第4章　基于库和模式的设计

有经验的C++程序员不会从头开始开发一个项目，而是会从多种不同来源获取代码，如标准模板库、开源库、本单位的专用代码基，以及他们在以前项目中编写的代码。另外，好的C++程序员会重用一些方法或策略来解决各种常见的设计问题。从用于以往项目的某种技术，到正式的设计模式，都属于这样一些策略。本章将介绍设计程序时如何将既有的代码和策略考虑在内。

第2章已经介绍了重用的原则，并解释过这种原则不仅适用于代码重用，还适用于思想重用。本章将详细介绍这个内容，在此会提供一些特定的细节和策略，你可以在程序设计中使用这些策略。读完本章后，应该了解以下内容：

- 可供重用的不同类型的代码
- 代码重用的优点和缺点
- 重用代码的一般策略和原则
- 开源库
- C++标准库
- 设计技术和模式

4.1　重用代码

应当在设计中充分地重用代码。为了更准确地把握这个原则，需要理解可以重用哪些类型的代码，还要了解代码重用中存在哪些折衷考虑。

4.1.1　有关术语

在分析代码重用的优点和缺点之前，先要明确有关的术语，并划分重用代码的类型，这会很有帮助。可供重用的代码主要有三大类：

- 以前你自己写的代码
- 你的同事编写的代码
- 你所在组织或公司之外的某个第三方编写的代码

还可以采用多种方式对你使用的代码加以组织：

- 独立的函数或类。重用你自己的代码或同事所写的代码时，所遇到的一般都是这一类代码。
- 库。库（library）是用于完成一个特定任务（如解析XML）或者处理一个特定领域（如加密）的代码集。使用第三方代码时，这些代码往往都采用库的形式。你可能用过一些简单的库，如C或C++中的数学库，因此对于库应该并不陌生。另外一些通常由库提供的功能包括线程和同步支持、网络和图形等。
- 框架。框架（framework）是你设计程序所基于的一个代码集。例如，MFC基类（Microsoft Foundation Class）就为创建Microsoft Windows的图形用户界面应用提供了一个框架。框架通常

决定了程序的结构。第 25 章提供了有关框架的更多信息。

> 程序要使用（use）库，而要纳入（fit into）框架。库提供了一些特定的功能，而框架则是程序设计和结构的基础。

还有一个词经常出现，这就是应用编程接口（Application Programming Interface，API）。API 是一个库或面向某特定用途的代码体的接口。例如，程序员通常会谈到 socket API，这就是指 socket 网络库的对外接口，而并非库本身。

尽管人们会交替地使用 API 和库这两个术语，但二者并不等同。库指的是实现，而 API 是指库的公开接口。

为简单起见，本章余下部分会用“库”一词来表示所有的重用代码，实际上这可以是一个库、框架，也可以是你同事提供的任何函数集。

4.1.2　决定是否重用代码

从抽象的层次看，重用代码的规则很容易理解。不过，真正落到实处时，就有些含糊了。如何知道什么时候适于重用代码，要重用哪些代码？这往往存在一个折衷，究竟做何决定取决于具体的情况。不过，重用代码有一些一般性的优点和缺点。

重用代码的优点

重用代码可以为你和你的项目提供很多好处。

- 首当其冲地，你可能无法编写（或者不想编写）要重用的代码。你真的想编写代码来处理格式化输入和输出吗？回答当然是否定的，正是出于这个原因，你才会使用 C++ 的标准 I/O 流。
- 重用代码可以节省时间。如果重用了代码，那么这些代码就无须自己编写了。另外，你的设计也将更简单，这是因为，应用中重用的那些组件无需你自行设计。
- 从理论上讲，与自己编写的代码相比，重用的代码需要的调试更少。自己写的代码很可能存在 bug，但一般可以认为库代码是没有 bug 的，因为库已经得到了充分的测试和使用。当然，这也存在例外，也有可能你选用的库编写得很糟糕，存在很多的 bug。
- 库可能还会处理更多的错误条件，而自己写代码时可能想不到这么多。在项目开始时，你可能会忘记处理一些生僻的错误或极端情况，等到以后出现问题时再去修正就会浪费很多时间。或者更糟糕的是，这些错误条件可能会作为 bug 暴露在你的用户面前。重用的库代码往往得到了充分的测试，而且已经由众多程序员使用过，因此可以认为库已经适当地处理了绝大多数错误。
- 与你为某个领域自行编写的代码相比，领域专家编写的重用代码更安全一些。例如，除非你是一个安全领域的专家，否则不要尝试编写自己的安全性代码。如果需要在程序中引入安全性或完成加密，可以直接使用一个库。假设自己编写安全性代码，如果处理不当，许多看似微小的细节则会破坏整个程序的安全性。
- 最后一点，库代码会持续地改进。如果重用代码，就会不断受益于这些改进，而无需自己动手！实际上，如果编写库的人适当地分离了接口和实现，只需升级库版本，不必修改与库的交互，就能获得库改进所带来的好处。真正好的升级只会修改底层实现，而不会修改接口。

重用代码的缺点

遗憾的是，重用代码也存在一些缺点。

- 如果只使用自己写的代码，你就能很清楚地了解这些代码是如何工作的。如果使用了并非自己写

的库，就必须花些时间来理解接口和正确的用法，在此之后才能真正使用它。由于项目开始时会花费这些额外的时间，设计和编写代码就会变慢（推迟）。

- 编写自己的代码时，它会做你想做的工作。但库代码可能并不能完全提供你需要的功能。例如，本书作者之一曾经就犯过一个错误，他没有注意到一个可扩展标记语言（eXtensible Markup Language，XML）解析库中存在一个显著的缺陷，就开始着手使用。乍一看这个库相当不错，它同时支持文档对象模型（Document Object Model，DOM）和面向XML的简单API（Simple API for XML，SAX）解析模型，能够高效运行，而且没有许可费用。但直到真正编写代码时，才注意到这个库不支持按照文档类型定义（Document Type Definition，DTD）进行验证。
- 即使库代码确实提供了你所需的功能，但性能可能不能如你所愿。一般来讲库代码的性能都不好，而且针对你的特定用例，其性能可能相当差，或者根本未公开性能到底如何。另外，编写库或文档的人与你的观念可能有所不同，在判别性能好坏时他们可能有不同的标准。
- 使用库代码的话，在支持方面就像是打开了一个潘多拉盒子。如果发现库中存在一个bug，该怎么办呢？你通常没有办法访问源代码，因此即使想修正错误也无从下手。如果你已经花了很多时间来学习库接口，以及如何使用这个库，很可能不舍得放弃它，但是你会发现，想在你的时间进度之内说服库开发人员修正这个bug可能很困难。另外，倘若你在使用一个第三方库，如果编写库的人不再对这个库提供支持，而你还没有放弃支持依赖于这个库的产品，又该如何是好呢？
- 除了支持问题，库还会带来许可问题。使用开源的库通常要求代码也是开源的。有时库还需要许可费用，在这种情况下，重新开发自己的代码可能更廉价一些。
- 对于重用代码还有一个考虑，这就是跨平台可移植性。大多数库和框架都是特定于具体平台的。例如，毫无疑问，MFC框架主要用于Microsoft Windows。即使是声称能跨平台的代码，在不同平台上也可能会表现出细微的差别。如果你想编写一个跨平台的应用，可能需要在不同的平台上使用不同的库。
- 开源软件也有自己的安全性问题。有些程序员出于安全性原因很忌讳使用开源代码。通过阅读程序的源代码，黑客（恶意攻击者）可能会发现并利用其中的bug，如果不开放源码的话，这些bug可能不会被发现。
- 最后一点，重用代码需要一定的信任。不论是谁编写的代码，都必须相信他，认为他（或她）很好地完成了工作。有些人喜欢对项目的方方面面全盘控制，包括每一行源代码。本书作者之一就发现，很难完全相信不是自己写的库代码。不过，在软件开发中，这通常不是一个有益的观点。

