C++ STL 编程轻松入门基础

作为 C++标准不可缺少的一部分，STL 应该是渗透在 C++程序的角角落落里的。STL 不是实验室里的宠儿，也不是程序员桌上的摆设，她的激动人心并非昙花一现。本教程旨在传播和普及 STL 的基础知识，若能借此机会为 STL 的推广做些力所能及的事情，到也是件让人愉快的事情。

1 初识 STL：解答一些疑问

1.1 一个最关心的问题：什么是 STL

"什么是 STL？"，假如你对 STL 还知之甚少，那么我想，你一定很想知道这个问题的答案，坦率地讲，要指望用短短数言将这个问题阐述清楚，也决非易事。因此，如果你在看完本节之后还是觉得似懂非懂，大可不必着急，在阅读了后续内容之后，相信你对 STL 的认识，将会愈加清晰、准确和完整。不过，上述这番话听起来是否有点像是在为自己糟糕的表达能力开脱罪责呢？:）

不知道你是否有过这样的经历。在你准备着手完成数据结构老师所布置的家庭作业时，或者在你为你所负责的某个软件项目中添加一项新功能时，你发现需要用到一个链表(List)或者是映射表（Map）之类的东西，但是手头并没有现成的代码。于是在你开始正式考虑程序功能之前，手工实现 List 或者 Map 是不可避免的。于是……，最终你顺利完成了任务。或许此时，作为一个具有较高素养的程序员的你还不肯罢休（或者是一个喜欢偷懒的优等生:），因为你会想到，如果以后还遇到这样的情况怎么办？没有必要再做一遍同样的事情吧！

如果说上述这种情形每天都在发生，或许有点夸张。但是，如果说整个软件领域里，数十年来确实都在为了一个目标而奋斗--可复用性(reusability)，这看起来似乎并不夸张。从最早的面向过程的函数库，到面向对象的程序设计思想，到各种组件技术（如：COM、EJB），到设计模式（design pattern）等等。而 STL 也在做着类似的事情，同时在它背后蕴涵着一种新的程序设计思想--泛型化设计（generic programming）。

继续上面提到的那个例子，假如你把 List 或者 map 完好的保留了下来，正在暗自得意。且慢，如果下一回的 List 里放的不是浮点数而是整数呢？如果你所实现的 Map 在效率上总是令你不太满意并且有时还会出些 bug 呢？你该如何面对这些问题？使用 STL 是一个不错的选择，确实如此，STL 可以漂亮地解决上面提到的这些问题，尽管你还可以寻求其他方法。

说了半天，到底 STL 是什么东西呢？

STL（Standard Template Library），即标准模板库，是一个具有工业强度的，高效的 C++程序库。它被容纳于 C++标准程序库（C++ Standard Library）中，是 ANSI/ISO C++标准中最新的也是极具革命性的一部分。该库包含了诸多在计算机科学领域里所常用的基本数据结构和基本算法。为广大 C++程序员们提供了一个可扩展的应用框架，高度体现了软件的可复用性。

从逻辑层次来看，在 STL 中体现了泛型化程序设计的思想（generic programming），引入了诸多新的名词，比如像需求（requirements），概念（concept），模型（model），容器（container），算法（algorithmn），迭代子（iterator）等。与 OOP（object-oriented programming）中的多态（polymorphism）一样，泛型也是一种软件的复用技术。

从实现层次看，整个 STL 是以一种类型参数化（type parameterized）的方式实现的，这种方式基于一个在早先 C++标准中没有出现的语言特性--模板（template）。如果查阅任何一个版本的 STL 源代码，你就会发现，模板作为构成整个 STL 的基石是一件千真万确的事情。除此之外，还有许多 C++的新特性为 STL 的实现提供了方便。

不知你对这里一下子冒出这么多术语做何感想，希望不会另你不愉快。假如你对它们之中的大多数不甚了解，敬请放心，在后续内容中将会对这些名词逐一论述。正如开头所提到的。

有趣的是，对于 STL 还有另外一种解释--STepanov & Lee，前者是指 Alexander Stepanov，STL 的创始人；而后者是 Meng Lee，她也是使 STL 得以推行的功臣，第一个 STL 成品就是他们合作完成的。

1.2 追根溯源：STL 的历史

在结识新朋友的时候，大多数人总是忍不住想了解对方的过去。本节将带您简单回顾一下 STL 的过去。

被誉为 STL 之父的 Alexander Stepanov，出生于苏联莫斯科，早在 20 世纪 70 年代后半期，他便已经开始考虑，在保证效率的前提下，将算法从诸多具体应用之中抽象出来的可能性，这便是后来泛型化思想的雏形。为了验证自己的思想，他和纽约州立大学教授 Deepak Kapur，伦塞里尔技术学院教授 David Musser 共同开发了一种叫做 Tecton 的语言。尽管这次尝试最终没有取得实用性的成果，但却给了 Stepanov 很大的启示。

在随后的几年中，他又和 David Musser 等人先后用 Schema 语言（一种 Lisp 语言的变种）和 Ada 语言建立了一些大型程序库。这其间，Alexander Stepanov 开始意识到，在当时的面向对象程序设计思想中所存在的一些问题，比如抽象数据类型概念所存在的缺陷。Stepanov 希望通过对软件领域中各组成部分的分类，逐渐形成一种软件设计的概念性框架。

1987 年左右，在贝尔实验室工作的 Alexander Stepanov 开始首次采用 C++语言进行泛型软件库的研究。但遗憾的是，当时的 C++语言还没有引入模板（template）的语法，现在我们可以清楚的看到，模板概念之于 STL 实现，是何等重要。是时使然，采用继承机制是别无选择的。尽管如此，Stepanov 还是开发出了一个庞大的算法库。与此同时，在与 Andrew Koenig（前 ISO C++标准化委员会主席）和 Bjarne Stroustrup（C++语言的创始人）等顶级大师们的共事过程中，Stepanov 开始注意到 C/C++语言在实现其泛型思想方面所具有的潜在优势。就拿 C/C++中的指针而言，它的灵活与高效运用，使后来的 STL 在实现泛型化的同时更是保持了高效率。另外，在 STL 中占据极其重要地位的迭代子概念便是源自于 C/C++中原生指针（ native pointer）的抽象。

