ///:~
```

如果希望使用编译时条件来控制自定义代码生成，那么可以使用 true 和 false 的特化：

```
//: C05:Conditionals.cpp
// 使用编译时条件来选择执行的代码
#include <iostream>
using namespace std;

template<bool cond> struct Select {};

// true 特化
template<> class Select<true> {
    static void statement1() {
        cout << "正在执行语句 1 – statement1\n";
    }
public:
    static void f() {
        statement1();
    }
};

// false 特化
template<> class Select<false> {
    static void statement2() {
        cout << "正在执行语句 2 - statement2\n";
    }
public:
    static void f() {
        statement2();
    }
};
```

```
template<bool cond> void execute() {
    Select<cond>::f();
}

int main() {
    // int 类型的大小确实为 4 个字节
    execute<sizeof(int) == 4>();
} ///:~
```

该程序等价于以下表达式：

```
if(cond)
    statement1();
else
    statement2();
```

区别在于，条件表达式 cond 在编译时求值，而且编译器会实例化适当的 execute<>和 Select<>版本。Select<>::f()函数在运行时执行。可以用类似的方式模拟 switch 语句，只是注意要针对每个 case 值进行特化，而不是特化 true 和 false 值的版本。

编译时断言

在第 2 章中，我们强调了将断言纳入整体防御性编程策略的优点。简单地说，断言就是对布尔表达式进行求值，然后根据求值结果采取适当行动：如果条件为真则什么都不做，否则停止并显示诊断信息。最好尽快发现断言失败。如果能在编译时对一个表达式进行求值，那么就使用编译时断言。下例展示了一种将布尔表达式映射到数组声明的技术：

```
//: C05:StaticAssert1.cpp
#define STATIC_ASSERT(x) \
do { typedef int a[(x) ? 1 : -1]; } while(0)

int main() {
    STATIC_ASSERT(sizeof(int) <= sizeof(long));   // 通过
    STATIC_ASSERT(sizeof(double) <= sizeof(int)); // 失败（编译时报错：负下标）
} ///:~
```

do 循环为数组 a 的定义创建了一个临时作用域，数组 a 的大小取决于特定条件。定义大小为-1 的数组是非法的，因此当条件为假时，该语句应该失败。

上一节展示了如何对编译时布尔表达式进行求值。为了模拟编译时断言，现在唯一剩下的挑战就是如何打印有意义的错误消息并停止。为了使编译器停止，只需造成一个编译错误

即可；难办的是在错误消息中插入有意义的文本。以下示例来自 Alexandrescu[①]，它使用模板特化、局部类[②]以及一些宏魔术来完成任务：

```
//: C05:StaticAssert2.cpp
#include <iostream>
using namespace std;

// 一个模板和一个特化
template<bool> struct StaticCheck {
    StaticCheck(...);
};
template<> struct StaticCheck<false> {};

// 宏（用于生成一个局部类）
#define STATIC_CHECK(expr, msg) { \
    class Error_##msg {}; \
    sizeof((StaticCheck<expr>(Error_##msg()))); \
}

// 检测缩小转换
template<class To, class From> To safe_cast(From from) {
    STATIC_CHECK(sizeof(From) <= sizeof(To), NarrowingConversion);
    return reinterpret_cast<To>(from);
}

int main() {
    void* p = 0;
    int i = safe_cast<int>(p);
    cout << "可以安全转换为 int\n";
    //! char c = safe_cast<char>(p);
} ///:~
```

本例定义了一个函数模板 `safe_cast<>`，它检查要进行强制类型转换的对象是否**不**大于目标类型的大小。如果目标对象类型较小，那么会在编译时通知用户正在尝试“缩小转换”。注意，`StaticCheck` 类模板的奇特之处在于，任何东西都可以转换为 `StaticCheck<true>` 的实例（注意构造函数中的省略号[③]），而没有任何东西可以转换为 `StaticCheck<false>`，因为没有为这一特化提供转换。这个程序的思路是，只要所测试的条件为真，就尝试在编译时创建新类的实例，并尝试将其转换为 `StaticCheck<true>`，或者在条件为假时转换为

[①] *Modern C++ Design*，pp.23–26。

[②] 译注：记住，在函数内定义的类称为局部类，在类内定义的另一个类称为嵌套类。

[③] 不允许将除内置类型之外的对象类型传递给省略号参数。但是，由于我们只想获取其大小（一种编译时操作），因此该表达式在运行时从未进行求值。

StaticCheck<false>。由于sizeof操作符会在编译时求值，因此用它来尝试转换。如果条件为假，那么编译器会抱怨不知道如何将新的类类型转换为 StaticCheck<false>。（在STATIC_CHECK()中，sizeof 调用内额外的圆括号是为了防止编译器认为我们试图在一个函数上调用 sizeof，这自然是非法的。）为了在错误消息中插入一些有意义的信息，我们在新类的名称中携带了一些关键文本。

为了真正理解这一技术，最佳的方式就是通过一个实际的例子来逐步分析。来考虑main()函数中的下面这一行：