综合来做出决定

既然已经熟悉了重用代码的术语、优点和缺点，应该已经有了更充分的准备，可以决定是否重用代码了。通常，这个决定是很显然的。例如，如果你想用C++为Microsoft Windows编写一个图形用户界面（GUI），就应当使用诸如MFC的一个框架。你可能不知道如何编写底层代码在Windows中创建一个GUI，而且更重要的是，你不想浪费时间来学如何做到这一点。在这种情况下，通过使用框架，就能省下很多人年（person-year）（译者注：人年是开发成本的一个度量单位）。

不过，在另外一些情况下，如何选择可能并不这么明确。例如，如果你对一个库或框架不太熟悉，而且只需要一个简单的数据结构，倘若花时间学习整个框架，又只是使用其中的一个组件（如果你自己编写这个组件，也不过花费几天的功夫），就很不值得了。

最后要说明的是，这个决定是一个主观选择，你要针对自己特定的需求来做出选择。最后往往会在两个时间之间有所权衡，即一方面是如果自行编写需要多少时间，另一方面是学习如何使用一个库来解决问题所需要的时间。请仔细考虑，针对你的特殊情况是否存在上述优缺点，并确定对你来说哪些因素

最重要。最后要记住，你的想法可能并非一成不变！

4.1.3　重用代码的策略

在使用库、框架、同事提供的代码或你自己的代码时，有一些需要谨记的原则。

理解功能和局限性

花些时间来熟悉重用的代码。不仅要了解它的功能，还要了解它的局限性，这一点很重要。先从文档和公开接口或 API 开始。理想情况下，这对于理解如何使用代码已经足够了。不过，如果库并未将接口和实现清晰地分离，可能就要分析源代码本身了。另外，还可以与使用过这个代码而且能够做出解释的其他程序员交流。要先学习基本功能。如果这是一个库，它会提供哪些行为？如果这是一个框架，那么你的代码如何纳入这个框架？要派生哪些类？哪些代码需要自己编写？依据不同类型的代码，还要考虑一些特定的问题。

以下是要记住的对库或框架通用的几点：

- 此代码对多线程程序是否安全？
- 库或框架需要哪些初始化调用？它需要哪些清除工作？
- 这个库或框架依赖于另外的哪些库？

针对你使用的库调用，要记住以下的几点：

- 如果一个调用返回内存指针，由谁负责释放内存：调用者还是库？如果库负责释放，内存会在何时释放？
- 库调用检查了哪些错误条件，它认为哪些是错误条件？它会如何处理错误？
- 调用的所有返回值（按值传递或按引用传递）是什么？会抛出哪些可能的异常？

针对框架，要记住以下几点：

- 如果继承自一个类，应当调用哪一个构造函数？需要覆盖哪些虚方法？
- 哪些内存要由你负责释放，而哪些内存要由框架负责释放？

理解性能

要知道库或其他代码提供怎样的性能保证，这很重要。即使你的程序不强调性能，也应当确保所用代码在用于特定用途时不会性能太差。例如，一个用于 XML 解析的库可能声称很快，但实际上它会在文件中存储临时信息，这会带来磁盘 I/O 读写，相应地会使性能严重降低。

大 O 记法

程序员通常会用大 O 记法（big-O notation）来讨论和说明算法与库的性能。本节将解释算法复杂性分析的一般概念和大 O 记法，但不会提供太多不必要的数学知识。如果你对这些概念已经很熟悉了，可以跳过本节。

大 O 记法指定的是相对性能，而非绝对性能。例如，我们不会说一个算法运行的具体时间（如 300 毫秒），而是会用大 O 记法指出算法的性能随其输入规模的增加将如何变化。排序算法中待排序的项数、基于键查找时散列表中的元素个数，以及要在磁盘间复制的文件的大小，这些都是输入的规模。

注意，大 O 记法只适用于速度依赖于输入的算法。对于无输入或者运行时间不定的算法，大 O 记法并不适用。在实际中，你会发现我们关心的大多数算法的运行时间都依赖于输入，因此这个限制不是大问题。

更正式的理解是：大 O 记法指定了算法运行时间要作为其输入规模的一个函数，这也称为算法的复杂性（complexity）。不过，实际并没有听上去那么复杂。例如，假设一个排序算法要花 50 毫秒完成 500

个元素的排序，花100毫秒完成1 000个元素的排序。由于元素个数加倍时，排序花费的时间也加倍，因此这个算法的性能就是其输入的一个线性函数。也就是说，要画出性能相对于输入规模的图时，这就是一条直线。采用大O记法，就能总结如下这个排序算法的性能：O（n）。这里的O只是表示你在使用大O记法，而n表示输入规模。O（n）指出排序算法的速度是输入规模的一个直接线性函数。

遗憾的是，并非所有算法的性能都随输入规模线性增长。如果真是如此（即所有算法的性能都是输入规模的线性函数），计算机程序运行的速度就会快得多！表4-1总结了几种常见的（性能）函数，按性能从好到坏排列。

表　4-1

算法复杂性	大O记法	解　释	算法举例
常数	O（1）	运行时间独立于输入规模	访问数组中的一个元素
对数	O（log n）	运行时间是输入规模以2为底的一个对数函数	使用二分查找寻找有序表中的一个元素
线性	O（n）	运行时间相对于输入规模呈比例变化	查找无序表中的一个元素
线性对数	O（n log n）	运行时间是输入规模的一个线性函数乘以它的对数函数	归并排序
平方	O（n^2）	运行时间是输入规模的一个平方函数	选择排序之类的较慢排序算法

将性能指定为输入规模的函数而不是绝对的数，这样做有两个优点：

1. 这是平台无关的。指出一段代码在某台计算机上运行时间为200毫秒，并不能说明它在另一台计算机上的运行速度如何。如果不基于完全相同的负载在相同的计算机上实际运行两个不同的算法，将很难对这两个算法作出比较。另一方面，性能如果指定为输入规模的函数，那么它就能应用于任何平台。

2. 将性能指定为输入规模的函数，这样只用一个规范就可以涵盖所有的输入。算法运行所需的具体时间（秒数）只是针对一种特定的输入，并不能说明针对其他输入的性能。

理解性能的有关提示

既然已经熟悉了大O记法，你就能理解大多数性能说明了。特别地，C++标准模板库就使用大O记法来描述其算法和数据结构性能。不过，有时大O记法可能还不够，甚至会产生误导。在考虑大O记法描述的性能规范时，请注意以下问题：

- 如果数据量加倍，算法运行的时间也加倍，这并不能说明这个算法本身的性能好坏！（译者注：虽然采用大O记法性能可以写作O（n），但是还可能有其他因素影响实际性能）。如果算法写得很糟糕，即使它能很好地扩展，你可能也不想使用这个算法。例如，假设算法做了不必要的磁盘访问。这也许不会影响大O记法，但是实际性能会非常差。
- 如果仅基于大O记法，对于采用大O记法运行时间相同的算法，将很难做出比较。例如，如果两个不同的排序算法都声称性能为O（n log n），不实际运行测试的话，就很难区别哪个更快一些。
- 如果输入很少，大O记法给出的时间可能会产生误导。对于小的输入规模，O（n^2）算法可能比O（log n）算法表现更好。在做出决定之前，先要考虑可能的输入规模是多大。

除了考虑大O记法的特点，还应当了解算法性能的其他方面。以下是需要记住的一些原则：

- 应当考虑使用特定库代码段的频度如何。有些人认为“90/10”规则就很有帮助：大多数程序90%的运行时间都只是花在运行10%的代码上（Hennessy and Patterson，2002）。如果你要使用的库代码落在这10%经常运行的代码范围内，就一定要仔细分析它的性能特征。另一方面，如果它落在那90%不太运行的代码范围内，就不必花太多时间分析它的性能，因为它对程序的整体性能没有太大影响。