1988 年，Alexander Stepanov 开始进入惠普的 Palo Alto 实验室工作，在随后的 4 年中，他从事的是有关磁盘驱动器方面的工作。直到 1992 年，由于参加并主持了实验室主任 Bill Worley 所建立的一个有关算法的研究项目，才使他重新回到了泛型化算法的研究工作上来。项目自建立之后，参与者从最初的 8 人逐渐减少，最后只剩下两个人--Stepanove 本人和 Meng Lee。经过长时间的努力，最终，信念与汗水所换来的是一个包含有大量数据结构和算法部件的庞大运行库。这便是现在的 STL 的雏形（同时也是 STL 的一个实现版本--HP STL）。

1993 年，当时在贝尔实验室的 Andrew Koenig 看到了 Stepanove 的研究成果，很是兴奋。在他的鼓励与帮助下，Stepanove 于是年 9 月的圣何塞为 ANSI/ISO C++标准委员会做了一个相关演讲（题为"The Science of C++ Programming"），向委员们讲述了其观念。然后又于次年 3 月，在圣迭戈会议上，向委员会提交了一份建议书，以期使 STL 成为 C++标准库的一部分。尽管这一建议十分庞大，以至于降低了被通过的可能性，但由于其所包含的新思想，投票结果以压倒多数的意见认为推迟对该建议的决定。

随后，在众人的帮助之下，包括 Bjarne Stroustrup 在内，Stepanove 又对 STL 进行了改进。同时加入了一个封装内存模式信息的抽象模块，也就是现在 STL 中的 allocator，它使 STL 的大部分实现都可以独立于具体的内存模式，从而独立于具体平台。在同年夏季的滑铁卢会议上，委员们以 80%赞成，20%反对，最终通过了提案，决定将 STL 正式纳入 C++标准化进程之中，随后 STL 便被放进了会议的工作文件中。自此，STL 终于成为了 C++家族中的重要一员。

此后，随着 C++标准的不断改进，STL 也在不断地作着相应的演化。直至 1998 年，ANSI/ISO C++标准正式定案，STL 始终是 C++标准中不可或缺的一大部件。

1.3 千丝万缕的联系

在你了解了 STL 的过去之后，一些名词开始不断在你的大脑中浮现，STL、C++、C++标准函数库、泛型程序设计、面向对象程序设计……，这些概念意味着什么？他们之间的关系又是什么？如果你想了解某些细节，这里也许有你希望得到的答案。

1.3.1 STL 和 C++

没有 C++语言就没有 STL，这么说毫不为过。一般而言，STL 作为一个泛型化的数据结构和算法库，并不牵涉具体语言（当然，在 C++里，它被称为 STL）。也就是说，如果条件允许，用其他语言也可以实现之。这里所说的条件，主要是指类似于"模板"这样的语法机制。如果你没有略过前一节内容的话，应该可以看到，Alexander Stepanov 在选择 C++语言作为实现工具之前，早以采用过多种程序设计语言。但是，为什么最终还是 C++幸运的承担了这个历史性任务呢？原因不仅在于前述那个条件，还在于 C++在某些方面所表现出来的优越特性，比如：高效而灵活的指针。但是如果把 C++作为一种 OOP（Object-Oriented Programming，面向对象程序设计）语言来看待的话（事实上我们一般都是这么认为的，不是吗？），其功能强大的继承机制却没有给 STL 的实现帮上多大的忙。在 STL 的源代码里，并没有太多太复杂的继承关系。继承的思想，甚而面向对象的思想，还不足以实现类似 STL 这样的泛型库。C++只有在引入了 "模板"之后，才直接导致了 STL 的诞生。这也正是为什么，用其他比 C++更纯的面向对象语言无法实现泛型思想的一个重要原因。当然，事情总是在变化之中，像 Java 在这方面，就是一个很好的例子，jdk1.4 中已经加入了泛型的特性。

此外，STL 对于 C++的发展，尤其是模板机制，也起到了促进作用。比如：模板函数的偏特化（template function partial specialization），它被用于在特定应用场合，为一般模板函数提供一系列特殊化版本。这一特性是继 STL 被 ANSI/ISO C++标准委员会通过之后，在 Bjarne 和 Stepanov 共同商讨之下并由 Bjarne 向委员会提出建议的，最终该项建议被通过。这使得 STL 中的一些算法在处理特殊情形时可以选择非一般化的方式，从而保证了执行的效率。

1.3.2 STL 和 C++标准函数库

STL 是最新的 C++标准函数库中的一个子集，这个庞大的子集占据了整个库的大约 80%的分量。而作为在实现 STL 过程中扮演关键角色的模板则充斥了几乎整个 C++标准函数库。在这里，我们有必要看一看 C++标准函数库里包含了哪些内容，其中又有哪些是属于标准模板库（即 STL）的。

C++标准函数库为 C++程序员们提供了一个可扩展的基础性框架。我们从中可以获得极大的便利，同时也可以通过继承现有类，自己编制符合接口规范的容器、算法、迭代子等方式对之进行扩展。它大致包含了如下几个组件：

C 标准函数库，基本保持了与原有 C 语言程序库的良好兼容，尽管有些微变化。人们总会忍不住留恋过去的美好岁月，如果你曾经是一个 C 程序员，对这一点一定体会颇深。或许有一点会让你觉得奇怪，那就是在 C++标准库中存在两套 C 的函数库，一套是带有.h 扩展名的（比如<stdio.h>），而另一套则没有（比如<cstdio>）。它们确实没有太大的不同。

语言支持（language support）部分，包含了一些标准类型的定义以及其他特性的定义，这些内容，被用于标准库的其他地方或是具体的应用程序中。

诊断（diagnostics）部分，提供了用于程序诊断和报错的功能，包含了异常处理（exception handling），断言（assertions），错误代码（error number codes）三种方式。