```
int i = safe_cast<int>(p);
```

由于调用了 safe_cast<int>(p)，所以该函数定义的第一行代码会进行如下所示的宏展开：

```
{                                                   \
    class Error_NarrowingConversion {};             \
    sizeof(StaticCheck<sizeof(void*) <= sizeof(int)> \
            (Error_NarrowingConversion()));         \
}
```

（记住，预处理操作符##将其操作数拼接成单个标记。因此，Error_##NarrowingConversion 在预处理后就成了 Error_NarrowingConversion）。Error_NarrowingConversion 是一个局部类（这意味着它在一个非命名空间的作用域内声明），因为它在程序其他任何地方都用不上。通过在这里应用 sizeof 操作符，我们尝试确定 StaticCheck<true>的一个实例的大小（因为在我们的平台上 sizeof(void*) <= sizeof(int)为真），注意，该实例是从 Error_NarrowingConversion()这个调用所返回的临时对象隐式创建的。由于编译器知道新类 Error_NarrowingConversion 的大小（它是空的），因此可以合法地在 STATIC_CHECK()宏中的外层使用 sizeof。由于从 Error_NarrowingConversion 临时对象到 StaticCheck<true>的转换成功，因此外层的sizeof也成功。执行将继续，并输出“可以安全转换为int”。

现在，考虑如果删除main()函数最后一行的注释会发生什么：

```
char c = safe_cast<char>(p);
```

在这种情况下，safe_cast<char>(p)内部的STATIC_CHECK()宏展开为：

```
{                                                   \
    class Error_NarrowingConversion {};             \
    sizeof(StaticCheck<sizeof(void*) <= sizeof(char)> \
            (Error_NarrowingConversion()));         \
}
```

由于表达式 `sizeof(void*) <= sizeof(char)` 为假[①]，因此现在是尝试从 `Error_NarrowingConversion` 临时对象转换为 `StaticCheck<false>`，如下所示：

```
sizeof(StaticCheck<false>(Error_NarrowingConversion()));
```

这会导致失败，因此编译器终止程序并报告以下错误消息（MSVC 编译器的结果）：

```
“<function-style-cast>”：无法从“safe_cast::Error_NarrowingConversion”转换为
“StaticCheck<false>”
```

`Error_NarrowingConversion` 类名本身就是一条有意义的消息，是由程序员精心挑选的。一般来说，要用这种技术执行静态断言，只需使用 `STATIC_CHECK` 宏来检查编译时条件，并使用一个能够描述错误的有意义的名字。

5.6.2 表达式模板

模板最强大的应用或许就是**表达式模板**，这一技术由 Todd Veldhuizen[②]和 Daveed Vandevoorde[③]在 1994 年各自独立发现。表达式模板使得某些计算能够在编译时得到全面优化，结果生成的代码至少与手工优化的 Fortran 一样快。与此同时，还通过重载操作符保留了数学的自然表示。虽然在日常编程中不太可能用到这种技术，但它是一些用 C++编写的高性能数学库的基础。[④]

为了理解人们为什么需要表达式模板，下面来考虑典型的数值线性代数运算，例如将两个矩阵或向量[⑤]相加，如下所示：

```
D = A + B + C;
```

按照初学者的实现方式，这个表达式会导致多个临时对象——一个针对 `A+B`，另一个针对 `(A+B)+C`。如果这些变量代表的是巨大的矩阵或向量，那么对资源的消耗是不可接受的。

[①] 译注：在主流系统中，指针类型、`int` 和 `char` 的 size 分别为 4、4 和 1。

[②] Todd 的原始文章可以在由 Lippman 主编的 *C++ Gems* 一书中找到（SIGS，1996）。还应当指出的是，除了保持数学表示和优化的代码外，表达式模板还允许在 C++库中集成来自其他编程语言的范式和机制，例如 lambda 表达式。另一个例子是神奇的类库 Spirit，它是一个大量使用了表达式模板的解析器。它使我们可以在 C++中直接使用（近似）EBNF 表示法来生成极其高效的解析器。欲知详情，请访问 https://sourceforge.net/projects/spirit/。

[③] 参见他和 Nico 的 *C++ Templates* 一书，本章开头曾引用过这本书。

[④] 例如，Blitz++（https://github.com/blitzpp/blitz）和矩阵模板库（https://github.com/simunova/mtl4）。

[⑤] 这里所说的“向量”是指数学意义上的向量，即固定长度的一维数值数组。

表达式模板允许使用相同的表达式而不生成临时对象。

以下程序定义了一个 MyVector 类来模拟任意大小的数学向量。我们使用一个**非类型模板参数**（参见 5.1.1 节）作为向量的长度。还定义了一个 MyVectorSum 类作为代表 MyVector 对象之和的代理类。这样就可以使用延迟计算（惰性求值），不需要临时对象，按需执行向量的各个“分量”的加法。