• 不要太相信文档。一定要运行性能测试来确定库代码是否提供了可以接受的性能。

理解平台局限性

开始使用库代码之前，要保证已经了解它会在哪些平台上运行。听上去可能很显然。当然，如果一个应用还要在 Linux 上运行，就不要在这个应用中使用 MFC。不过，尽管有些库声称是跨平台的，但在不同平台上它们也可能存在微小的差别。

另外，平台不光包括不同的操作系统，还包括同一操作系统的不同版本。如果你编写了一个要在 Solaris 8、Solaris 9 和 Solaris 10 上运行的应用，要确保使用的所有库也支持前述各种版本。不能自认为操作系统各个版本之间存在向前或向后兼容性。也就是说，一个库能在 Solaris 9 上运行，这并不能说明它还能在 Solaris 10 上运行，反之亦然。Solaris 10 上的库可能会使用这个版本新增的操作系统特性或其他库。另一方面，Solaris 9 上的库使用的某些特性在 Solaris 10 中可能已经删除，或者这些库可能使用了一种古老的二进制格式。

理解许可和支持

使用第三方库通常会带来复杂的许可问题。要使用第三方开发商的库，有时必须向他们缴纳一定的许可费用。而且可能还有其他的许可限制，包括出口限制。另外，开源库往往会在一定的许可条件之下发布，即要求使用这些开源库的代码本身也必须是开源的。

> 如果你计划发布或出售你开发的代码，一定先要了解所用第三方库的许可限制。如果有疑问，可以咨询一个法律方面的专家。

使用第三方库还会带来支持问题。在使用一个库之前，要确保了解如何提交 bug，而且要知道修正 bug 需要多长时间。如果可能的话，还要明确这个库还有多长的寿命（即开发商在多长时间内还会对其提供支持），以便做相应的规划。

有意思的是，即使是使用本组织内部的库也可能带来支持问题。你会发现，要想说服本公司另一个部门的一位同事，请他修正他的库中存在的 bug，与说服别家公司里一个素不相识的人相比，难度可能一样大。实际上，你可能会发现说服本公司的人更加困难，因为你并不是一个顾客，没有向他支付任何费用。在使用内部库之前，一定要了解本组织的有关章程条例。

知道从哪里寻求帮助

有时，最初你可能不太敢使用库和框架。幸运的是，你可以从多个渠道获得支持。首先，可以参考库随附的文档。如果这个库得到了广泛使用，如标准模板库（STL），或 MFC，你还能找到有关这一主题的不错的参考书。实际上，要得到有关 STL 的帮助，就完全可以参考这本书中的第 21 章～第 23 章。如果有一些特殊的问题，而在书和产品文档中没有相应的解答，可以在网上查查看。在一个诸如 Google（www.google.com）的搜索引擎上键入问题，找到讨论这个库的相关网页。例如，我在 google 上查“introduction to C++ STL”时，就找到了数百个有关 C++和 STL 的网站。

> 请注意：不要完全相信 Web！与公开出版的书和文档相比，网页一般没有经过同样的审查过程，因此可能存在不正确的地方。

还可以考虑浏览新闻组和注册邮件列表。可以搜索 http://groups.google.com 的 Usenet 新闻组，了解有关库或框架（译者注：更确切的说，应该是你所用的库或框架）的信息。例如，假设你不知道C++标准中不包括 STL 的散列表。在 google 新闻组中搜索“hashtable in C++ STL”时，会发现有很多帖子解释了标准中没有散列表，不过许多开发商以某种方式提供了自己的实现。

新闻组中通常气氛不太好。有些帖子可能很无礼，让人不快，你要根据自己的判断来浏览和发表帖子。

最后一点，许多网站都对特定主题建立了它们自己的专用新闻组，你也可以注册这些新闻组。

原型

刚开始接触一个新库或框架时，编写一个快速原型通常是不错的想法。对代码做些试验，这是熟悉库功能的最好的方法。甚至在着手程序设计之前，就应该考虑对库做试验，以便在将其纳入到你的设计之前真正熟悉库的功能和局限性。通过这种经验性的测试，你还能确定库的性能特征。

即使原型应用看上去根本不像是最终的应用，花些时间建立原型绝对不是浪费。不过，不要认为必须为实际应用编写一个原型。可以编写一个“假”程序测试你想用的库功能。目的只是要熟悉这个库。

由于时间限制，程序员有时会发现他们的原型演变成了最终产品。如果你建立了一个原型，而它还不足以作为最终产品的基础，那么一定要避免这种用法。

4.1.4 捆绑第三方应用

你的项目可能包括多个应用。也许需要一个Web服务器前端来支持新的电子商务基础设施。你的软件可能要捆绑第三方的应用，如一个Web服务器。这种方法将代码重用发挥到了极致，因为你重用了整个应用！

不过，使用库的大多警告和原则也同样适用于捆绑第三方的应用。一定要具体了解你所做决定的合法性和许可问题。

将你的软件发布版捆绑第三方应用之前，要咨询一个法律方面的专家。

另外，支持问题会变得更加复杂。如果顾客遇到的问题出自你捆绑的Web服务器，他们应该与你联系，还是要与Web服务器开发商联系呢？在发行软件之前一定要解决这些问题。

4.1.5 开源库

开源库是越来越流行的一类可重用代码，开源（open-source）的一般含义是源代码可供任何人查看。如果要在你发布的产品中包括源代码，对此有许多正式的条文和法律，不过对于开源软件，要记住重要的一点是：任何人（包括你）都能查看源代码。需要注意的是，开源并不仅限于开源库。实际上，最著名的开源产品莫过于Linux操作系统了。

开源活动

遗憾的是，开源群体中在术语上有些混乱。首先，对于开源活动就同时有两个不同的名字（有人可能会说这是两个单独但相似的活动）。Richard Stallman和GNU项目就使用自由软件（free software）一词。要注意，“自由”（free）这个词并不表示最终完成的产品没有任何费用就一定可用。开发人员也可以为一个自由软件产品交费，或者根据他们的意愿交一点点费用。实际上，自由一词是指人们可以自由地检查源代码、修改源代码以及重新发布这个软件。要把这个词理解为自由谈话中的“自由”；而非免费啤酒中的“免费”。通过访问www.gnu.org还能了解到有关Richard Stallman和GNU项目的更多信息。

不要把自由软件与免费软件混为一谈。免费软件（freeware）或**共享软件**（shareware）无需费用就可以得到，不过源代码可能是专有的，或私有的（proprietary）。自由软件则不同，可能需要一定费用才能使用，不过源代码肯定可以得到。

开源发起组织（Open Source Initiative）使用开源软件（open-source software）一词来描述源代码可用的软件。与自由软件一样，开源软件不要求产品或库一定要免费使用。可以访问 www.opensource.org 来了解有关 Open Source Initiative 的更多信息。

由于“开源”比“自由软件”含义更明确，这本书将使用“开源”来表示源代码可用的产品和库。选择这个名字并不表示我们认为开源理论优于自由软件理论，做此选择只是为了便于理解。

查找并使用开源库

不论选用哪一个词（开源还是自由软件），通过使用开源软件总能获得很多好处。最主要的好处就是功能。已经有大量可用的开源 C++库能够完成各种各样的任务，从 XML 解析到跨平台的错误日志记录，都能找到有关的库。

尽管不要求开源库提供免费的发布版本和许可，但许多开源库确实无需付费就可以拿到。使用开源库的话，在许可费用这个环节上总能省一些钱。

最后一点，通常可以自由地修改开源库来满足具体需求。

大多数开源库都可以在网上获得。可以尝试在 google 上查找需要的东西。例如，如果查找的串是“open-source C++ library XML parsing”，Google 上首先找到的链接就是一组 C 和 C++的 XML 库的链接，包括 libxml 和 Xerces C++ Parser。