通用工具（general utilities）部分，这部分内容为 C++标准库的其他部分提供支持，当然你也可以在自己的程序中调用相应功能。比如：动态内存管理工具，日期/时间处理工具。记住，这里的内容也已经被泛化了（即采用了模板机制）。

字符串（string）部分，用来代表和处理文本。它提供了足够丰富的功能。事实上，文本是一个 string 对象，它可以被看作是一个字符序列，字符类型可能是 char，或者 wchar_t 等等。string 可以被转换成 char*类型，这样便可以和以前所写的 C/C++代码和平共处了。因为那时侯除了 char*，没有别的。

国际化（internationalization）部分，作为 OOP 特性之一的封装机制在这里扮演着消除文化和地域差异的角色，采用 locale 和 facet 可以为程序提供众多国际化支持，包括对各种字符集的支持，日期和时间的表示，数值和货币的处理等等。毕竟，在中国和在美国，人们表示日期的习惯是不同的。

容器（containers）部分，STL 的一个重要组成部分，涵盖了许多数据结构，比如前面曾经提到的链表，还有：vector（类似于大小可动态增加的数组）、queue（队列）、stack（堆栈）……。string 也可以看作是一个容器，适用于容器的方法同样也适用于 string。现在你可以轻松的完成数据结构课程的家庭作业了。

算法（algorithms）部分，STL 的一个重要组成部分，包含了大约 70 个通用算法，用于操控各种容器，同时也可以操控内建数组。比如：find 用于在容器中查找等于某个特定值的元素，for_each 用于将某个函数应用到容器中的各个元素上，sort 用于对容器中的元素排序。所有这些操作都是在保证执行效率的前提下进行的，所以，如果在你使用了这些算法之后程序变得效率底下，首先一定不要怀疑这些算法本身，仔细检查一下程序的其他地方。

迭代器（iterators）部分，STL 的一个重要组成部分，如果没有迭代器的撮合，容器和算法便无法结合的如此完美。事实上，每个容器都有自己的迭代器，只有容器自己才知道如何访问自己的元素。它有点像指针，算法通过迭代器来定位和操控容器中的元素。

数值（numerics）部分，包含了一些数学运算功能，提供了复数运算的支持。

输入/输出（input/output）部分，就是经过模板化了的原有标准库中的 iostream 部分，它提供了对 C++程序输入输出的基本支持。在功能上保持了与原有 iostream 的兼容，

并且增加了异常处理的机制，并支持国际化（internationalization）。

总体上，在 C++标准函数库中，STL 主要包含了容器、算法、迭代器。string 也可以算做是 STL 的一部分。

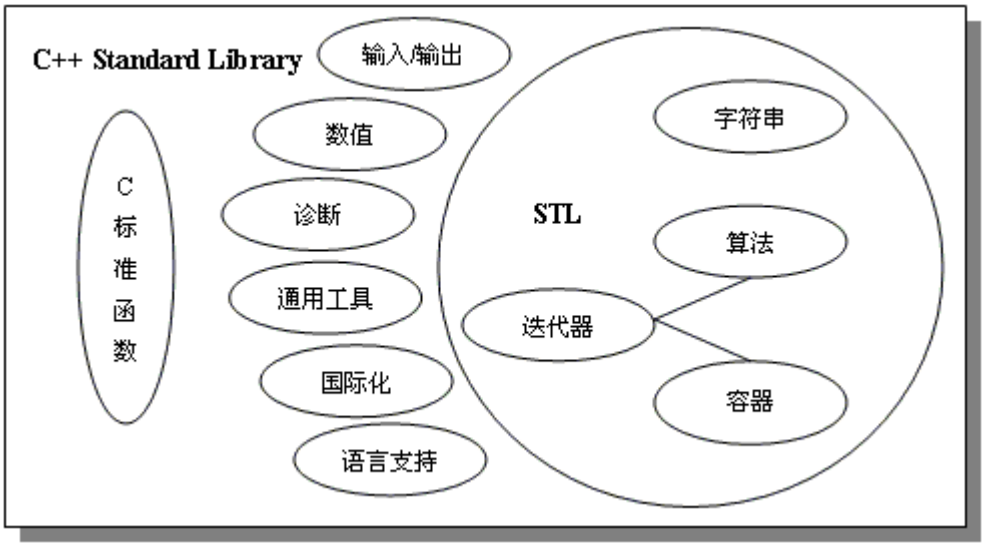

图 1：STL 和 C++标准函数库

1.3.3 STL 和 GP，GP 和 OOP

正如前面所提到的，在 STL 的背后蕴含着泛型化程序设计（GP）的思想，在这种思想里，大部分基本算法被抽象，被泛化，独立于与之对应的数据结构，用于以相同或相近的方式处理各种不同情形。这一思想和面向对象的程序设计思想（OOP）不尽相同，因为，在 OOP 中更注重的是对数据的抽象，即所谓抽象数据类型（Abstract Data Type），而算法则通常被附属于数据类型之中。几乎所有的事情都可以被看作类或者对象（即类的实例），通常，我们所看到的算法被作为成员函数（member function）包含在类（class）中，类和类则构成了错综复杂的继承体系。

尽管在象 C++这样的程序设计语言中，你还可以用全局函数来表示算法，但是在类似于 Java 这样的纯面向对象的语言中，全局函数已经被"勒令禁止"了。因此，用 Java 来模拟 GP 思想是颇为困难的。如果你对前述的 STL 历史还有印象的话，应该记得 Alexander Stepanove 也曾用基于 OOP 的语言尝试过实现 GP 思想，但是效果并不好，包括没有引入模板之前的 C++语言。站在巨人的肩膀上，我们可以得出这样的结论，在 OOP 中所体现的思想与 GP 的思想确实是相异的。C++并不是一种纯面向对象的程序设计语言，它的绝妙之处，就在于既满足了 OOP，又成全了 GP。对于后者，模板立下了汗马功劳。另外，需要指出的是，尽管 GP 和 OOP 有诸多不同，但这种不同还不至于到"水火不容"的地步。并且，在实际运用的时候，两者的结合使用往往可以使问题的解决更为有效。作为 GP 思想实例的 STL 本身便是一个很好的范例，如果没有继承，不知道 STL 会是什么样子，似乎没有人做过这样的试验。

