本文是Bjarne Stroustrup的《**Evolving a language in and for the real world: C++ 1991-2006**》中文翻译，文中有许多术语翻译不当，有许多地方翻译得相当粗糙，请大家指正！

翻译：overcomeunicom@hotmail.com

版本：beta 1

立足现实 与时俱进:C++ 1991-2006

摘要

本文概述C++程序语言的历史，从早期的ISO标准化（1991）到1998年成为ISO标准，到稍后的C++0x标准修正版本（2006）。重点在于阐述理想、约束、编程技术和那些塑造语言的人们，而不是语言特征的细节方面。其中主要的主题有泛型编程和STL（C++标准库的算法和容器）。特定的主题包括模板分离编译、异常处理和对嵌入式系统编程的支持。在大部分时期内，C++是一个有着百万级用户的成熟的语言。因此，本文讨论各种C++的使用、技术以及商业的压力，这些都是C++持续演化的背景。

分类和主题描述 K.2[计算历史]:系统

普通术语 设计，编程语言，历史

关键字 C++，语言使用，演化，库，标准化，ISO，STL，多范例编程

1、 导言

在1991年10月，估计的C++用户的数字是400,000[121]；而到了2004年10月，相应的数字是3,270,000[61]。在90年代初C++用户不再呈指数增长但在以后的10年中，C++用户稳定增长。那段时间的工作主要有以下几个方面：

1、使用语言（显然）；

2、提供更好的编译器、工具和库；

3、避免语言分裂成各种方言；

4、避免语言和它的社团停滞不前；

显然，C++社团在第一项工作上花费了大量的时间、金钱；ISO C++委员会则倾向于把精力放在第二和第三项工作上；我的主要工作是放在第三和第四项。ISO标准委员会是那些想要改进C++语言和标准库的人们的中心。通过这样的改变，他们（我们）希望能够改进在现实世界中C++编程的艺术地位。C++标准委员会的工作主要放在C++的演化上。

认为C++是用于应用程序开发的平台，许多人想知道为什么—在C++成功之初—C++为什么没有抛弃它从C继承到的东西，走向“食物链的顶层”，成为一个“真正的面向对象”的带“完整”标准库的应用程序编程语言。任何向那个方向转变的趋势都被系统编程、与C的兼容性、同标准委员会有关联的许多社团中使用的C++早期版本兼容所压制。同时，在社团中永远不会有充足的资源用于大工程。另一个重要因素就是由主要软件、硬件提供商带来的朝气蓬勃的商业投机主义，他们看到了C++创建的应用程序是用来区别出他们和他们的竞争对手并且锁定他们用户的机会。其它玩家也看到了这一点，但他们的目标是成长和繁荣。不管怎么样，在委员会所选择的活动领域内并不缺乏支持，提供商的系统、高要求的应用程序依赖于C++。因此，他们提供稳定和最有价值的支持用于标准化工作，通过主持会议的形式，并且—更重要的是—关键的技术人员的参与。

不管好坏，ISO C++[66]仍是一个通用的编程语言并且偏向系统编程，有以下特点：

1、更好的C；

2、支持数据抽象；

3、支持面向对象编程；

4、支持泛型编程；

从历史的观点对前三项的解释可以[120，120]中找到；对支持泛型编程的解释是本文的重点主题。把泛型编程引进主流很可能是C++在这个时期对软件开发社区的最大贡献。

本文的以散乱的年表为次序进行组织：

§2 背景：1979 -1991的C++—早期的历史，设计标准，语言特征
§3 1991年的C++世界— C++标准化进程，年表
§4 标准库设施 1991-1998 — C++标准库，重点强调它的最重要和最具创新的组件：STL（容器、算法和迭代器）
§5语言特征 1991-1998 —集中在分离模板编译、异常安全、运行时类型信息和名字空间
§6 标准维护 1997-2003 —为了稳定性，C++标准没有增加新的内容；然而，委员会并没有因此而闲着
§7 C++在真实世界中的使用—应用程序领域；应用编程vs系统编程；编程风格；库、应用程序二进制接口（ABI）和环境；工具和研究；Java、C#和C；方言
§8 C++0x—目标、约束、语言特征及库设施
§9 回顾—影响和冲击；展望C++