```
//: C05:MyVector.cpp
// 通过模板优化掉临时变量
#include <cstddef>
#include <cstdlib>
#include <ctime>
#include <iostream>
using namespace std;

// 用于表示“向量和”的代理类
template<class T, size_t N> class MyVectorSum;

template<class T, size_t N> class MyVector {
    T data[N];
public:
    MyVector<T,N>& operator=(const MyVector<T,N>& right) {
        for(size_t i = 0; i < N; ++i)
            data[i] = right.data[i];
        return *this;
    }
    MyVector<T,N>& operator=(const MyVectorSum<T,N>& right);
    const T& operator[](size_t i) const { return data[i]; }
    T& operator[](size_t i) { return data[i]; }
};

// 代理类容纳了引用，并使用延迟加法
template<class T, size_t N> class MyVectorSum {
    const MyVector<T,N>& left;
    const MyVector<T,N>& right;
public:
    MyVectorSum(const MyVector<T,N>& lhs,
                const MyVector<T,N>& rhs)
                    : left(lhs), right(rhs) {}
    T operator[](size_t i) const {
        return left[i] + right[i];
    }
};

// 用于支持 v3 = v1 + v2 的操作符
template<class T, size_t N> MyVector<T,N>&
MyVector<T,N>::operator=(const MyVectorSum<T,N>& right) {
    for(size_t i = 0; i < N; ++i)
        data[i] = right[i];
```

```
    return *this;
}

// operator+只存储引用
template<class T, size_t N> inline MyVectorSum<T,N>
operator+(const MyVector<T,N>& left, const MyVector<T,N>& right) {
    return MyVectorSum<T,N>(left, right);
}

// 为稍后的测试程序提供便利的辅助函数（init 和 print）
template<class T, size_t N> void init(MyVector<T,N>& v) {
    for(size_t i = 0; i < N; ++i)
        v[i] = rand() % 100;
}

template<class T, size_t N> void print(MyVector<T,N>& v) {
    for(size_t i = 0; i < N; ++i)
        cout << v[i] << ' ';
    cout << endl;
}

int main() {
    srand(time(0));
    MyVector<int, 5> v1;
    init(v1);
    print(v1);
    MyVector<int, 5> v2;
    init(v2);
    print(v2);
    MyVector<int, 5> v3;
    v3 = v1 + v2;
    print(v3);
    MyVector<int, 5> v4;
    // 还不支持：
    // v4 = v1 + v2 + v3;
} ///:~
```

MyVectorSum 类在创建时不执行任何计算；它只负责容纳对要相加的两个向量的引用。只有在访问向量和的一个“分量”时，才会实际执行加法运算（参见它的 operator[]()）。在 MyVector 类中，有一个赋值操作符重载接受 MyVectorSum 参数，它是为了处理如下所示的表达式：

```
v3 = v1 + v2; // 两个向量相加
```

对表达式 v1 + v2 进行求值时，会返回一个 MyVectorSum 对象（实际上，由于 operator+() 被声明为 inline，因此该对象会被直接嵌入到代码（编译流）中，避免了函数调用的开销）。这是一个很小的固定大小的对象（其中只容纳了两个引用，分别指向 v1 和 v2）。然后，会在 v3 上调用前面提到的赋值操作符：

```
v3.operator=<int, 5>(MyVectorSum<int,5>(v1, v2));
```

在这个赋值过程中，v3 的每个元素都会被赋值为 v1 和 v2 对应元素之和。这个加法操作是在赋值过程中才真正进行的，而不是在创建 MyVectorSum 对象时，因此不会创建临时 MyVector 对象。

不过，这个程序还不允许表达式中的操作数超过两个，例如以下表达式：