1.4 STL 的不同实现版本

相信你对 STL 的感性认识应该有所提高了，是该做一些实际的工作了，那么我们首先来了解一下 STL 的不同实现版本。ANSI/ISO C++文件中的 STL 是一个仅被描述在纸上的标准，对于诸多 C++编译器而言，需要有各自实际的 STL，它们或多或少的实现了标准中所描述的内容，这样才能够为我们所用。之所以有不同的实现版本，则存在诸多原因，有历史的原因，也有各自编译器生产厂商的原因。以下是几个常见的 STL 实现版本。

1.4.1 HP STL

HP STL 是所有其它 STL 实现版本的根源。它是 STL 之父 Alexander Stepanov 在惠普的 Palo Alto 实验室工作时，和 Meng Lee 共同完成的，是第一个 STL 的实现版本（参见 1.2 节）。这个 STL 是开放源码的，所以它允许任何人免费使用、复制、修改、发布和销售该软件和相关文档，前提是必须在所有相关文件中加入 HP STL 的版本信息和授权信息。现在已经很少直接使用这个版本的 STL 了。

1.4.2 P.J. Plauger STL

P. J. Plauger STL 属于个人作品，由 P. J. Plauger 本人实现，是 HP STL 的一个继承版本，因此在其所有头文件中都含有 HP STL 的相关声明，同时还有 P. J. Plauger 本人的版权声明。P. J. Plauger 是标准 C 中 stdio 库的早期实现者，现在是 C/C++ User's Journal 的主编，与 Microsoft 保持着良好的关系。P. J. Plauger STL 便是被用于 Microsoft 的 Visual C++中的。在 Windows 平台下的同类版本中，其性能不错，但是 queue 组件（队列，一种容器）的效率不理想，同时由于 Visual C++对 C++语言标准的支持不是很好（至少直到 VC6.0 为止，还是如此），因此一定程度上影响了 P. J. Plauger STL 的性能。此外，该版本的源代码可读性较差，你可以在 VC 的 Include 子目录下找到所有源文件（比如：C:\Program Files\Microsoft Visual Studio\VC98\Include）。因为不是开放源码的（open source），所以这些源代码是不能修改和销售的，目前 P.J. Plauger STL 由 Dinkumware 公司提供相关服务，详情请见 http://www.dinkumware.com。据称 Visual Studio.NET 中的 Visual C++.NET（即 VC7.0），对 C++标准的支持有所提高，并且多了以哈希表（hash table）为基础而实现的 map 容器，multimap 容器和 set 容器。

1.4.3 Rouge Wave STL

Rouge Wave STL 是由 Rouge Wave 公司实现的，也是 HP STL 的一个继承版本，除了 HP STL 的相关声明之外，还有 Rouge Wave 公司的版权声明。同时，它也不是开放源码的，因此无法修改和销售。该版本被 Borland C++ Builder 所采用，你可以在 C++ Builder 的 Include 子目录下找到所有头文件（比如：C:\Program Files\Borland\Cbuilder5\Include）。尽管 Rouge Wave STL 的性能不是很好，但由于 C++ Builder 对 C++语言标准的支持还算不错，使其表现在一定程度上得以改善。此外，其源代码的可读性较好。可以从如下网站得到更详细的情况介绍： http://www.rougewave.com。遗憾的是该版本已有一段时间没有更新且不完全符合标准。因此在 Borland C++ Builder 6.0 中，它的地位被另一个 STL 的实现版本--STLport（见后）取代了。但是考虑到与以前版本的兼容，C++ Builder 6.0 还是保留了 Rouge Wave STL，只是如果你想查看它的源代码的话，需要在别的目录中才能找到（比

如：C:\Program Files\Borland\Cbuilder6\Include\oldstl）。

1.4.4 STLport

STLport 最初源于俄国人 Boris Fomitchev 的一个开发项目，主要用于将 SGI STL 的基本代码移植到其他诸如 C++Builder 或者是 Visual C++这样的主流编译器上。因为 SGI STL 属于开放源码，所以 STLport 才有权这样做。目前 STLport 的最新版本是 4.5。可以从如下网站得到更详细的情况介绍：//www.stlport.org，可以免费下载其源代码。STLport 已经被 C/C++技术委员会接受成为工业标准，且在许多平台上都支持。根据测试 STLport 的效率比 VC 中的 STL 要快。比 Rouge Wave STL 更符合标准，也更容易移植。Borland C++ Builder 已经在其 6.0 版中加入了对 STLport 的支持，它使用的 STLport 就是 4.5 版的，C++ Builder 6.0 同时还提供了 STLport 的使用说明。你可以在 C++ Builder 的 Include\Stlport 子目录下找到所有头文件（比如：C:\Program Files\Borland\Cbuilder6\Include\Stlport）。

1.4.5 SGI STL

SGI STL 是由 Silicon Graphics Computer System, Inc 公司实现的，其设计者和编写者包括 Alexander Stepanov 和 Matt Austern，同样它也是 HP STL 的一个继承版本。它属于开放源码，因此你可以修改和销售它。SGI STL 被 GCC（linux 下的 C++编译器）所采用，你可以在 GCC 的 Include 子目录下找到所有头文件（比如：C:\cygnus\cygwin-b20\include\g++\include）。由于 GCC 对 C++语言标准的支持很好，SGI STL 在 linux 平台上的性能相当出色。此外，其源代码的可读性也很好。可以从如下网站得到更详细的情况介绍：http://www.sgi.com，可以免费下载其源代码。目前的最新版本是 3.3。