还有一些开源门户网站，可以在这里搜索你要找的东西，包括：

- www.opensource.org
- www.gnu.org
- www.sourceforge.net

通过自己的搜索，应该能很快在网上发现更多的资源。

使用开源代码的原则

开源库会带来一些特有的问题，并且需要新的策略。首先，开源库通常是人们在其“自由”时间内编写出来的。任何想要投入和参与开发或修正 bug 的程序员一般都可以得到源码基。作为程序设计群体中的一个好公民，如果你发现自己从开源库中得到了好处，也应该尝试为开源项目做点贡献。如果你为一家公司工作，管理层往往不能接受这种想法，因为这不会为公司带来任何直接收益。不过，你可以向管理层指出这样做的间接好处，比如你的公司可能会因此而声名大增，如果能使公司支持开源活动，你就能更好地从事有关工作。

其次，由于开源库开发本身所具有的分布性，而且缺乏一个具体的所有人，所以开源库往往会带来支持问题。如果你非常需要修正一个库中的某个 bug，可以自行修正，这通常比等着别人来修正效率更高。如果你确实修正了 bug，就应该把所做的修正放到该库的公共源代码基中。即使你没有修正任何 bug，也应当报告发现的问题，这样其他程序员就不会因为遇到同样的问题而浪费时间。

> 在使用开源库时，一定要遵从开源活动的“自由”原则。不要滥用这种自由，或者只是获取而不付出。

4.1.6　C++标准库

作为一个 C++程序员，要使用的最重要的库就是 C++标准库。顾名思义，这个库是 C++标准的一部分，因此遵循标准的任何编译器都带有这个库。标准库并不是单独的一个库，它包括多个分开的组件，其中有一些你可能已经用到了。甚至可以认为这些库是核心语言的一部分。这一节将从设计的角度介绍标准库中的各个组件。你将了解到可以使用哪些工具，不过在此你不会看到编写代码的细节。有关的详

细内容将在本书其他章中介绍。

需要注意以下概述并不全面。有些详细内容将在本书后面更合适的地方做讲解，而有些细节则会完全忽略。标准库内容太多，要在一本一般性的C++书中完整地介绍它不太可能，专门有一些介绍标准库的书，而这些书本身就有800页之多！

C标准库

由于C++是C的一个超集，因此整个C库仍然可用。其功能包括数学函数（abs()、sqrt()和pow()），利用srand()和rand()提供随机数，以及一些错误处理辅助工具（如assert()和errno）。另外，C库中有一些将字符数组作为字符串来管理的工具，如strlen()和strcpy()，还有一些C风格的I/O函数，如printf()和scanf()，这些在C++中还能用。

C++提供了比C更好的字符串和I/O支持。尽管在C++中还能使用C风格的字符串和I/O例程，不过要避免使用这些函数，而应当转而使用C++的字符串和I/O流。

我们假设你对C库已经很熟悉了。如果不是这样，可以参考附录B所列的某一本C参考书。还要注意，C的头文件在C++中文件名会有不同。有关详细内容，请参见网站上的标准库参考（Standard Library Reference）资源。

字符串

C++提供了内置的string类。尽管你可能还在使用C风格的字符串（字符数组），但不论从哪个方面讲，C++的string类都要更胜出一筹。C++的string类会处理内存管理；提供某种越界检查、赋值语义和比较；而且还支持诸如连接、子串抽取和子串或字符替换等处理。

> 从理论上讲，C++ string实际上是一个typedef类型名，这是basic_string模板的一个char实例化。不过，无需你操心这些细节；你可以简单地使用string，就好像它是一个真的非模板类一样。

如果忘记了有关内容，可以参考第1章，其中回顾了字符串类的功能。网站上的标准库参考（Standard Library Reference）资源则提供了更详细的信息。

I/O流

C++引入了使用流（stream）完成输入和输出的新模型。C++库对于从文件、控制台/键盘和字符串读取内置类型提供了一些标准例程。另外，C++还提供了一些工具以便编写自己的例程来读写自己的对象。

第1章复习了I/O流的基础知识。第14章将介绍流的详细内容。

国际化

C++还提供了对国际化（internationalization）的支持。基于这些特性，可以编写使用不同语言、字符格式和数字格式的程序。

第14章将讨论国际化。

智能指针

C++提供了一个有限的智能指针模板，称为auto_ptr。基于这个模板类，可以包装任何类型的指针，这样当它出了作用域时（译者注：即其工作结束时）就会对所包装的指针自动地调用delete。

不过，这个类并不支持引用计数，因此指针一次只能有一个所有者。

第13章、第15章、第16章和第25章将更详细地讨论智能指针。

异常

C++语言支持异常，从而允许函数或方法将不同类型的错误上传至调用函数或方法（即调用此函数

的上层函数)。C++标准库提供了异常的类层次体系，可以在你的程序中加以使用，或者可以派生这些异常来创建自己的异常类型。第 15 章介绍了异常的详细内容和标准异常类。

数学工具

C++库提供了一些数学工具类。尽管这些类都是模板类，以便这些数学工具可以用于任何类型，但一般并不认为这些数学工具类是标准模板库的一部分。除非使用 C++来完成数值计算，一般不需要使用这些工具。

标准库提供了复数类，名为 complex，它提供了一个抽象，可以处理包含有实部和虚部的复数。

标准库还包含一个名为 valarray 的类，从数学角度看，这实际上就是一个向量。标准库提供了一些相关的类来表示向量元素的概念。根据这些构造模块，可以建立完成矩阵处理的类。不过，并没有内置的矩阵类。

C++还提供了一种新的方法来得到有关值域的信息，如当前平台上整数可能的最大值。在 C 中，可以访问 # define 宏来得到这些信息（如 INT _ MAX)，尽管在 C++中还可以这样用，不过你还可以使用一组新的 numeric _ limits 模板类。

标准模板库

C++标准库的核心就是它的通用容器和算法。标准库的这个方面通常称为*标准模板库*（standard template library)，或者简称为 STL，这是因为它大量使用了模板。STL 的妙处在于，它以某种方式提供了通用的容器和通用的算法，从而可以在大多数容器中完成大多数算法，而不论容器存储的是何种类型的数据。这一节将介绍 STL 中的各种容器和算法。第 21 章到第 23 章将提供具体的代码来详细说明如何在程序中使用 STL。

STL 容器

STL 提供了大多数标准数据结构的实现。在使用 C++时，不必再编写诸如链表或队列之类的数据结构。*数据结构*（或容器，container）会以某种方式存储信息片（或元素，element)，从而能够适当地存取这些元素。不同的数据结构有不同的插入、删除和存取行为和性能特征。一定要熟悉可用的数据结构，这很重要，这样就能为给定的任务选择最合适的数据结构。

STL 中的所有容器都是模板，因此可以使用这些容器来存储任何类型，从内置类型（如 int 和 double) 到自己的类都可以。需要注意，任何给定容器中都必须存储相同类型的元素。也就是说，不能在同一个队列中同时存储 int 和 double 类型的元素。不过，可以分别创建两个单独的队列，一个用于存储 int，另一个用来存储 double。

> C++ STL 容器是同构的：每个容器中只允许有相同类型的元素。

要注意，C++标准指定了每个容器和算法的接口，而非实现。因此，不同开发商完全可以提供不同的实现。不过，标准还将性能需求指定为接口的一部分，实现必须满足这些性能需求。

本节将提供 STL 中各种可用容器的概述。

向量

向量（vector）会存储一个元素序列，并提供对这些元素的随机访问。可以把向量看作是一个元素数组，当插入元素时它会动态扩展，而且提供了某种越界检查。类似于数组，向量的元素也存储在连续的内存空间中。

C++中的向量是动态数组的同义词：随着所存储元素个数的变化，向量会动态地扩展和收缩。C++ vector 并不是指数学意义上的向量概念。C++ 将数学意义上的向量建模为 valarray 容器。

对向量插入和删除元素时，如果在向量最后位置进行插入和删除，速度会很快（常量时间），但是在别处插入和删除时速度就较慢（线性时间）。此时插入和删除之所以比较慢，是因为操作必须将所有元素“下移”或“上移”一个位置，以便为新元素留出空间（插入操作），或者填充所删除元素原来占的空间（删除操作）。类似于数组，向量提供了快速（常量时间）的元素存取，即可以在常量时间内存取任何元素。