重点在于早期和后期：早期塑造了现在的C++（C++98）；后期则反映了对C++98的经历的反应也适用于C++0x。离开代码讨论怎么在代码中表达想法不可能的，因此，代码的例子用于说明关键的想法、技术和语言设施。这些例子的解释也能被那些不是C++程序员的人们所理解。然而，介绍的焦点是塑造C++的人们、想法、理想、技术和约束，而不是语言-技术细节。对于描述今天的ISO C++是什么样的，请参考[66，126]。本文的重点在于回答这些直观的问题：发生了什么事？什么时候发生的？当时谁在场？他们的理由是什么？哪些已经完成（或未完成）？

C++是一个活的语言。本文的主要目标是描述它的演化。然而，它也是一个强调向后兼容的语言。在本文中我描述的代码在今天仍能编译和运行。因此，我倾向于使用现在时态去描述它。考虑到对兼容性的强调，这些代码可能在15年后仍然可以运行。因而，我使用现在时态强调C++演化的一个重要方面。

2、 背景： C++ 1979-1991

C++的早期历史是从我的HOPL-II论文到1991年，在我的《C++程序设计和演化》中指的前15年。它讲述在1994年之前的C++前历史。然而，为设置C++下一个年代的场景，这里简单总结一下C++的早期历史。

C++的设计是为了提供类似于Simula的设施（用于程序组织）和C的高效和灵活性（系统编程）。当时是希望是在有了这些想法的半年之内就将它实现并交付实际的工程使用，它成功了！在那时，我意识到谨慎或荒谬都不是目标，目标是适度，卷入革新和不合理，在时间标度和严峻的效率和灵活性需求。如果早期没有引入适当的革新，那么效率和灵活性必须维护并付出代价；C++的目标进一步被精炼、详细描述并且更加明确，今天所使用的C++直接反映了它的原来目标。

从1979年开始，当我还在Bell实验室的计算机科学研究中心时，我首先设计了一种C方言，称之为“带类的C”。从1979年到1983年，这项工作和与带类的C的经历决定了C++的轮廓。另一个方面，带类的C是基于我在剑桥大学攻读博士生的学习中（分布式系统）使用BCPL和Simula的经验。为了攻克系统编程任务，我觉得需要一种工具，它具有如下属性：

• 好的工具有Simula用于程序组织的组织—类、某种形式的类层次、某种形式的并发支持，和基于类的类型系统的编译期检查。我把这看成是对创造编程进程的一种支持，换句话说，是对设计的支持而不是对编译器的支持。

• 好的工具产生的程序跟BCPL一样快并且具有BCPL的将多个分离的已编译单元组合成一个程序能力。简单的链接规定是必不可少的，用于那些用多种语言写的单元，比如C、Algol68、Fortran、BCPL、汇编等，组合成一个程序，从而不会遭遇单一语言中的内在限制的阻碍。

• 好的工具应该允许高可移植性的编译器。我的经验就是“好”的编译器通常不会出现直到“下一年”并且是在一种我负担不起的机器上才能使用。这就意味着工具必须有多种编译器来源（没有专利会是对于使用“不常见”机器的用户和没有钱的研究生的一个足够的响应），并且不需要

移植复杂的运行时支持系统，在工具和宿主操作系统之间只有非常有限的结合。
我在贝尔实验室的早些年间，这些理想变成了一组用于C++设计的“经验法则”。

- 一般规则：
  - C++的演化必须由真实问题所驱动
  - 不涉及毫无结果地对完美的追求
  - C++在现在就必须是有用的
  - 每个特征必须有一个合理的明显的实现
  - 总是提供转变路径
  - C++是一门语言，而不是一个完整的系统
  - 对每一种已被支持的风格提供综合的支持
  - 不强迫人们使用特定的编程风格
- 设计支持的规则：
  - 支持健全的设计观点
  - 提供用于程序组织的设施
  - 用语言直接表达你的意思
  - 所有的特征是负担得起的
  - 允许有用的特征比防止每一种误用更重要
  - 支持将独立开发组件组合成软件
- 语言-技术规则：
  - 没有隐式地违反静态类型系统
  - 对用户自定义类型提供同内建类型一样好的支持
  - 局部化
  - 避免顺序依赖
  - 如果有疑问，选择更容易教学的特征
  - 句法问题（通常是以错误的方法）
  - 消灭预处理器的使用
- 底层的编程规则：
  - 使用传统的（没有提示信息）链接器
  - 除非有其它理由，否则要与C兼容
  - 在C++之下没有其它低级语言的使用空间（除了汇编程序）
  - 你不需要为你不需要的东西付出代价（0开销原则）
  - 如果有疑问，提供手动控制的方法