2 牛刀小试：且看一个简单例程

2.1 引子

如果你是一个纯粹的实用主义者，也许一开始就可以从这里开始看起，因为此处提供了一个示例程序，它可以带给你有关使用 STL 的最直接的感受。是的，与其纸上谈兵，不如单刀直入，实际操作一番。但是，需要提醒的是，假如你在兴致昂然地细细品味本章内容的时候，能够同时结合前面章节作为佐餐，那将是再好不过的。你会发现，前面所提到的有关 STL 的那些优点，在此处得到了确切的应证。本章的后半部分，将为你演示在一些主流 C ++ 编译器上，运行上述示例程序的具体操作方法，和需要注意的事项。

2.2 例程实作

为展现 STL 的巨大魅力，我选用了一个稍稍复杂一点的例子，它的大致功能是：
从标准输入设备（一般是键盘）读入一些整型数据，然后对它们进行排序，最终将结

果输出到标准输出设备（一般是显示器屏幕）。

这是一种典型的处理方式，程序本身具备了一个系统所应该具有的几乎所有的基本特征：输入 + 处理 + 输出。

你将会看到三个不同版本的程序：

第一个是没有使用 STL 的普通 C++程序，你将会看到完成这样看似简单的事情，需要花多大的力气，而且还未必没有一点问题（真是吃力不讨好）。

第二个程序的主体部分使用了 STL 特性，此时在第一个程序中所遇到的问题就基本可以解决了。同时，你会发现采用了 STL 之后，程序变得简洁明快，清晰易读。

第三个程序则将 STL 的功能发挥到了及至，你可以看到程序里几乎每一行代码都是和 STL 相关的。这样的机会并不总是随处可见的，它展现了 STL 中的几乎所有的基本组成部分，尽管这看起来似乎有点过分了。

有几点是需要说明的：

这个例程的目的，在于向你演示如何在 C++程序中使用 STL，同时希望通过实践，证明 STL 所带给你的确确实实的好处。程序中用到的一些 STL 基本组件，比如：vector（一种容器）、sort（一种排序算法），你只需要有一个大致的概念就可以了，这并不影响阅读代码和理解程序的含义。

很多人对 GUI（图形用户界面）的运行方式很感兴趣，这也难怪，漂亮的界面总是会令人赏心悦目的。但是很可惜，在这里没有加入这些功能。这很容易解释，对于所提供的这个简单示例程序而言，加入 GUI 特性，是有点本末倒置的。这将会使程序的代码量骤然间急剧膨胀，而真正可以说明问题的核心部分确被淹没在诸多无关紧要的代码中间（你需要花去极大的精力来处理键盘或者鼠标的消息响应这些繁琐而又较为规范的事情）。即使你有像 Borland C++ Builder 这样的基于 IDE（集成化开发环境）的工具，界面的处理变得较为简单了（框架代码是自动生成的）。请注意，我们这里所谈及的是属于 C++标准的一部分（STL 的第一个字母说明了这一点），它不涉及具体的某个开发工具，它是几乎在任何 C++编译器上都能编译通过的代码。毕竟，在 Microsoft Visual C++和 Borland C++ Builder 里，有关 GUI 的处理代码是不一样的。如果你想了解这些 GUI 的细节，这里恐怕没有你希望得到的答案，你可以寻找其它相关书籍。

2.2.1 第一版：史前时代--转木取火

在 STL 还没有降生的"黑暗时代"，C++程序员要完成前面所提到的那些功能，需要做很多事情（不过这比起 C 程序来，似乎好一点），程序大致是如下这个样子的：

```
// name:example2_1.cpp
// alias:Rubish

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

int compare(const void *arg1, const void *arg2);

void main(void)
{
const int max_size = 10; // 数组允许元素的最大个数
```

```
int num[max_size]; // 整型数组

// 从标准输入设备读入整数，同时累计输入个数，
// 直到输入的是非整型数据为止
int n;
for (n = 0; cin >> num[n]; n ++);

// C 标准库中的快速排序（quick-sort）函数
qsort(num, n, sizeof(int), compare);

// 将排序结果输出到标准输出设备
for (int i = 0; i < n; i ++)
cout << num[i] << "\n";
}

// 比较两个数的大小，
// 如果*(int *)arg1 比*(int *)arg2 小，则返回-1
// 如果*(int *)arg1 比*(int *)arg2 大，则返回 1
// 如果*(int *)arg1 等于*(int *)arg2，则返回 0
int compare(const void *arg1, const void *arg2)
{
return (*(int *)arg1 < *(int *)arg2) ? -1 :
(*(int *)arg1 > *(int *)arg2) ? 1 : 0;
}
```

这是一个和 STL 没有丝毫关系的传统风格的 C++程序。因为程序的注释已经很详尽了，所以不需要我再做更多的解释。总的说来，这个程序看起来并不十分复杂（本来就没有太多功能）。只是，那个 compare 函数，看起来有点费劲。指向它的函数指针被作为最后一个实参传入 qsort 函数，qsort 是 C 程序库 stdlib.h 中的一个函数。以下是 qsort 的函数原型：

void qsort(void *base, size_t num, size_t width, int (__cdecl *compare)(const void *elem1, const void *elem2)); 看起来有点令人作呕，尤其是最后一个参数。大概的意思是，第一个参数指明了要排序的数组（比如：程序中的 num），第二个参数给出了数组的大小（qsort 没有足够的智力预知你传给它的数组的实际大小），第三个参数给出了数组中每个元素以字节为单位的大小。最后那个长长的家伙，给出了排序时比较元素的方式（还是因为 qsort 的智商问题）。

以下是某次运行的结果：

输入：0 9 2 1 5

输出：0 1 2 5 9 有一个问题，这个程序并不像看起来那么健壮（Robust）。如果我们输入的数字个数超过 max_size 所规定的上限，就会出现数组越界问题。如果你在 Visual C++的 IDE 环境下以控制台方式运行这个程序时，会弹出非法内存访问的错误对话框。这个问题很严重，严重到足以使你开始重新审视这个程序的代码。为了弥补程序中的这一缺陷。我们不得不考虑采用如下三种方案中的一种：