如果需要快速存取元素，但不打算经常增加或删除元素，就应当在程序中使用向量。有一个很好的经验，即只要使用数组的地方都可以用向量。例如，系统监控工具可能会把它所监控的计算机系统列表保存在一个向量中。很少会向这个列表中增加新的计算机，也很少从这个列表中删除当前的计算机。不过，用户可能经常希望查找有关一台特定计算机的信息，因此查找时间应当很快才行。

> 应当尽可能使用向量，而不是数组。

列表

STL列表（list）是标准的链表结构。类似于数组或向量，它也存储了一个元素序列。不过，与数组或向量不同，链表的元素不一定存储在连续的内存空间中。相反，列表中的每个元素指定了在哪里寻找列表中的下一个和前一个元素（通常通过指针来实现）。需要注意，如果列表中的元素同时指出了下一个和前一个元素，这种列表就称为**双向链表**（doubly linked list）。

列表的性能特征与向量恰好相反。列表提供了较慢（线性时间）的元素查找和存取，但是一旦找到相关的位置，完成元素的插入和删除则很快（常量时间）。因此，如果打算插入和删除很多元素，但是不要求查找很快，就应当使用一个列表。例如，在聊天室实现中，可以将当前参与聊天的所有人存储在一个列表中。聊天室里的人可能会频繁地进来出去，所以需要快速的插入和删除。另一方面，很少需要查看列表中的人员，所以即使查找时间较慢，你也不会太在意。

双端队列

双端队列（deque）是double-ended queue的简写。双端队列是介于向量和列表之间的中间产物，不过更接近于向量。与向量类似，它提供了快速（常量时间）的元素存取。另外，类似于列表，在序列两端插入和删除时，速度也很快（摊分常量时间）。不过，与列表不同的是，在序列中间插入和删除时，速度则较慢（线性时间）。

如果需要从序列的两端插入或删除，但仍需快速地存取所有元素，就应当使用双端队列而不是向量。不过，这个需求对于许多编程问题都不适用。在大多数情况下，向量或队列应该就足够了。

> 向量、列表和双端队列也称为**顺序容器**（sequential container），因为它们都存储了一个元素序列。

队列

队列（queue）一词实际上取自英语中对队列的定义，这表示一队人或一队对象。队列容器提供了标准的先进先出（first in, first out，或FIFO）语义。队列作为一个容器，你可以在其一端插入元素，而在另一端将元素取出。元素的插入和删除都很快（常量时间）。

如果想对实际生活中的“先来先服务”语义建模，就应当使用一个队列结构。例如，可以考虑一个银行。顾客到达银行时，他们要排队。出纳完成了第一个顾客的业务后，会为排队的下一个顾客提供服务，这样就提供了一种“先来先服务”的行为。通过把Customer对象存储在一个队列中，可以实现一个银行仿真。每个顾客到达银行时，把他或她增加到队尾。出纳要为顾客提供服务时，处在队头的顾客就会得到服务。如此一来，顾客就会按其到达的顺序获得服务。

优先队列

优先队列（priority queue）提供了一种队列功能，其中每个元素都有一个优先级。元素会按优先级顺序从队列中删除。如果优先级相同，则仍然遵循 FIFO 语义，即第一个插入的元素会先删除。优先队列的插入和删除一般都比简单队列的插入和删除慢，因为必须对元素重新调整顺序，以支持按优先级排序。

可以使用优先队列对“超常队列”建模。例如，在上述银行仿真中，假设有企业账户的顾客比一般的顾客优先级高。实际生活中的许多银行都会排两个队来实现这种行为：一个是企业顾客的队，另一个是其他顾客的队。企业队列中的顾客会在另一个队中的顾客之前得到服务。不过，银行用一个队也可以提供这种行为，只需要简单地将企业顾客移到所有非企业顾客的前面去。在你的程序中，可以使用一个优先队列，其中顾客的优先级要么是企业优先级，要么是一般优先级。所有企业顾客都应当在所有一般顾客之前得到服务，但是在优先级相同的同一组顾客中，仍然按先来先服务的顺序提供服务。

栈

STL 的栈（stack）提供了标准的先进后出（first-in, last-out 或 FILO）语义。与队列相似，会向这个队列中插入和删除元素。不过，在栈中，最近插入的元素会最先删除。栈这个名字得自于这个结构的一个可视化显示，即在一堆对象中，只能看到最上面的一个对象（译者注，栈也称为堆栈）。向栈中增加对象时，就会盖住（隐藏）在它之下的所有对象。

栈是对实际生活中的“先来后服务”行为建模。举例来说，可以考虑一个大城市的停车场，先到的车会被后来的车“堵”在里面。在这种情况下，最后来的车反而能够最早离开。

STL 栈容器提供了元素的快速（常量时间）插入和删除。如果想得到 FILO 语义，应当使用栈结构。例如，一个错误处理工具可能希望把错误存储在一个栈中，这样最后出现的错误能够最早让管理员看到。按 FILO 顺序处理错误通常很有用，因为新的错误有时会使老错误失去意义。

> 从技术上讲，队列、优先队列和栈容器都是**容器适配器**（container adapter）。它们是建立在三种标准顺序容器（向量、双端队列和列表）之上的接口。

集合和多集

STL 中的集合（set）就是一个元素集合（collection）。尽管集合的数学定义指出它是无序的，但是 STL 的集合会按有序的方式存储元素，这样它就能提供相当快的查找、插入和删除。事实上，集合的插入、删除和查找性能都是对数函数，这比向量提供的插入和删除快，并且比列表提供的查找要快。不过，与列表的插入和删除相比会慢一些，另外查找则比向量的查找慢。其底层实现往往是一个平衡二叉树，如果你平常要使用平衡二叉树结构，就应当使用集合。具体地，如果插入/删除和查找一样多，而且希望尽可能地优化，集合就很适用。例如，在一个业务繁忙的书店里，书目记录程序可能就会使用一个集合来存储图书。只要有新书来或者有书卖出，店内的书目列表就必须得到更新，因此插入和删除必须很快。顾客还需要能够查找某一本特定的书，因此这个程序还应当提供快速的查找。

> 如果希望插入、删除和查找的性能相当，则应当使用集合，而不是向量或列表。

需要注意的是，集合中不允许有重复的元素。也就是说，集合中的每个元素必须惟一。如果你希望存储重复的元素，就必须使用一个多集（multiset）。

> 多集只不过是一个允许元素重复的集合。

映射和多映射

映射（map）存储了键/值对。元素按键排序。在其他各个方面，它与集合完全相同。如果想要建立键和值的关联，就应当使用一个映射。例如，在一个在线的多人游戏中，你可能希望存储有关各个玩家的一些信息，如他或她的登录名、实际姓名、IP 地址和其他特征。可以把这个信息存储在一个映射中，并使用玩家的登录名作为键。

多映射（multimap）与映射的关系就相当于多集与集合之间的关系。具体地，多映射就是允许有重复键的映射。

需要注意的是，可以将映射用作为一个关联数组（associative array）。也就是说，可以把它用作为一个数组，其中索引可以是任何类型，如可以是一个字符串。

> 集合和映射容器也称为**关联容器**（associative container），因为它们建立了键与值的关联。应用于集合时，这个词有些让人困惑，因为集合中，键本身就是值。由于这些容器会对其元素排序，所以也称为有序（sorted）关联容器。

位集

C 和 C++程序员经常会在一个 int 或 long 中存储一组标志，每个标志用一位表示。他们使用位操作符（&、|、^、~、≪、和≫）来设置和访问这些位。C++ 标准库则提供了一个 bitset 类来抽象这种位操作，所以不用再使用位处理操作符了。