在D&E[121]第四章中有对这些标准的更详细的探讨。C++在1989年的release2.0发布时就严格地满足这些标准；基本的压力来自用于C++的模板设计和异常处理的机制，需要背离这些标准中的某些方面。我认为这些准则中最重要的属性是它们只与特定的编程语言特征只有松散的联系，而不是说它们强加约束于处理设计问题的解决方法之上。

在2006年，当我再次回顾这个列表时，我惊讶地发现两个设计规则（理想）没有明确的列在上面：

- 有直接将C++语言构建到硬件的映射
- 标准库是具体的且由C++实现

由于是从C走过来的并且长期工作于C++98标准（§3.1）的制定，这些标准（在那时候和更早期）看起来是这么的显然以致我通常没有去强调它们。其它语言，如Lisp、Smalltalk、Python、Ruby、Java和C#，并不享有这些理想。大多数语言提供抽象的机机制的同时还需要提供大多数有用的数据结构，如strings、list、tree、关联数组、vector、矩阵和set，如同内建设施，依赖于

其它语言用于（如汇编、C和C++）它们的编译器。对于多数的语言，只有C++提供由语言本身所实现一般、灵活、可扩展和高效的容器。从更广的程度说，这些理想来源于C。C有这两种属性，但没有定义重要的新类型所需要的抽象机制。

C++的理想—从带类的C开始—就是应该允许最佳地实现任意的数据结构和在它们之上的操作。这就限制了抽象机制的许多有用的方法[121]的设计，并且导致新的、有趣的编译器实现技术（例如，§4.1）。反过来，如果没有“直接将C++与硬件映射”这个准则，这个理想将很难达成。主要的想法（来自C）是操作符直接反映硬件的操作（算术和逻辑），对内存的访问操作也是由硬件直接提供的（指针和数组）。这两个理想的组合同时使得C++可以有效地用于嵌入式系统编程[131]（§6.1）。

2.1带类C的诞生

最终导致C++诞生的工作开始于我们企图去分析UNIX内核，并设法确定怎样才能把它分布到由局域网连接起来的一个计算机网络上。这项工作开始于1979年4月，在新泽西州Murray Hill的贝尔实验室计算机科学研究中心。有两个子问题很快就浮现出来：怎么分析由内核分布造成的网络流量；怎么模块化内核。这两个问题都要求提供一种描述方式，以便描述复杂系统的模块结构和模块间的通信模式。这正好就是我曾经决定的如果没有合适的工具就绝不去冲击的那一类问题。因此我决定根据自己在剑桥形成的一套标准去开发一种合适的工具。

在1979年10月，我已经完成了预处理器的雏形版本，叫Cpre，它添加了类似于Simula的类到C中。到了1980年3月，这个预处理器被改进，并且可以支持一个真实的工程和几个实验。我的记录显示到那时止，预处理器在16个系统中得到使用。第一个关键的C++库，叫“任务系统”，支持协程风格的编程[108，115]。它对于带类C的有效性是至关重要的，由于这个预处理器而被接受的语言被称之为“带类的C”。

在4月到10月这段时期内，从思考一个工具到思考一门语言的转变出现了，但带类的C仍被认为是对C的一种扩展，用以表达模块化和并发。尽管对并发的支持和模仿Simula风格是带类C的主要目标，语言本身并不包含表达并发的原语；然而，继承（类继承）和定义能被预处理器识别的具有特殊意义的类成员函数的组合，被用以编写支持期望的并发风格的库。请注意这里的风格是复数的。我认为这是至关重要的—我现在仍持此观点—这个语言应该能够表达多种并发的概念。

因此，语言提供了一般的机制用于组织程序而不是仅支持特定的应用领域。这使得带类的C和后来的C++成为一门通用语言而不是成为带支持特定应用的扩展的C的变种。后来，提供对特定应用的支持还是对一般抽象机制的支持的选择复杂出现，每一次，答案总是改进抽象机制。