```
v4 = v1 + v2 + v3;
```

原因在于，在第一次加法之后，又尝试了一次加法：

```
((v1 + v2) + v3);
```

这需要 operator+()的一个新的重载，其第一个实参是 MyVectorSum 类型，第二个是 MyVector 类型。虽然可以提供多个重载以应对所有情况，但更好的方法是让模板来做这项工作，如以下修改版本的程序所示：

```
//: C05:MyVector2.cpp
// 使用表达式模板处理任意长度的求和表达式
#include <cstddef>
#include <cstdlib>
#include <ctime>
#include <iostream>
using namespace std;

// 用于表示“向量和”的代理类
template<class T, size_t N, class Left, class Right> class MyVectorSum;

template<class T, size_t N> class MyVector {
    T data[N];
public:
    MyVector<T,N>& operator=(const MyVector<T,N>& right) {
        for(size_t i = 0; i < N; ++i)
            data[i] = right.data[i];
        return *this;
    }
    template<class Left, class Right> MyVector<T,N>&
    operator=(const MyVectorSum<T, N, Left, Right>& right);
    const T& operator[](size_t i) const {
        return data[i];
    }
    T& operator[](size_t i) {
        return data[i];
    }
};

// 允许混合使用 MyVector 和 MyVectorSum
template<class T, size_t N, class Left, class Right> class MyVectorSum {
```

```
    const Left& left;
    const Right& right;
public:
    MyVectorSum(const Left& lhs, const Right& rhs) : left(lhs), right(rhs) {}
    T operator[](size_t i) const {
        return left[i] + right[i];
    }
};

template<class T, size_t N>
template<class Left, class Right>
MyVector<T,N>& MyVector<T,N>::
    operator=(const MyVectorSum<T, N, Left, Right>& right) {
    for(size_t i = 0; i < N; ++i)
        data[i] = right[i];
    return *this;
}

// operator+只存储引用
template<class T, size_t N>
inline MyVectorSum<T, N, MyVector<T,N>, MyVector<T,N> >
operator+(const MyVector<T,N>& left, const MyVector<T,N>& right) {
    return MyVectorSum<T, N, MyVector<T,N>, MyVector<T,N> >(left, right);
}

template<class T, size_t N, class Left, class Right>
inline MyVectorSum<T, N, MyVectorSum<T, N, Left, Right>, MyVector<T,N> >
operator+(const MyVectorSum<T, N, Left,
          Right>& left, const MyVector<T,N>& right) {
    return MyVectorSum<T, N, MyVectorSum<T, N, Left, Right>,
        MyVector<T,N>>(left, right);
}

// 为稍后的测试程序提供便利的辅助函数（init 和 print）
template<class T, size_t N> void init(MyVector<T,N>& v) {
    for(size_t i = 0; i < N; ++i)
        v[i] = rand() % 100;
}

template<class T, size_t N> void print(MyVector<T,N>& v) {
    for(size_t i = 0; i < N; ++i)
        cout << v[i] << ' ';
    cout << endl;
}

int main() {
    srand(time(0));
    MyVector<int, 5> v1;
    init(v1);
    print(v1);
```

```
    MyVector<int, 5> v2;
    init(v2);
    print(v2);
    MyVector<int, 5> v3;
    v3 = v1 + v2;
    print(v3);
    // 现在支持了:
    MyVector<int, 5> v4;
    v4 = v1 + v2 + v3;
    print(v4);
    MyVector<int, 5> v5;
    v5 = v1 + v2 + v3 + v4;
    print(v5);
} ///:~
```

模板机制使用模板参数 Left 和 Right 来推断求和结果的实参类型，而不是提前就锁定这些类型。MyVectorSum 模板通过使用这两个额外的参数，就可以表示 MyVector 和 MyVectorSum 的任何组合对之和。

赋值操作符现在是一个成员函数模板。这允许任何<T, N>对与任何<Left, Right>对进行组合。因此，一方面 MyVectorSum 能容纳由 MyVector 和 MyVectorSum 的任意组合构成的“对”，另一方面 MyVector 对象又可以像往常那样从 MyVectorSum 中赋值[①]。

和以前一样，让我们跟踪一次示例赋值以确切了解发生了什么，先从表达式开始：

```
v4 = v1 + v2 + v3;
```

由于结果表达式会变得相当臃肿，因此在后面的解释中，我们将使用 MVS 作为 MyVectorSum 的缩写，并省略模板实参。

第一个操作是 v1+v2，这会调用内联的 operator+()，后者直接将 MVS(v1, v2)嵌入到编译流中。然后将其加到 v3 上，这会因为表达式 MVS(MVS(v1, v2), v3)而生成一个临时对象。整个语句的最终形式是：