位集（bitset）容器并不是一般意义上的容器，因为它没有实现一种特定的数据结构以供插入和删除元素。不过，可以把它看作是可读写的布尔值序列。

STL 容器小结

表 4-2 对 STL 提供的容器做了一个总结。在此使用大 O 记法来表示包含 N 个元素的容器的性能特征。表中的 N/A 项表示：相应操作不是容器语义中的一部分。

表 4-2

容器类名	容器类型	插入性能	删除性能	查找性能	何时使用
vector （向量）	顺序容器	在最后位置插入时性能为 O（1），在其他位置插入时性能为 O（*N*）	在最后位置删除时性能为 O（1），在其他位置删除时性能为 O（*N*）	O（1）	需要快速查找，不在意插入/删除的速度慢，就应当使用向量 能使用数组的地方都能使用向量
list （列表）	顺序容器	O（1）	O（1）	O（*N*）	需要快速的插入/删除 不在意查找的速度慢，就应当使用列表
deque （双端队列）	顺序容器	在开始或最后位置插入时性能为O（1），在其他位置插入时性能为 O（*N*）	在开始或最后位置删除时性能为O（1），在其他位置删除时性能为 O（*N*）	O（1）	一般不需要双端队列；可以转而使用一个 vector 或 list
queue （队列）	容器适配器	O（1）	O（1）	N/A	需要一个 FIFO 结构时就可以使用队列
priority_queue （优先队列）	容器适配器	O（log（*N*））	O（log（*N*））	N/A	需要一个带优先级的 FIFO 结构时就可以使用优先队列
stack （栈）	容器适配器	O（1）	O（1）	N/A	需要一个 FILO 结构时就可以使用栈

（续）

容器类名	容器类型	插入性能	删除性能	查找性能	何时使用
set /multiset（集合/多集）	有序关联容器	O (log (*N*))	O (log (*N*))	O (log (*N*))	需要一个元素集合，而且对元素的查找、插入和删除同样多，就可以使用集合/多集
map /multimap（映射/多映射）	有序关联容器	O (log (*N*))	O (log (*N*))	O (log (*N*))	希望将键与值相关联，就可以使用映射/多映射
bitset（位集）	特殊容器	N/A	N/A	O (1)	需要一个标志集合时就可以使用位集

需要注意，string 从技术上讲也是容器。可以认为它们是字符的向量（vector）。因此，以下介绍的一些算法同样适用于 string。

STL 算法

除了容器外，STL 还提供了许多通用算法的实现。算法（algorithm）就是一种完成特定任务的策略，如排序或查找。这些算法也实现为模板，因此在大多数不同的容器类型上都能使用。需要注意，算法一般不是容器中的一部分。STL 采用了一种不寻常的方法，即把数据（容器）与功能（算法）分离。尽管这种方法看上去与面向对象程序设计的精神有所违背，但为了支持 STL 中的通用程序设计，这是必要的。正交性（orthogonality）指导原则就要求算法和容器是独立的，（几乎）任何算法都能用于（几乎）任何容器。

尽管算法和容器理论上是独立的，但是有些容器以类方法的形式提供了某些算法，因为对于这些特定容器来说，通用算法的表现可能不太好。例如，集合提供了自己的 find() 算法，它就比通用的 find() 算法要快。如果提供了方法形式的算法，就应当使用这样的算法，因为通常它的效率更高，或者更适合于当前容器。

需要注意，算法并非直接在容器上工作。它们使用了一个“中间人”，称之为迭代器（iterator）。STL 中的每个容器都提供了一个迭代器，它会把容器中的元素遍历到一个序列中。即使是集合和映射中的元素，迭代器也会临时地将这些容器中的元素转换到序列中。对应各种容器的不同迭代器都遵循标准接口，因此算法可以使用迭代器完成工作，而不必操心底层的容器实现。要了解有关迭代器、算法和容器的所有详细内容，可以参考第 21 章～第 23 章以及网站上的资料。

迭代器作为算法和容器之间的中间人，它们提供了一个标准接口，可以遍历访问容器中的元素，并置于一个序列中，这样任何算法都可以在任何容器上工作。后面将进一步讨论迭代器设计模式。

STL 中大约有 60 种算法（要看你怎么来数了），通常可以划分为几个不同的大类。不同书中在此分类上可能稍有不同。这本书中认为所有 STL 算法可以分为 5 类：工具算法、非修改算法、修改算法、排序算法和集合算法。有些类还可以进一步细分。要注意，如果以下算法指定为在一个元素“序列”上工作，那么该序列就会作为一个迭代器提供给算法。

在查看以下算法表时，要记住一点，STL 是由一个委员会设计的。援引一句玩笑话：“斑马就是委员会设计马时得到的杰作。”换句话说，委员会所得到的设计中通常会包含一些额外的或不需要的功能，如非要为马加上斑马的条纹。你可能会发现，STL 中的某些算法同样很奇怪，或者是不必要的。确实如此。你不必使用 STL 提供的每一个算法。重要的是，在需要某个算法的时候只要知道有这样一个算法可用就行了。

工具算法

与其他算法不同，工具算法不在数据序列上工作。之所以认为它们也是STL的一部分，只是因为它们是模板化的算法。

算法名	算法说明
min()、max()	返回两个值中的最小值或最大值
swap()	交换两个值

非修改算法

所谓非修改算法是指，这些算法只查看一个元素序列，并返回有关元素的某个信息，或者在各元素上执行某个函数。既然是“非修改”算法，它们不会修改元素的值，也不会改变序列中元素的顺序。这一类中包括4种算法。下面的各表列出了各种非修改算法，并提供了简要的总结。利用这些算法，应该很少需要编写for循环来对一个值序列进行迭代处理了。

查找算法

算法名	算法说明	是否需要有序序列
find()、find_if()	查找与某个值匹配的第一个元素，或者导致一个谓词返回true的第一个元素	不需要
find_first_of()	类似于find，不过会同时搜索多个元素中的某一个	不需要
adjacent_find()	查找彼此相等的两个连续元素的第一个实例	不需要
search()、find_end()	查找序列中与另一个序列相匹配的第一个子序列（search()）或最后一个子序列（find_end()）	不需要
search_n()	查找等于一个给定值的n个连续元素的第一个实例	不需要
lower_bound()，upper_bound()，equal_range()	查找包括一个特定元素的范围的开始位置（即下界，lower_bound()），结束位置（即上界，upper_bound()），或者上下界（equal_range()）	需要
binary_search()	在一个有序序列中查找一个值	需要
min_element()，max_element()	在一个序列中查找最小或最大元素	不需要

数值处理算法

算法名	算法说明
count()、count_if()	统计与一个值匹配或者导致某个谓词返回true的元素个数
accumulate()	“累计”序列中所有元素的值。默认行为是对元素求和，不过调用者可以提供一个不同的二元函数
inner_product()	类似于累计（accumulate()），但在两个序列上工作。它会基于两个序列中相对应的两个元素（并行元素）调用一个二元函数，并将结果累加。默认的二元函数是乘法，如果序列表示的是数学意义上的向量，那么此算法就会计算向量的点乘
partial_sum()	生成一个新序列，其中每个元素都是源序列中相应位置上的元素及前面所有元素的和（或其他二元操作）
adjacent_difference()	生成一个新序列，其中每个元素都是源序列中相应位置上的元素与其前一个元素的差

比较算法

算 法 名	算法说明
equal()	通过检查两个序列的元素顺序是否相同（即相应位置上的元素是否相同），从而确定两个序列是否相等
mismatch()	返回各序列中与另一个序列同一位置上的元素不匹配的第一个元素
lexicographical _ compare()	比较两个序列，确定它们的“字典”顺序。将第一个序列中的各个元素与第二个序列中相应的元素进行比较。如果一个元素小于另一个元素，那么前一个序列就先于后一个序列。如果两个元素相等，则按顺序比较下一对元素。