2.2 特征概览

早期的特征包括

- 类
- 派生类
- 构造函数和析构函数
- 公有/私有访问权控制
- 类型检查和隐式函数参数转换

在1981年，基于可预知的需求，又添加了几个特征：

- 内联函数
- 默认参数
- 赋值操作符重载

由于带类的C是用预处理器实现的，只有少数的C中没有的特征需要描述，C的全部能力都是用户所可以使用的。这两个方面在当时都受到称赞。特别地，让C成为其子集显著地减小了所需的支持和文档的重写工作。带类的C仍被看作是C的一种方言。而且，类被称为是“用于C语言的

一种抽象数据类型设施”[108]。直到1983年，C++提供虚函数[1983]，才提出了支持面向对象编程的说法。

关于“带类的C”和后来的C++的公共问题是“为什么使用C？比如说为什么不使用pascal去构建语言？”我对于这个问题的回答可以在[114]找到：

C显然不是最易懂的也不是最容易使用的语言，但为什么这么多人使用它呢？

- C是灵活的：几乎在所有的应用领域都可以使用C，并且几乎每一种编程技巧都可以用C。C语言没有排除特定的编程方式的内在限制。
- C是高效的：语义上C是“低级的”，即是说，C的基本概念是传统计算机的基本概念的反映。因此，编译器和程序员可以比较容易地高效利用硬件资源用于C程序。
- C是触手可及的：随便一台计算机，不管是极小的还是最大的超级计算机，都有相应的质量可以接受的C编译器存在，并且C编译器支持一个可接受的完整的、标准的C语言和库。并且还有许多可以获取到的库和支持工具，因此程序员很少需要从零开始设计一个新的系统。
- C是可移植的：一个C程序不能自动的从一台机器移植到另一台机器（操作系统），也不是说这样的移植必须容易完成。然而，通常来说，移植软件的大部分与机器相关的代码在技术和经济上都是可行的。

与这些“第一位”的优点的比较，那些“第二位”的缺点，比如古里古怪的C声明语法和缺乏一些语言构筑的安全就变得不那么重要了。

Pascal被认为是一种玩具语言[78]，往C中添加类型检查看起来比往pascal中添加那些认为对系统编程来说是必须的特征要容易和安全得多。在那个时候，我非常害怕犯由于家长式的误导或由于简单的无知从而使得语言在某些重要领域无法应用的错误。10年后的结果清晰表明，选择C作为基础使得我能在系统编程的主流中，而这正是我想要的。语言复杂性的代价也是相当可观的，但（仅仅）是受控的。今天，保持与演化中的C语言和标准库兼容是个严重的问题（参见§7.6）。

除了C和Simula，我还把Modula-2，Ada，Smalltalk，Mesa和Clue作为C++的理想的来源[111]，因此这并不缺乏灵感。

2.3 工作环境

带类的C的是由我设计和实现，作为在贝尔实验室的计算机科学研究中心的一个研究工程。中心提供了可能的独一无二的环境做这样的工作。1979年当我加入到那里时，我被告知去“做某些有趣的事”，并给适当的计算机资源，鼓励和那些有趣的和有才能的人交流，并在对我的工作进行评估之前给我了一年的时间去实现它。

那里有一种文化倾向，反对需要很多人参与的“伟大的工程”；反对“伟大的计划”，比如让其它人去实现未经考验的论文设计；反对设计者和实现者之间的级别区分。如果你喜欢这样的东西，贝尔实验室和其它组织中有许多这样地方，在那里你可以让你沉迷于这些嗜好。然而，在计算机科学研究中心，对你总有一个要求：如果你不是搞理论的，那么就亲自动手做出某种东西来具体表达你的想法，并找到一些能从你所构建的东西中受益的人。这种环境非常支持这样的工作，并且实验室提供一大群有想法和乐于挑战问题和测试任何构建的东西的人。因此我在[114]中写道：“从来就没有一篇关于C++设计的论文；设计、文档的编写和实现都是同时进行的。自然，C++的前端是用C++写的。也从来没有一个“C++工程”或“C++设计委员会”。解决使用者所遇到的问题和作者与他的朋友、同事间的讨论而产生的问题贯穿整个C++的演化过程。

2.4 从带类的C到C++

在1982年，我清楚地认识到带类的C只是“中等成功”，并将保持这样直到它消亡。带类的C的成功，我认为是一个满足了它的设计目标的简单的结果：带类的C帮助组织一大组程序，这比C要好得多。至关重要的，这是在没有运行时效率损失、也不需要接受不可接受的开发组织的文化上的改变的基础上达到的。影响它成功的因素部分是提供C所没有的一组有限的设施，部分