采用大容量的静态数组分配。

限定输入的数据个数。

采用动态内存分配。

第一种方案比较简单，你所做的只是将 max_size 改大一点，比如：1000 或者 10000。但是，严格讲这并不能最终解决问题，隐患仍然存在。假如有人足够耐心，还是可以使你的这个经过纠正后的程序崩溃的。此外，分配一个大数组，通常是在浪费空间，因为大多数情况下，数组中的一部分空间并没有被利用。

再来看看第二种方案，通过在第一个 for 循环中加入一个限定条件，可以使问题得到解决。比如：for (int n = 0; cin >> num[n] && n < max_size; n ++); 但是这个方案同样不甚理想，尽管不会使程序崩溃，但失去了灵活性，你无法输入更多的数。

看来只有选择第三种方案了。是的，你可以利用指针，以及动态内存分配妥善的解决上述问题，并且使程序具有良好的灵活性。这需要用到 new，delete 操作符，或者古老的 malloc()， realloc()和 free()函数。但是为此，你将牺牲程序的简洁性，使程序代码陡增，代码的处理逻辑也不再像原先看起来那么清晰了。一个 compare 函数或许就已经令你不耐烦了,更何况要实现这些复杂的处理机制呢？很难保证你不会在处理这个问题的时候出错，很多程序的 bug 往往就是这样产生的。同时，你还应该感谢 stdlib.h，它为你提供了 qsort 函数，否则，你还需要自己实现排序算法。如果你用的是冒泡法排序，那效率就不会很理想。……，问题真是越来越让人头疼了！

关于第一个程序的讨论就到此为止，如果你对第三种方案感兴趣的话，可以尝试着自己编写一个程序，作为思考题。这里就不准备再浪费笔墨去实现这样一个让人不甚愉快的程序了。

2.2.2 第二版：工业时代--组件化大生产

我们应该庆幸自己所生活的年代。工业时代，科技的发展所带来的巨大便利已经影响到了我们生活中的每个细节。如果你还在以原始人类的方式生活着，那我真该怀疑你是否属于某个生活在非洲或者南美丛林里的原始部落中的一员了，难道是玛雅文明又重现了？

STL 便是这个时代的产物，正如其他科技成果一样，C++程序员也应该努力使自己适应并充分利用这个"高科技成果"。让我们重新审视第一版的那个破烂不堪的程序。试着使用一下 STL，看看效果如何。

```
// name:example2_2.cpp
// alias:The first STL program

#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

void main(void)
{
vector<int> num; // STL 中的 vector 容器
int element;

// 从标准输入设备读入整数，
// 直到输入的是非整型数据为止
while (cin >> element)
```

```
num.push_back(element);

// STL 中的排序算法
sort(num.begin(), num.end());

// 将排序结果输出到标准输出设备
for (int i = 0; i < num.size(); i ++)
cout << num[i] << "\n";
}
```

这个程序的主要部分改用了 STL 的部件，看起来要比第一个程序简洁一点，你已经找不到那个讨厌的 compare 函数了。它真的能很好的运行吗？你可以试试，因为程序的运行结果和前面的大致差不多，所以在此略去。我可以向你保证，这个程序是足够健壮的。不过，可能你还没有完全看明白程序的代码，所以我需要为你解释一下。毕竟，这个戏法变得太快了，较之第一个程序，一眨眼的功夫，那些老的 C++程序员所熟悉的代码都不见了，取而代之的是一些新鲜玩意儿。

程序的前三行是包含的头文件，它们提供了程序所要用到的所有 C++特性（包括输入输出处理，STL 中的容器和算法）。不必在意那个.h，并不是我的疏忽，程序保证可以编译通过，只要你的 C++编译器支持标准 C++规范的相关部分。你只需要把它们看作是一些普通的 C++头文件就可以了。事实上，也正是如此，如果你对这个变化细节感兴趣的化，可以留意一下你身旁的佐餐。

同样可以忽略第四行的存在。加入那个声明只是为了表明程序引用到了 std 这个标准名字空间（namespace），因为 STL 中的那些玩意儿全都包含在那里面。只有通过这行声明，编译器才能允许你使用那些有趣的特性。

程序中用到了 vector，它是 STL 中的一个标准容器，可以用来存放一些元素。你可以把 vector 理解为 int [?]，一个整型的数组。之所以大小未知是因为，vector 是一个可以动态调整大小的容器，当容器已满时，如果再放入元素则 vector 会悄悄扩大自己的容量。push_back 是 vector 容器的一个类属成员函数，用来在容器尾端插入一个元素。main 函数中第一个 while 循环做的事情就是不断向 vector 容器尾端插入整型数据，同时自动维护容器空间的大小。

sort 是 STL 中的标准算法，用来对容器中的元素进行排序。它需要两个参数用来决定容器中哪个范围内的元素可以用来排序。这里用到了 vector 的另两个类属成员函数。begin()用以指向 vector 的首端，而 end()则指向 vector 的末端。这里有两个问题，begin()和 end()的返回值是什么？这涉及到 STL 的另一个重要部件--迭代器（Iterator），不过这里并不需要对它做详细了解。你只需要把它当作是一个指针就可以了，一个指向整型数据的指针。相应的 sort 函数声明也可以看作是 void sort(int* first, int* last)，尽管这实际上很不精确。另一个问题是和 end()函数有关，尽管前面说它的返回值指向 vector 的末端，但这种说法不能算正确。事实上，它的返回值所指向的是 vector 中最末端元素的后面一个位置，即所谓 pass-the-end value。这听起来有点费解，不过不必在意，这里只是稍带一提。总的来说，sort 函数所做的事情是对那个准整型数组中的元素进行排序，一如第一个程序中的那个 qsort，不过比起 qsort 来，sort 似乎要简单了许多。

程序的最后是输出部分，在这里 vector 完全可以以假乱真了，它所提供的对元素的访问方式简直和普通的 C++内建数组一模一样。那个 size 函数用来返回 vector 中的元素个数，就相当于第一个程序中的变量 n。这两行代码直观的不用我再多解释了。