运算算法

算 法 名	算法说明
for _ each()	对序列中的每个元素执行一个函数。这个算法对于打印容器中的各个元素很有用

修改算法

修改算法会修改序列中的某些或全部元素。有些算法会就地（in place）修改元素，因此源序列会改变。其他一些算法则会把结果复制到另一个序列中，因此源序列保持不变。下表总结了修改算法。

算 法 名	算法说明
transform()	在每个元素或每对元素上调用一个函数
copy()、copy _ backward()	将元素从一个序列复制到另一个序列
iter _ swap()、swap _ ranges()	交换两个元素或两个元素序列
replace()、replace _ if()、replace _ copy()、replace _ copy _ if()	将与某个值匹配或者导致一个谓词返回 true 的所有元素替换为一个新元素，可以就地替换，也可以将结果复制到一个新序列中
fill()、fill _ n()	将序列中的所有元素设置为一个新值
generate()、generate _ n()	类似于 fill() 和 fill _ n()，不过会调用一个指定的函数来生成放在序列中的值
remove()、remove _ if()、remove _ copy()、remove _ copy _ if()	从序列中删除与某个给定值匹配或导致一个谓词返回 true 的元素，可以就地完成，也可以将结果复制到另一个序列中
unique()、unique _ copy()	从序列中删除连续的重复出现，可以就地完成，也可以将结果复制到另一个序列中
reverse()、reverse _ copy()	将序列中的元素逆序放置，可以就地完成，也可以将结果复制到另一个序列中
rotate()、rotate _ copy()	交换序列的前一半和后一半，可以就地完成，也可以将结果复制到另一个序列中。交换的两个子序列的大小不必相同
next _ permutation()、prev _ permutation()	将序列转换为“下一个”或“前一个”排列，从而修改序列。连续地调用某个排列函数（一直调用 next _ permutation()，或一直调用 prev _ permutation()），就可以得到元素的所有可能的排列

排序算法

排序算法是一类特殊的修改算法，它会对序列中的元素进行排序。STL 提供了多种不同的排序算法，其性能保证也有所不同。

算法名	算法说明
sort()、stable_sort()	就地对元素排序，原先重复元素的先后顺序可能依然保持，也可能有变化（译者注：这就是排序算法的稳定性概念，如果能保持重复元素的先后顺序，则称算法具有稳定性，否则就是非稳定的）。sort()的性能类似于快速排序，stable_sort()的性能类似于归并排序（不过具体算法可能存在差别）
partial_sort()、partial_sort_copy()	对序列进行部分排序：前n个元素会排序，余下的元素则未排序。可以就地完成，也可以将结果复制到一个新序列中
nth_element()	重新查找序列的第n个元素，就好像整个序列已经排序一样
merge()、inplace_merge()	合并两个有序序列，可以就地完成，也可以将结果复制到一个新序列中
make_heap()、push_heap()、pop_heap()、sort_heap()	堆是一种标准数据结构，数组或序列中的元素在堆中以一个半有序的方式排列，因此要找到“堆顶”元素是很快的。利用这4个算法，可以对序列使用堆排序（译者注：这里的堆是指一种可以与完全二叉树对照考虑的序列结构，而堆顶就是序列中的最小或最大元素）
partition()、stable_partition()	对序列排序，从而让某个谓词返回true的所有元素都出现在让该谓词返回false的元素之前，各部分中原来的元素顺序可能保持，也可能不保持
random_shuffle()	随机重排元素，将序列置为无序（unsort）

集合算法

集合算法是一种特殊的修改算法，即在序列上完成集合操作。这些算法最适于处理来自set容器的序列，不过，对于来自大多数容器的有序序列也可以使用集合算法。

算法名	算法说明
includes()	确定一个序列是否是另一个序列的子集
set_union()、set_intersection()、set_difference()、set_symmetric_difference()	对两个有序序列完成指定的集合操作，将结果复制到第三个有序序列中。有关集合操作的解释请参见第22章

选择一个算法

有这么多的算法，而且算法的功能也这么多，刚开始时你可能会被吓住。乍一看确实很难了解如何使用这些算法。不过，既然已经熟悉了有哪些可用的选择，应该能更好地完成程序设计了。要了解如何在代码中使用这些算法，第21章～第23章会介绍有关的详细内容。

STL还少些什么

STL非常强大，但是它也不是十全十美的。以下列出的是STL缺少和未支持的功能：

- STL不支持同步来保证多线程安全。STL标准未提供任何多线程同步支持，因为线程是依赖于平台的。因此，如果你有一个多线程程序，就必须为容器实现自己的同步。
- STL不支持散列表。更一般地说，它不支持任何散列关联容器，在这种关联容器中，元素不是有序的，而是根据散列方法来存储。注意，STL的许多实现提供了一个散列表或散列映射，但是由于它不是标准的一部分，使用这样的实现不是可移植的。第23章提供了一个示例散列映射实现。
- STL不支持任何通用的树或图结构。尽管映射和集合一般都实现为平衡二叉树，但STL并未在接口中对外提供这种实现。如果需要一个树或图结构来完成像编写解析器之类的工作，就需要实现自己的树或图结构，或者在另一个库中找到这样的一个实现。
- STL不支持任何表抽象。如果想实现棋盘之类的东西，可能需要使用一个二维数组。

不过，要记住重要的一点，STL 是可扩展的（extensible）。可以编写自己的容器或算法，它们能够与既有的算法或容器一同工作。因此，如果 STL 没有提供你所需要的东西，可以考虑自行编写所需的代码，让它与 STL 共同达成目的。

决定是否使用 STL

设计 STL 时把功能、性能和正交性摆在了优先的位置上。它的设计并没有太多地考虑易用性，因此，很自然地，最后就表现得不那么容易使用。实际上，本章的介绍还只是触及到 STL 复杂性的皮毛而已。因此，在学习如何使用 STL 时，存在一个很陡的学习曲线。不过，STL 的优点也是显著的。你可以考虑一下，原先在对付链表或平衡二叉树实现中的指针错误时花了多少时间，或者对一个未能正确排序的排序算法进行调试时耗费了多少精力。但如果正确地使用了 STL，就很少需要（甚至不需要）再完成这样的编码工作了。

如果决定在程序中采用 STL，可以参考第 21 章～第 23 章。这几章对于如果使用 STL 提供的容器和算法提供了深入的介绍。

4.2　利用模式和技术完成设计

学习 C++语言和成为一个好的 C++程序员完全是两码事。如果坐下来阅读 C++标准，把每一条都记下来，那么你对 C++的了解与别人并没有两样。不过，如果不查看代码并编写自己的程序，由此来获得一些经验，你肯定成不了一个好的程序员。原因在于，C++语法只是以最原始的方式定义了这种语言能够做什么，而没有指出每个特性应该如何使用。

随着 C++程序员积累了越来越多使用 C++语言的经验，他们会对使用该语言的特性建立起自己的一些方法。C++群体作为一个整体也建立了一些标准方法，以便充分地利用 C++语言，其中有些方法是正式的，有些则是非正式的。在本书中，我们指出了这些可重用的语言应用，这也称为设计技术（design technique）和设计模式（design pattern）。另外，第 25 章和第 26 章对设计技术和模式做了比较完备的介绍。有些模式和技术可能是显而易见的，因为它们只是形式化地表述了一个显然的解决方案。其他一些模式则描述了全新的解决方案，可用以解决以前所遇到的问题。

一定要熟悉这些模式和技术，这很重要，由此可以了解到在什么情况下需要利用其中的某个解决方案来解决特定的设计问题。除了本书中介绍的方法外，还有更多可应用于 C++的技术和模式。尽管作者认为这本书已经介绍了最有用的一些模式，不过你可能还希望参考一本有关设计模式的书，来了解更多的不同模式和技术。请参见附录 B 建议的参考书目。

4.2.1　设计技术

设计技术只是一种用于解决 C++中某个特定问题的标准方法。通常，设计技术的目的是为了克服一个不好的特性，或者解决 C++中存在的语言缺陷。在另外一些情况下，设计技术就是一段代码，可以在多个不同程序中用来解决一个常见的问题。