是用于实现带类的C的预处理器技术。带类的C不提供对那些希望花费巨大努力去获得大致相当的回报的人们的支持：带类的C是向正确方向迈出的重要的一步，但只是一小步。从这个分析得到的结果，我开始设计带类的C的更清晰和可扩展的继任者，并使用传统的编译器技术实现它。

在从带类的C到C++的转变中，只有完全理解所有语法和语义才有可能使用恰当的编译器前端对类型检查进行改进。这里提出带类的C的一个主要问题。另外还增加了

- 虚函数
- 函数名和操作符重载
- 引用
- 常量（const）
- 许多微小的设施

其中虚函数是主要的增加，因此它使得面向对象编程成为可能。我没有办法让我的同事们相信这些设施的实用性，但把它们看为是支持一种重要编程风格（“范例”）的本质。

经过许多年的使用，release2.0是主要的发行版本，它提供一组重要的扩展特征。比如：

- 类型安全链接
- 抽象类
- 多继承

大部分的扩展和精炼代表了从C++中获得的经验，并且这些特征不可能早一点添加进来，因为我没有太多的远见。

2.5 年表

早些年的年表总结如下：

1979　带类的C的工作开始；第一个带类的C投入使用
1983　第一个C++编译器投入使用
1984　C++命名
1985　Cfront 1.0发布（第一个商用发行版）；C++程序设计语言（TC++PL）[112]
1986　第一个商用Cfront PC移植（Cfront 1.1,Glockenspiel）
1987　第一个GNU C++发布
1988　第一个Oregon软件的C++发布；Zortech的第一个C++发布
1989　Cfront Release 2.0；C++注解参考手册[35]；ANSI C++委员会（J16）成立（华盛顿,D.C.）
1990　第一届ANSI X3J16技术会议（Somerset，新泽西州）；模板被接纳（西雅图，WA）；异常被接纳（Palo Alto,CA）；Borland第一个C++发布
1991　第一届ISO WG21会议（Lund，瑞典）；Cfront3.0发布（包含模板）；C++程序设计语言（第二版）[118]

C++用户的数目平均每7.5个月就翻一倍，从1979年10月的只有一个用户变成1991年10月400,000用户[121]。计算用户数目当然很困难，但在早些年，我和那些卖编译器、库、书籍等的人们有联系，所以我对这些数字相当有信心。这些数字跟IDE提供的数字[61]也是一致的。

3、 1991时的C++世界

在1991年，我的《C++程序设计语言》第二版出版，作为对1989年定义的《C++注解参考手册》（“ARM”）[35]的补充。这两本书设置了C++编译器的标准并且从某种程度上说设置了多年后的编程技术的标准。因此，1991被认为是C++为进入主流所做的准备的最后阶段。从那时起，团结就是主要问题，那一年，有5个编译器可供选择（AT&T，Borland，GNU，Oregon和Zortech），在1992年还出现了另外三个（IBM，DEC和微软）。1991年10月，AT&T的Cfront3.0发布（我原来的编译器[121]）是第一个支持模板的。DEC和IBM的编译器都支持模板和异常，但微软的不支持，因而使鼓励程序员使用更现代化的风格的工作受挫。C++标准化的工作，开始于1989年，

并在ISO的主办下正式转变成国际化的工作。然而，这跟1989年的由许多非美国人参与的组织会议并没有实质上的不同。主要的不同在于我们现在是指ISO C++而不是ANSI C++。C++程序员—还有将成为C++程序员的人们—现在可以从100多种不同目标、不同范围和不同质量的书中进行选择。

基本上，1991年对于C++社区来说是普通的、成功的一年。它跟其它年份并没有什么不同。我强调1991年的理由是因为我的HOPL-II[120]对年份的描述到1991年就停止了。

3.1 ISO C++标准进程

对于C++社团，ISO标准进程是主要的：总体上，C++社团没有其它正式的中心，没有其它论坛去驱动语言的演化，没有其它组织关心语言和关心社团。C++没有自己的公司去决定语言的轨迹，从经济上支持它的发展和提供市场营销。因此，C++标准委员会成了认真考虑语言和标准库演化的地方。从更大的范围说，从1991-2006，C++的演化是由委员会中的自愿者们所能够做的工作和避免令人畏惧的“由委员会设计”的程度所决定的。