```
v4.operator+(MVS(MVS(v1, v2), v3));
```

这个变换由编译器自动完成，这就是为什么这种技术被称为“表达式模板”的原因。MyVectorSum 模板代表的是一个表达式（本例是一个加法表达式），而上面的嵌套调用类似于左结合表达式 v1+v2+v3 的语法树。

Angelika Langer 和 Klaus Kreft 写了一篇优秀的文章来解释如何将这种技术扩展到更复杂的

[①] 译注：和上个例子一样，赋值操作符（函数）遍历 MyVectorSum 对象中的所有元素，并将它们逐个赋值给 MyVector 对象的相应位置。

计算中。[①]

5.7 模型编译模型

您可能已经注意到，在我们的所有模板示例中，每个编译单元[②]都容纳了完全定义好的模板。也就是说，它们都完整地放在单文件程序中，或者放在多文件程序的头文件中。这刚好与传统做法相反。传统做法是将普通函数定义和它们的声明分开，将后者放在头文件中，将实现放在单独的（即.cpp）文件中。

传统上之所以要做这样进行分离，原因是：

- 头文件中的非内联函数体会导致函数被多次定义，从而导致链接错误。
- 在客户端面前隐藏实现，从而减少编译时的耦合。
- 库供应商可以连同头文件一起分发预编译的代码（针对特定编译器），这样用户就看不到函数实现。
- 由于头文件较小，因此编译时间较短。

5.7.1 包含模型

另一方面，模板不是真正的代码，而是用于指示如何生成代码的指令。只有模板实例化才是真实的代码。当编译器在一次编译过程中看到了完整的模板定义，并在同一编译单元中遇到了该模板的实例化点时，它必须处理“在另一个编译单元中可能存在等效实例化点”[③]这一事实。最常见的方法是在每个编译单元中都为实例化生成代码，并让链接器剔除重复项。这一方法同样适用于不能内联的内联函数和虚函数表，[④]这也正是它流行的主要原因之一。尽管如此，一些编译器更倾向于依赖更复杂的方案来避免多次生成特定的实例化。

[①] Langer and Kreft，“C++ Expression Templates”，*C/C++ Users Journal*，March 2003。另见该杂志 2003 年 6 月号 Thomas Becker 关于表达式模板的文章，那篇文章为本节的内容提供了灵感。

[②] 译注：更正式的名称是“翻译单元”。根据预处理器指令，预处理器对每个源代码文件进行文本替换和其他文本处理。一个预处理好的源代码码文件就是一个翻译单元（translation unit）。不过，本书采用“编译单元”的说法。详情请访问 https://en.wikipedia.org/wiki/Translation_unit_(programming)。

[③] 译注：所谓“实例化点”，就是指需要对模板进行实例化的位置，比如模板函数的调用或模板类的对象声明，此时会生成具体的代码。

[④] 译注：内联函数通常会被编译器直接展开到调用处，以减少函数调用的开销。然而，有些内联函数由于各种原因（例如，递归、复杂度等）无法被内联。在这种情况下，“包含模型”仍然有效，因为编译器可以在每个编译单元中生成该函数的代码，并由链接器处理重复。

无论如何，C++编译系统的责任是避免因为多个等效的实例化点而引起的错误。

这种方法的一个缺点是，所有模板源代码都对客户端可见，因此库供应商几乎没有机会隐藏其实现策略。这种**包含模型**[①]的另一个缺点是，由于函数体需要各自独立编译，因此头文件会比实际的大得多。这可能导致比传统编译模型长得多的编译时间。[②]

为了减轻包含模型带来的头文件过大的问题，C++提供了两种（非排他性的）替代代码组织机制：可以使用**显式实例化**（explicit instantiation）对每个特化版本进行手动实例化，或者使用**导出模板**（exported templates），后者可以实现高度独立的编译[③]。

5.7.2 显式实例化

可以向编译器手动发出一个指令，要求它实例化自己选择的任何模板特化。使用这种技术时，对于每个这样的特化，都必须有且只能有一个这样的（实例化）指令；否则可能会报告多重定义错误，这和具有相同签名的普通非内联函数是一样的。为了理解这一点，让我们首先（错误地）将本章前面提到的 `min()`模板声明与其定义分开，并且遵循普通非内联函数的常规模式。下例由 5 个文件组成：

- **OurMin.h**：包含 `min()`函数模板的声明。
- **OurMin.cpp**：包含 `min()`函数模板的定义。
- **UseMin1.cpp**：尝试使用 `min()`的 `int` 实例化。
- **UseMin2.cpp**：尝试使用 `min()`的 `double` 实例化。
- **MinMain.cpp**：调用 `usemin1()`和 `usemin2()`。