我想我的耐心讲解应该可以使你大致看懂上面的程序了，事实上 STL 的运用使程序的

逻辑更加清晰，使代码更易于阅读。试问，有谁会不明白 begin、 end、size 这样的字眼所表达的含义呢（除非他不懂英语）？试着运行一下，看看效果。再试着多输入几个数，看看是否会发生数组越界现象。实践证明，程序运行良好。是的，由于 vector 容器自行维护了自身的大小，C++程序员就不用操心动态内存分配了，指针的错误使用毕竟会带来很多麻烦，同时程序也会变得冗长无比。这正是前面第三种方案的缺点所在。

再仔细审视一下你的第一个 STL 版的 C++程序，回顾一下第一章所提到的那些有关 STL 的优点：易于使用，具有工业强度……，再比较一下第一版的程序，我想你应该有所体会了吧！

2.2.3 第三版：唯美主义的杰作

事态的发展有时候总会趋向极端，这在那些唯美主义者当中犹是如此。首先声明，我并不是一个唯美主义者，提供第二版程序的改进版，完全是为了让你更深刻的感受到 STL 的魅力所在。在看完第三版之后，你会强烈感受到这一点。或许你也会变成一个唯美主义者了，至少在 STL 方面。这应该不是我的错，因为决定权在你手里。下面我们来看看这个绝版的 C++程序。

```
// name:example2_3.cpp
// alias:aesthetic version

#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>

using namespace std;

void main(void)
{
typedef vector<int> int_vector;
typedef istream_iterator<int> istream_itr;
typedef ostream_iterator<int> ostream_itr;
typedef back_insert_iterator< int_vector > back_ins_itr;

// STL 中的 vector 容器
int_vector num;

// 从标准输入设备读入整数，
// 直到输入的是非整型数据为止
copy(istream_itr(cin), istream_itr(), back_ins_itr(num));

// STL 中的排序算法
sort(num.begin(), num.end());

// 将排序结果输出到标准输出设备
```

```
copy(num.begin(), num.end(), ostream_itr(cout, "\n"));
}
```

在这个程序里几乎每行代码都是和 STL 有关的（除了 main 和那对花括号，当然还有注释），并且它包含了 STL 中几乎所有的各大部件（容器 container，迭代器 iterator，算法 algorithm，适配器 adaptor），唯一的遗憾是少了函数对象（functor）的身影。

还记得开头提到的一个典型系统所具有的基本特征吗？--输入+处理+输出。所有这些功能，在上面的程序里，仅仅是通过三行语句来实现的，其中每一行语句对应一种操作。对于数据的操作被高度的抽象化了，而算法和容器之间的组合，就像搭积木一样轻松自如，系统的耦合度被降到了极低点。这就是闪耀着泛型之光的 STL 的伟大力量。如此简洁，如此巧妙，如此神奇！就像魔术一般，以至于再一次让你摸不着头脑。怎么实现的？为什么在看第二版程序的时候如此清晰的你，又坠入了五里雾中（窃喜）。

请留意此处的标题（唯美主义的杰作），在实际环境中，你未必要做到这样完美。毕竟美好愿望的破灭，在生活中时常会发生。过于理想化，并不是一件好事，至少我是这么认为的。正如前面提到的，这个程序只是为了展示 STL 的独特魅力，你不得不为它的出色表现所折服，也许只有深谙 STL 之道的人才会想出这样的玩意儿来。如果你只是一般性的使用 STL，做到第二版这样的程度也就可以了。

实在是因为这个程序太过"简单"，以至于我无法肯定，在你还没有完全掌握 STL 之前，通过我的讲解，是否能够领会这区区三行代码，我将尽我的最大努力。

前面提到的迭代器可以对容器内的任意元素进行定位和访问。在 STL 里，这种特性被加以推广了。一个 cin 代表了来自输入设备的一段数据流，从概念上讲它对数据流的访问功能类似于一般意义上的迭代器，但是 C++中的 cin 在很多地方操作起来并不像是一个迭代器，原因就在于其接口和迭代器的接口不一致（比如：不能对 cin 进行++运算，也不能对之进行取值运算--即*运算）。为了解决这个矛盾，就需要引入适配器的概念。istream_iterator 便是一个适配器，它将 cin 进行包装，使之看起来像是一个普通的迭代器，这样我们就可以将之作为实参传给一些算法了（比如这里的 copy 算法）。因为算法只认得迭代器，而不会接受 cin。对于上面程序中的第一个 copy 函数而言，其第一个参数展开后的形式是：istream_iterator(cin)，其第二个参数展开后的形式是：istream_iterator()（如果你对 typedef 的语法不清楚，可以参考有关的 c++语言书籍）。其效果是产生两个迭代器的临时对象，前一个指向整型输入数据流的开始，后一个则指向"pass-the-end value"。这个函数的作用就是将整型输入数据流从头至尾逐一"拷贝"到 vector 这个准整型数组里，第一个迭代器从开始位置每次累进，最后到达第二个迭代器所指向的位置。或许你要问，如果那个 copy 函数的行为真如我所说的那样，为什么不写成如下这个样子呢？

```
copy(istream_iterator<int>(cin), istream_iterator<int>(), num.begin());
```

你确实可以这么做，但是有一个小小的麻烦。还记得第一版程序里的那个数组越界问题吗？如果你这么写的话，就会遇到类似的麻烦。原因在于 copy 函数在" 拷贝"数据的时候，如果输入的数据个数超过了 vector 容器的范围时，数据将会拷贝到容器的外面。此时，容器不会自动增长容量，因为这只是简单地拷贝，并不是从末端插入。为了解决这个问题，另一个适配器 back_insert_iterator 登场了，它的作用就是引导 copy 算法每次在容器末端插入一个数据。程序中的那个 back_ins_itr(num)展开后就是：back_insert_iterator(num)，其效果是生成一个这样的迭待器对象。终于将讲完了三分之一（真不容易！），好在第二句和前一版程序没有差别，这里就略过了。至于第三句，ostream_itr(cout, "\n")展开后的形式是：ostream_iterator(cout, "\n")，其效果是产生一个处理输出数据流的迭待器对象，其位置指向数据流的起始处，并且以"\n"作为分割符。第二个 copy 函数将会从头至尾将 vector 中的内容"拷贝"到输出设备，第一个参数所代表的迭代器将会从开始位置每次累进，