1989年，ANSI C++（ANSI J16）成立；它现在仍在INCITS（InterNational Committee for Information Technology Standards,一个美国组织）的主办下运行。在1991年，在ISO的帮助下它成为国际化工作的一部分。其它几个国家（如法国、日本和英国）有他们自己的国家委员会主持他们自己的会议（亲自或电子化的）并派代表参加ANSI/ISO会议。直到1997年C++98标准[63]的最后投票为止，委员会每年开三次为时一周的会议。现在，它每年开两次，在会议之间更依赖于通过网络交流。委员会在北美洲和欧洲或其它地方轮流召开会议，但大部分还是在欧洲召开。

J16委员会的成员是自愿者，他们必须付钱（一年大约800美元）去获得做这些工作的荣幸。因此，大部分的成员代表公司，公司愿意支付会费和旅行开销，但总有一小部分人得为自己买单。每个参加的公司都有一票，正如每一个个体，因此一个公司无法向委员会插入它的许多员工。在ISO（WG21）中，代表自己国家的人们和加入国家C++小组的人们是否能获得报酬取决于他们国家的标准组织的规定。J16的召集人主持技术会议和正式投票。投票过程是非常民主的。首先是在工作组中进行一次或多次的“民意调查”，作为改进提议和争取达到一致性的进程的一部分。然后是在所有委员中进行更加正式的J16投票，只有授权的成员才能投票。最后（ISO）的投票是在在每个国家的基础上完成的（per-nation basis）。目标是一致性通过，一致性的定义是大多数，因此最终的标准对于每个人都很好，但不是精确到每一个成员都喜欢。对于1997/1998的最后标准投票，ANSI投票是43-0，ISO是22-0。我们真的达到了一致性，甚至是全体通过。我被告知这样明确的投票是很不寻常的。

会议倾向于变得象大学组织，但是当必须对重要和有争议的决定做出抉择时，也会有紧张的时刻。例如，为export投票做准备的辩论就使得委员会很紧张（§5.2）。共同掌权对于C++来说是十分重要的。委员会是惟一的真正的开放的论坛，在这里，实现者和使用者来自不同的—并且常常是朝气蓬勃的相互竞争的—组织能够聚在一起并交换意见。如果没有一个由专家、个人和组织组成的密集的网状的委员会，C++将会分裂成许多方言。参加会议的人数在40到120之间。显然，参加会议的人会随着会议地点中C++使用者人数（比如硅谷）或对重要事情要做出抉择（比如我们是否要采纳STL？）而增加。在1996的Santa Cruz会议上，当时有105人在那个房间参加会议。

大多数的技术工作是由个体在会议之间完成或者是由工作组完成。这些工作组官方称之为“ad hoc”，并且不是常设机构。然而，有些持续了多年并一直专注于某项工作，成为委员会制度上的回忆。主要的长命的工作组有：

- 核心—主席：Andrew Koenig，Josée Lajoie，Bill Gibbons，Mike Miller和Steve Adamczyk
- 演化(原来的扩展)—主席：Bjarne Stroustrup
- 库—主席：Mike Vilot，Beman Dawes，MattAustern和Howard Hinnant

当工作特别紧张时，这些组会分成更小的专注于特定的主题的工作组。目标是提高这个过程的并

行程度和更好地利用许多人所展现出来的技能。

整个委员会的官方的“主席”的主要工作是确保所有的正式的规则都被遵守和向SC22报告，他被称之为召集人。原来的召集人是Steve Carter（BellCore），然后是Sam Harbinson（Tartan Labs和Texas Instrucments），Tom Plum（Plum Hall）和Herb Sutter（微软）担任。

当整个委员会集合在一起的时候由J16的主席管理会议，Dmitri Lenkov（Hewlett-Packard）是第一任，Steve Clamage（Sun）是第二任。

标准文档的草稿维护是由工程编辑者完成，第一任工程编辑者是Jonathan Shopiro（AT&T）。然后1992-2004年是Andrew Koenig（AT&T），而后是Pete Becker（Dinkumware）。

委员会由许多具有不同兴趣、不同关注点和不同背景的人组成。有些代表他们自己，有些代表大公司；有些使用PC机，有些使用UNIX盒子，有些使用大型机；有些想让C++变得更象面向对象的语言（根据各种不同的对面向对象的定义），有些想让ANSI C成为C演化的终点。有些人有C的背景，有些则没有。有些有标准化工作的背景，而大部分人则没有。有些有计算机科学背景，有些则没有。大多数人是程序员，有些则不是。有些是语言学专家，有些则不是。有些是为终端用户服务，有些是工具提供者。有些喜欢参与大项目，有些则不喜欢；有些喜欢与C兼容，有些则不喜欢。很难找到一个能概括他们的东西。