```
//: C05:OurMin.h
#ifndef OURMIN_H
#define OURMIN_H
// min()的声明
template<typename T>
const T& min(const T&, const T&);
```

---

① 译注：包含模型（inclusion model）基于这样的思想：在同一个编译单元中，模板的定义和其所有实例化都应该可见。换言之，什么都“包含”在其中。

② 译注：在包含模型中，为了让编译器能够在任何地方实例化模板，模板的完整定义（包括函数体） 必须放在头文件中。对于内联函数，编译器会将函数体直接插入到调用处，如果该模板函数被多次实例化，就会导致代码的重复。对于模板类成员函数，它的定义也必须放在头文件中，这进一步增加了头文件的大小。

③ 译注：换言之，修改一个编译单元对其他编译单元的影响较小。

```
#endif // OURMIN_H ///:~
```

```
// OurMin.cpp
#include "OurMin.h"

// min()的定义
template<typename T>
const T& min(const T& a, const T& b) {
    return (a < b) ? a : b;
}
```

```
//: C05:UseMin1.cpp {O}
#include <iostream>
#include "OurMin.h"
void usemin1() {
    std::cout << min(1,2) << std::endl;
} ///:~
```

```
//: C05:UseMin2.cpp {O}
#include <iostream>
#include "OurMin.h"
void usemin2() {
    std::cout << min(3.1,4.2) << std::endl;
} ///:~
```

```
//: C05:MinMain.cpp
UseMin1 UseMin2 MinInstances
void usemin1();
void usemin2();

int main() {
    usemin1();
    usemin2();
} ///:~
```

当尝试 build 这个程序时[①]，链接器会报告无法解析的外部符号 `min<int>()`和

[①] 译注：编译命令是 `cl /EHsc MinMain.cpp UseMin1.cpp UseMin2.cpp`。

min<double>()。原因是当编译器遇到 UseMin1 和 UseMin2 中对 min()特化的调用时，只有 min()的声明才是可见的。由于定义不可用，因此编译器假定它们来自其他编译单元。由于在那个位置所需的特化没有被实例化，因此链接器最后会抱怨找不到它们。

为了解决这个问题，我们需要引入一个新文件 MinInstances.cpp，它显式地实例化所需的 min()特化：

```
//: C05:MinInstances.cpp {O}
#include "OurMin.cpp"

// 显式实例化 int 和 double 这两个版本的特化
template const int& min<int>(const int&, const int&);
template const double& min<double>(const double&, const double&);
```

要手动实例化一个特定的模板特化，只需在特化声明前加上 template 关键字。注意，这里包含的必须是 OurMin.cpp 而不能是 OurMin.h，因为编译器需要依据模板定义来进行实例化。不过，这也是在程序中唯一需要这么做的地方[①]，因为它提供了我们需要的独一无二的 min()实例化——对于其他文件，仅仅提供声明就足够了。由于是用宏预处理器来包含 OurMin.cpp，因此我们还添加了包含保护机制（#ifndef 和#endif）：[②]

```
//: C05:OurMin.cpp {O}
#ifndef OURMIN_CPP
#define OURMIN_CPP
#include "OurMin.h"

// min()的定义
template<typename T>
const T& min(const T& a, const T& b) {
    return (a < b) ? a : b;
}
#endif // OURMIN_CPP ///:~
```

现在，将所有文件一起编译成完整的程序[③]，就可以顺利找到 min()的唯一实例化。程序将正确执行，并产生以下输出：

[①] 如前所述，每个程序都只能显式实例化一次模板。

[②] 译注：对于大多数现代项目，使用#pragma once 指令是一个更好的选择，因为它简洁且高效。但是，如果项目需要高度的跨平台兼容性，或者使用的是非常严格的标准化环境，本书所采用的传统的头文件保护宏仍然是一个可靠的选择。要想更多地了解在现代 C++中如何防止重复包含同一个定义，请参见《学习 C++20（中文版）》，清华大学出版社 2023 年出版。

[③] 译注：编译命令是 cl /EHsc MinMain.cpp UseMin1.cpp UseMin2.cpp MinInstances.cpp。