最后到达第二个参数所代表的迭代器所指向的位置。

这就是全部的内容。

2.3 历史的评价

历史的车轮总是滚滚向前的，工业时代的文明较之史前时代，当然是先进并且发达的。回顾那两个时代的 C++程序，你会真切的感受到这种差别。简洁易用，具有工业强度，较好的可移植性，高效率，加之第三个令人目眩的绝版程序所体现出来的高度抽象性，高度灵活性和组件化特性，使你对 STL 背后所蕴含的泛型化思想都有了些微的感受。

真幸运，你可以横跨两个时代，有机会目睹这种"文明"的差异。同时，这也应该使你越加坚定信念，使自己顺应时代的潮流。

2.4 如何运行

在你还没有真正开始运行前面后两个程序之前，最好先浏览一下本节。这里简单介绍了在特定编译器环境下运行 STL 程序的一些细节，并提供了一些可能遇到的问题的解决办法。

此处，我选用了目前在 Windows 平台下较为常见的 Microsoft Visual C++ 6.0 和 Borland C++ Builder 6.0 作为例子。尽管 Visual C++ 6.0 对最新的 ANSI/ISO C++标准支持的并不是很好。不过据称 Visual C++ .NET（也就是 VC7.0）在这方面的性能有所改善。

你可以选用多种方式运行前面的程序，比如在 Visual C++下，你可以直接在 DOS 命令行状态下编译运行，也可以在 VC 的 IDE 下采用控制台应用程序（Console Application）的方式运行。对于 C++ Builder，情况也类似。

对于 Visual C++而言，如果是在 DOS 命令行状态下，你首先需要找到它的编译器。假定你的 Visual C++装在 C:\Program Files\Microsoft Visual Studio\VC98 下面，则其编译器所在路径应该是 C:\Program Files\Microsoft Visual Studio\VC98\Bin，在那里你可以找到 cl.exe 文件。编译时请加上/GX 和/MT 参数。如果一切正常，结果就会产生一个可执行文件。如下所示：

```
cl /GX /MT example2_2.cpp
```

前一个参数用于告知编译器允许异常处理（Exception Handling）。在 P. J. Plauger STL 中的很多地方使用了异常处理机制（即 try…throw…catch 语法），所以应该加上这个参数，否则会有如下警告信息：

```
warning C4530: C++ exception handler used, but unwind semantics are not enabled.
```

后一个参数则用于使程序支持多线程，它需要在链接时使用 LIBCMT.LIB 库文件。不过 P. J. Plauger STL 并不是线程安全的（thread safety）。如果你是在 VC 环境下使用像 STLport 这样的 STL 实现版本，则需要加上这个参数，因为 STLport 是线程安全的。

如果在 IDE 环境下，可以在新建工程的时候选择控制台应用程序。

图 3：在 Visual C++ IDE 环境下运行 STL 程序

至于那些参数的设置,则可以通过在 Project 功能菜单项中的 Settings 功能【Alt+F7】中设置编译选项来完成。

;

图 4：在 Visual C++ IDE 环境下设置编译参数

有时，在 IDE 环境下编译 STL 程序时，可能会出现如下警告信息（前面那几个示例程序不会出现这种情况）：

warning C4786: '……' : identifier was truncated to '255' characters in the debug information

这是因为编译器在 Debug 状态下编译时，把程序中所出现的标识符长度限制在了 255 个字符范围内。如果超过最大长度，这些标识符就无法在调试阶段查看和计算了。而在 STL 程序中大量的用到了模板函数和模板类，编译器在实例化这些内容时，展开之后所产生的标识符往往很长（没准会有一千多个字符!）。如果你想认识一下这个 warning 的话，很简单，在程序里加上如下一行代码：

vector<string> string_array; // 类似于字符串数组变量

对于这样的 warning，当然可以置之不理，不过也是有解决办法的。 你可以在文件开头加入下面这一行：#pragma warning(disable: 4786)。它强制编译器忽略这个警告信息，这种做法虽然有点粗鲁，但是很有效。

至于 C++ Builder，其 DOS 命令行状态下的运行方式是这样的。假如你的 C++ Builder 装在 C:\Program Files\Borland\CBuilder6。则其编译器所在路径应该是 C:\Program Files\ Borland\CBuilder6\Bin，在那里你可以找到 bcc32.exe 文件，输入如下命令，即大功告成了：

bcc32 example2_2.cpp

至于 IDE 环境下，则可以在新建应用程序的时候，选择控制台向导（Console Wizard）。

图 5：在 C++ Builder IDE 环境下运行 STL 程序

现在你可以在你的机器上运行前面的示例程序了。不过，请恕我多嘴，有些细节不得不提请你注意。小心编译器给你留下的陷阱。比如前面第三个程序中有如下这一行代码：

typedef back_insert_iterator< int_vector > back_ins_itr;

请留意">"前面的空格，最好不要省去。如果你吝惜这点空格所占用的磁盘空间的话，那就太不划算了。其原因还是在于 C++编译器本身的缺陷。上述代码，相当于如下代码（编译器做的也正是这样的翻译工作）：

typedef back_insert_iterator< vector<int> > back_ins_itr;

如果你没有加空格的话，编译器会把">>"误认为是单一标识（看起来很像那个数据流输入操作符">>"）。为了回避这个难题， C++要求使用者必须在两个右尖括号之间插入空格。所以，你最好还是老老实实照我的话做，以避免不必要的麻烦。不过有趣的是，对于上述那行展开前的代码，在 Visual C++里即使你没有加空格，编译器也不会报错。而同样的代码在 C++ Builder 中没有那么幸运了。不过，最好还是不要心存侥幸，如果你采用展开后的书写方式，则两个编译器都不会给你留情面了。