背景的多样性有利于C++，只有多样化的组才能代表C++社团中的不同的利益。而最终的结果是—比如1998年标准和技术报告（§6.1，§6.2）—对每个人都是足够的好。而不是只是某一个小社团的理想的体现。然而，成员的多样性和规模为有建设性的讨论设置了困难并延缓了进程。特别地，这个非常开放的进程很容易被那些技术水平不够或不能理解或尊重他人观点的人破坏，部分上说，对提议的考虑就是对委员会成员的一种教育过程。有些成员宣称—不完全是开玩笑—他们参加讨论就是想要得到教育。我对C++使用者（C++应用程序的编程人员和设计者）的意见可能被语言专家的声音所淹没表示担忧，也可能是在语言设计者、标准委员会、编译器实现者和工具构造者等。

为获得有哪些组织出席相关的会议，下面给出1991-2005的成名名单：Apple，AT&T，Bellcore，Borland，DEC，Dinkumware，Edison Design Group (EDG)，Ericsson，Fujitsu，Hewlett-Packard，IBM，Indiana University，Los Alamos National Labs，Mentor Graphics，Microsoft，NEC，Object Design，Plum Hall，Siemens Nixdorf，Silicon Graphics，Sun Microsystems，Texas Instruments和Zortech。

改变已经被广泛使用的语言定义与从第一原则开始的简单设计是非常不同的。无论何时，当我们有一个“好的想法”时，无论或大或小，我们必须牢记：

- 已经有百万行代码存在—大部分不会重写，不管重写能够获得多大的好处
- 已经有百万的程序员存在—大部分不会花时间学习新的东西除非他们认为这是必不可少的
- 有些十年前的编译器仍在工作—许多程序员不会使用在他们所支持的平台上编译器不支持的语言特征
- 已经有数以百万的旧的教科书存在—许多在未来的五年内仍将被使用

委员会考虑了这些因素，明显地给出了一个保守的倾向。除了其它东西，委员会的成员间接地对超过100百万行的代码负责（也代表了他们的组织）。许多委员会的成员促进改变，但几乎所有人都非常小心和负责任。有些成员觉得，他们的作用是避免不必要和危险的不稳定性。对于大多数成员来说，与C++的早期版本（ARM C++[35]）兼容，与C的早期版本（K&R C[76]）兼容和公司的各种方言兼容是一个严重的事件。我们（委员会的成员们）尝试面对将来的挑战，比如并发，但我们真的牢记C++是许多工具链的基础。破坏C++，那么Java和C#的主要编译器也会被破坏。显然地，委员会不能通过不兼容性来“破坏C++”，尽管不兼容性会破坏C++。工业界会简单地忽视一个严重的带有不兼容的标准，同时开始偏离C++。

3.2 年表

从1991年向前看，我们通过列出一些重要的决定（投票）可以获得某些想法：

1993 接纳运行时类型信息（Porland,Oregon）§5.1.2；接纳名字空间（Munich ,Germany）§5.1.1

1994 string（由字符类型模板化）（圣地亚哥，加利福尼亚）

1994 STL（圣地亚哥，加利福尼亚）§4.1

1996 export（Stockholm，Sweden）§5.2

1997 对完整标准的最后一次投票（Morristown，新泽西州）

1998 ISO C++标准[63]通过

2003 技术勘误表（“中期bug修复发布”）[66]；C++0x的工作开始

2004 性能技术报告[67] §6.1；库技术报告（散列表，正则表达式，智能指针等）[68] §6.2

2005 第一次关于C++0x特征的投票（Lillehammer，挪威）；原则上接纳auto，static_assert和右值引用§8.3.2

2006 C++0x特征的第一次委员会（官方的）投票（柏林，德国）

城市名反映了会议的地点，决定是在那里做出的。他们给参与者另一种风味。当委员会在一个城市出现时，通常会有许多来自周边城市和国家的参与者。而举办者都会利用C++专家们的影响力—许多是国际有名的—安排演讲，增加对C++的理解和社区的标准化。第一个这样安排的是在1991年的Lund，当时瑞典代表团与Lund University合作举办了一个为期两天的专题研讨会。