```
1
3.1
```

也可以手动实例化类和静态数据成员。当显式实例化一个类时，针对所请求的特化，除了之前可能已经显式实例化的成员函数，其他所有成员函数都会被实例化。这一点很重要，因为这会使许多模板在使用这种机制时变得无用——特别是那些根据其参数化类型而实现不同功能的模板。隐式实例化在这方面有优势：只有被实际调用的成员函数才会实例化。

显式实例化在大型项目中更有意义，因为可以节省大量编译时间。注意，无论是使用包含模型还是分离模型（参见下一节），都可以选择隐式或显式实例化。这两种选择是相互独立的，可以根据具体情况进行组合。

5.7.3 分离模型

分离模型通过将函数模板定义或静态数据成员定义与其声明分离到不同的编译单元中，从而实现与普通函数和数据类似的组织方式。这种分离是通过对模板进行“导出”（参见5.7.1 节末尾）来实现的。在读完前两节后，你可能会感到困惑：既然可以维持现状，为什么还要使用包含模型呢？这背后既有历史原因，也有技术原因。

历史上，包含模型是最早经历全面商业应用的——所有 C++编译器都支持包含模型。部分原因是分离模型直到标准化过程后期才得到良好的定义，而且包含模型更容易实现。在分离模型的语义最终确定之前，已经存在了大量投入实际运行的代码。

分离模型的实现非常困难，以至于截至 2003 年夏季，只有一种编译器前端（EDG）支持分离模型，而且当时仍然要求在编译时能够访问模板的源代码，以实现“按需”实例化。标准委员会的计划是使用某种中间代码形式，而不要求源代码在手。届时就可以在不发送源代码的情况下发送“预编译”的模板。由于本章前面解释的在查找时的复杂性（即在模板定义的上下文中查找依赖名称），因此在编译一个对模板进行实例化的程序时，仍然需要以某种形式获得完整的模板定义。[①]

分离模板定义源代码与声明的语法很简单，使用 `export` 关键字即可：[②]

[①] 译注：C++20 引入的“模块”（Module）机制已经全面解决了这个问题。它提供了一种更高效、更模块化的方式来组织和管理代码，解决了传统头文件“包含”机制的一些问题。通过减少编译时间、提高封装性和简化依赖管理，显著提高了 C++开发效率。要想更多地了解这一主题，推荐阅读《学习 C++20（中文版）》一书，清华大学出版社 2023 年出版。

[②] 译注：`export` 关键字在 C++98 标准中引入，用于支持模板的分离编译，即允许模板的声明和定义分别位于不同的文件中。然而，由于其实现复杂性和实际使用中的问题，`export` 关键字在 C++11 标准中被废弃，并且在后续的 C++ 标准中不再支持。。

```
//: C05:OurMin2.h
// 将 min()声明为导出（的）模板
#ifndef OURMIN2_H
#define OURMIN2_H
export template<typename T>
const T& min(const T&, const T&);
#endif // OURMIN2_H ///:~
```

与 inline 或 virtual 类似，只需在编译流中提及一次 export 关键字，即引入一个导出模板的时候。因此，我们其实无需在实现文件中重复它。但是，目前一般都认为重复它是一个好的编程实践：①

```
//: C05:OurMin2.cpp
// “导出模板”min 的定义
#include "OurMin2.h"
export template<typename T>
const T& min(const T& a, const T& b) {
    return (a < b) ? a : b;
}
```

之前使用的两个 UseMin 文件现在只需包含正确的头文件（OurMin2.h），主程序不需要做任何改变。虽然这似乎提供了真正的分离，但包含模板定义的文件（OurMin2.cpp）还是必须发送给用户（因为在每次实例化 min()时都需要处理它）。除非 C++标准开始支持模板定义的某种形式的中间代码形式，否则这个问题是无解的。因此，虽然标准确实提供了真正的分离模型，但现在还无法完全利用其所有优点。②

5.8 小结

模板的意义远远超出了简单的“类型参数化”，不是用一个实际的类型（例如，int 或 double）来替换类型参数 T 那么简单。通过结合实参类型推断、自定义特化和模板元编程，C++ 模板将会成为一种强大的代码生成机制。

本章没有提到 C++模板的一个弱点，即它在解释编译时错误消息时比较困难。编译器输出

---

① 译注：export 关键字本质上是对编译器的一个指示，告诉它这个模板的定义将在其他地方提供。编译器在遇到 export 声明时，会将这个模板的信息记录下来，并在后续的链接阶段进行匹配。在链接阶段，链接器会根据 export 声明找到对应的模板定义，并将其与使用该模板的代码进行链接。

② 译注：像以前一样编译程序（cl /EHsc MinMain.cpp UseMin1.cpp UseMin2.cpp），编译器会报错：“导出的模板不是当前 C++ 标准的一部分”。如前所述，在现代 C++中，应该使用 C++20 引入的“模块”机制来实现模板声明和定义的分离。欲知详情，请参见《学习 C++20（中文版）》一书，清华大学出版社 2023 年出版。

的一大堆晦涩难懂的文本，可能会令人不知所措。当然，C++编译器也在不断改进其模板错误信息。Leor Zolman 写了一个名为 STLFilt 的工具，通过提取有用信息并丢弃其余部分，使得这些错误信息更加易读。[①]

通过本章的学习，我们可以获得另一个重要的思路，即模板相当于一个接口。换言之，即使 `template` 关键字说“我能接受任何类型”，但模板定义中的代码要求支持某些特定的操作符和成员函数——这其实就是接口！因此模板定义实际上是说“我能接受任何支持此接口的类型。”如果编译器能够简单地说，“嘿，你用来实例化模板的类型不支持那个接口”，那么就更好了。我们可以使用模板来执行类型检查，这其实比纯“面向对象”实践中要求所有类型都派生自特定的基类更为灵活。

第 6 章和第 7 章将深入探讨模板最著名的应用，即标准 C++库的一个子集，通常称为**标准模板库**（STL）。第 9 章和第 10 章也使用了本章没有讨论过的一些模板技术。

5.9 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. 编写接受一个类型参数的一元函数模板。为 `int` 类型创建该模板的全特化版本。另外，为该函数创建一个非模板重载版本，它接受一个 `int` 参数。在主程序中调用这三个函数变体。

2. 编写一个类模板，使用 `vector` 实现栈数据结构。

3. 修改上个练习的答案，将用于实现栈的容器类型作为一个“模板模板参数”。

4. 在以下代码中，`NonComparable` 类没有 `operator==`。为什么 `HardLogic` 结构体会导致编译错误，而 `SoftLogic` 不会？[②]