设计技术示例：智能指针

C++中的内存管理经常会带来错误和 bug。其中许多 bug 都是因为使用动态内存分配和指针造成的。如果在程序中大量使用动态内存分配，并在对象之间传递多个指针，此时往往很难记住要对每个指针调用一次（且仅一次）delete。如果没有这样做，后果是很严重的。如果把动态分配的内存释放了多次，就可能导致内存破坏，如果忘记释放动态分配的内存，又会带来内存泄漏。

智能指针（Smart pointer）可以帮助管理动态分配的内存。从概念上讲，智能指针就是动态分配的内存的一个指针，当该内存出作用域后（译者注：即其工作结束后），这个指针会记住要将此内存释放掉。

在程序中，智能指针通常就是一个对象，其中包含一个常规指针（或哑指针）。这个对象在栈上分配。当对象出作用域后，它的析构函数会在所包含的指针上调用 delete。

注意，有些语言实现提供了垃圾回收（garbage collection）机制，这样程序员不必负责释放任何内存。在这样一些语言中，所有指针都可以认为是智能指针，因为无需记住释放指针所指的任何内存。有些语言（如 Java）提供了垃圾回收作为常规做法，但是相比之下，对于 C++来说，编写一个垃圾回收器相当困难。因此，智能指针只是一种补救技术，用以弥补 C++未在内存管理中提供垃圾回收的这一缺陷。

管理指针时，除了要记住指针出作用域后应撤销（释放）指针，还存在很多其他问题。有时，多个对象或多个代码段包含有同一个指针的多个副本。这个问题称为别名（aliasing）。为了适当地释放所有内存，应当由使用内存的最后一段代码对指针调用 delete。不过，通常很难知道哪一段代码最后使用内存。甚至确定代码的先后顺序也是不可能的，因为这可能取决于运行时的输入。因此，还有一种更复杂的智能指针，它实现了引用计数（reference counting）来跟踪其所有者。如果所有所有者都使用完了指针，引用数就会减至 0，智能指针就会对其底层哑指针调用 delete。许多 C++框架都大量使用了引用计数，如 Microsoft 的对象链接和嵌入（Object Linking and Embedding，OLE）和组件对象模型（Component Object Model，COM）。即使你不想自己实现引用计数，也应当熟悉这些概念，这是很重要的。

C++提供了许多语言特性，使得智能指针更有魅力。首先，可以为所有使用模板的指针类型编写一个类型安全（typesafe）的智能指针类。其次，可以使用操作符重载为智能指针对象提供一个接口，从而允许代码像使用哑指针一样地使用智能指针对象。具体地，可以重载 * 和 -> 操作符，这样客户代码就可以像对正常指针解除引用一样，对一个智能指针对象解除引用。第 25 章提供了一个带引用计数的智能指针实现，可以把它作为插件直接在程序中使用。C++标准库还提供了一个简单的智能指针，称为 auto_ ptr，这在标准库的概述中已经介绍过。

4.2.2 设计模式

设计模式（design pattern）是用以解决一般问题的组织程序的标准方法。C++是一种面向对象语言，因此 C++程序员感兴趣的设计模式通常都是面向对象模式，指出了在程序中组织对象和对象关系的一些策略。这些模式通常可以应用于任何面向对象语言，如 C++、Java 或 Smalltalk。实际上，如果你对 Java 程序设计很熟悉，就会在其中发现这里的很多模式。

相对于设计技术来说，设计模式与具体语言的关系不大。模式和技术之间的区别很模糊，不同的书可能会有不同的定义。本书中把技术定义为一种特定于 C++语言的策略，用于克服语言本身存在的某个缺陷。而模式则是一种有关面向对象设计的更一般的策略，可以应用于任何面向对象语言。

需要注意，许多模式都有多个不同的名字。模式本身之间的区别也有些含糊，不同的资料对它们的描述和分析可能会稍有不同。实际上，取决于你用的书或其他资源，可能发现不同的模式还会有同样的名字。对于哪些设计方法可以作为模式，这一点更是未达成一致。除了少许例外，本书都采用《*Design Patterns: Elements of Reusable Object-Oriented Software*》一书中的术语，这是 Erich Gamma 等人所著的一本经典著作。不过，在适当的时候，本书还会指出其他的一些模式名和变种。

第 26 章提供了不同设计模式的一个名录，其中还包括 C++示例实现。

设计模式示例：迭代器

迭代器模式提供了这样一种机制，可以将算法或操作与其处理的数据相分离。乍一看，这种模式似乎与面向对象程序设计中的一个基本原则相违背，在面向对象程序设计中，要求将对象数据与处理数据的行为组合在一起。尽管在某个层次上看，这个观点是对的，但是需要指出，这种模式并不是提倡将基

本行为从对象中去除。相反，它解决了数据与行为紧耦合时通常出现的两个问题。

将数据和行为紧耦合时，第一个问题是：这会阻碍通用算法的编写和使用，这些算法可以处理多种对象，而所处理的对象并非都在同一个类层次体系中。为了编写通用算法，往往需要某种标准机制来访问对象的内容。

数据与行为紧耦合时的第二个问题是：有时很难增加新的行为。至少，需要访问数据对象的源代码。不过，如果要调整的对象层次体系是一个第三方框架或库的一部分，不允许修改，又当如何呢？如果能增加一个处理数据的算法或操作，而不用修改原来的数据对象层次体系就好了。

你已经看过 STL 中的一个迭代器模式的例子。从概念上讲，迭代器提供了一种机制，允许操作或算法访问一个容器中的元素并置于一个序列中。迭代器这个名字得自于英语单词 iterate（迭代），它的意思就是“重复”。由于迭代器要重复做一个动作，在序列中前移从而到达每一个新元素，因此这个词很贴切。在 STL 中，通用算法就使用迭代器来访问其操作的容器中的元素。STL 定义了一个标准迭代器接口，允许编写能够在任何容器上工作的算法，只要该容器提供了一个有适当接口的迭代器就可以。因此，利用迭代器，就能编写通用算法，而无须修改数据。图 4-1 把迭代器显示为一条装配线，它把数据对象元素发送给一个“操作”。

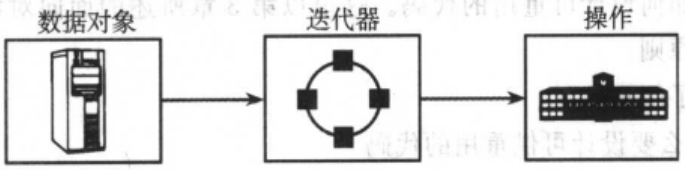

图　4-1

4.3　小结

本章强调了重用这一设计原则。你应当了解到，C++设计应当包括两类重用：代码重用和思想重用，代码重用的形式是库和框架，而思想重用的形式是技术和模式。

尽管代码重用是一个总目标，但是有关代码重用既有优点也有缺点。在本章中，你了解了重用代码的这些权衡考虑，以及有关的特定原则，包括理解功能和局限性、性能、许可和支持模型、平台限制、原型，以及从哪里寻求帮助。你还了解了性能分析和大 O 记法，以及使用开源库时存在的一些特殊问题和考虑。

本章还对 C++标准库提供了一个概述，这也是在编写代码时使用的最重要的一个库。这个库包容了 C 库，并为字符串、I/O、错误处理和其他任务加入了另外一些工具。其中还包括一些通用容器和算法，这些则统称为标准模板库。第 21 章～第 23 章将详细介绍标准模板库。

在设计程序时，重用模式和技术与重用代码同样重要。应当避免重新构建，还要避免重蹈覆辙！最后，这一章介绍了设计技术和模式的概念。第 25 章～第 26 章提供了另外一些例子，包括技术和模式的代码和示例应用。

不过，使用库和模式只是重用策略中的一部分。还需要适当地设计（design）自己的代码，以便你和别人尽可能地重用。第 5 章将介绍设计可重用代码的策略。