```
//: C05:Exercise4.cpp {-xo}
class Noncomparable {};

template<class T>
struct SoftLogic {
    Noncomparable nc1, nc2;
    void noOp() {}
    void compare() {
```

[①] 欲知详情，请访问 https://github.com/pranavkantgaur/STLfilt。

[②] 译注：记住，编译器在实例化模板时才会检查 `compare` 方法的具体实现。

```
        nc1 == nc2;
    }
};

struct HardLogic {
    Noncomparable nc1, nc2;
    void compare() {
       return nc1 == nc2; // 编译错误
    }
};

int main() {
    SoftLogic<Noncomparable> l;
    l.noOp();
}
```

5. 编写一个函数模板，它接受一个类型参数 T 和四个函数实参：T 类型的数组、起始索引、结束索引（含）以及一个可选的初始值。函数返回指定范围内的所有数组元素之和以及初始值。用 T 的默认构造函数来生成默认的初始值。

6. 重做上个练习，但按照本章的说明，使用“显式实例化”为 int 和 double 手动创建特化。

7. 以下代码为什么不能编译？（提示：类成员函数有权限访问的是什么？）[1]

```
//: C05:Exercise7.cpp {-xo}
class Buddy {};

template<class T>
class My {
    int i;
public:
    void play(My<Buddy>& s) {
        s.i = 3;
    }
};

int main() {
    My<int> h;
    My<Buddy> me, bud;
    h.play(bud);
    me.play(bud);
} ///:~
```

8. 以下代码为什么不能编译？（提示：无法明确 T 的实际类型。）

[1] 译注：答案在中文版配套代码中。

```
//: C05:Exercise8.cpp {-xo}
template<class T>
double pythag(T a, T b, T c) {
    return (-b + sqrt(double(b * b - 4 * a * c))) / 2 * a;
}

int main() {
    pythag(1, 2, 3);
    pythag(1.0, 2.0, 3.0);
    pythag(1, 2.0, 3.0);
    pythag<double>(1, 2.0, 3.0);
} ///:~
```

9. 编写一个模板来接受以下非类型参数：一个 int、一个指向 int 的指针、一个指向 int 类型的静态类成员的指针和一个指向静态成员函数的指针。

10. 编写一个接受两个类型参数的类模板。为第一个参数定义一个部分特化版本，为第二个参数定义另一个部分特化版本。在每个特化版本中，都引入不在主模板中的成员。

11. 定义一个名为 Bob 的类模板，它接受一个类型参数。使 Bob 成为所有 Friendly 模板类实例的友元。并且只有当 Bob 和 Picky 的类型参数完全一致时，使 Bob 成为 Picky 类模板的友元。为 Bob 添加演示其友元关系的成员函数。[①]

[①] 译注：可以看出，作者对于“类模板”和“模板类”的区分是非常严谨的。简单地说，类模板是抽象的，而模板类是具体的。