这个列表给出了解决主要问题的过程的提示，尝试产生一个一致的语言和库，而不是一组无关的“巧妙的特征”。在任何时候，工作集中在许多弱联系的特定主题，比如“undefined”的定义，异常发生后是否有可能继续运行（不可以），string应该提供什么函数等。让委员会完全认可是极端困难的。

3.3 为什么改变？

察看列表上的决定并牢记委员会的内在的保守偏向，一个显然的问题是：“为什么要改变某些东西？”有些人认为 “标准化就是把已经存在的实践进行文档化”，这些人只是委员会成员的中的极少数，并且不能代表C++委员会的声音。尽管人们说他们不想要“改变“，只想要“一两个改进”。在这一点上，C++委员跟其它语言的标准化工作组非常象。

基本上，我们（委员会的成员们）想要改变是因为我们有乐观的观点，即更好的语言特征和更好的库会导致更好的代码。这里“更好”意味着“代码更可维护”、“代码更容易被阅读”、“能捕获更多的错误”、“程序运行更快”、“程序更小”、“更容易移植”等。人们的标准是不同的，有时甚至是戏剧性的不同。这种观点是乐观，因为有足够多的证据显示人们可以—并且做到了—在每种语言里面写非常糟糕的代码。不管怎么样，大多数程序员群体—包括C++委员会—都是由乐观主义所主宰。特别地，大部分C++委员们相信高质量的C++代码可能通过提供更好的语言特征和标准库设施得以改进。做这样的工作需要时间：在某些情况下也许我们必须等到新一代的程序员得到教育。然而，委员会从根本上是由乐观主义和理想主义驱动的—受大量经验所节制—而不是俗世的观点，只一味地给人们所要求的或提供一些“可以卖钱”的东西。

另一种观点是世界是在变化的，因此一个活的语言也是变化的—每一个委员会的委员都这么说。因此，我们能够在改进上进行工作—或者让其它人去做。当世界改变时，C++必须演化以面对新的挑战。另一个最明显的选择不会是“死亡”，而是但被公司所拥有，正如Pascal和Objective C那样，分别变成Borland和Apple的公司语言。

在1991年的Lund（瑞典）会议后，下面的警戒性的传说在C++社团中流行起来：

我们常常提醒自己关于轮船Vasa的故事，那时它是瑞典海军的骄傲，计划建造成有史以来最大和最漂亮的战舰。不幸的是，为了装载足够的雕像和大炮，它在建造过程中经历了重大的重新设计和扩展。结果是它在跨Stockholm港湾的途中，一阵风过后它就沉了，大约

有50人遇难。这艘船已经被打捞上来，你可以从Stockholm的博物馆里看到它。它看起来非常漂亮—比它第一次扩展重新设计前漂亮得多。也比那些17世纪的战舰漂亮得多—但那不能成为它的设计者、建造者和预期的使用者的安慰。

这个故事经常被提起，作为反对增加特征的预警（那就是我告诉委员会的场景）。不管怎么样，对于复杂的事物，总有另一面：如果Vasa已经完成原来的设计，当首次遇到“现代的两层甲板”的战舰，它也同样会被打得满身是洞，沉到海底。忽视世界中的改变也不是选择。

4、 标准库：1991-1998

在1991年之后，C++标准的草稿中的最大改变是标准库。尽管我们做对标准文档做了非常多的细小的改进，与之相比较，语言特征只有很少的改变。为获得这样的观点，注意到标准总共有718页：310页定义了语言，366页定义了标准库。其它的就是附录等。另外，C++标准库通过引用包含C标准库，那是另外的81页。参考手册TC++PL2[118]作为标准化的基本文档，包含154页用于语言，而仅用了一页用于标准库。

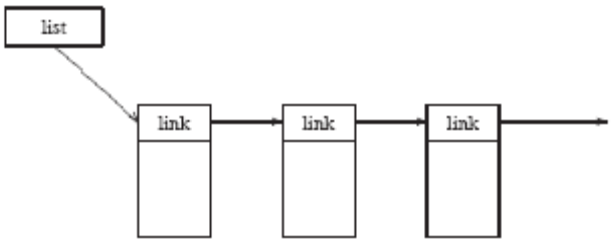
到目前为止，标准库的最具创新的组成部分是“STL”—容器、迭代器和算法。STL不仅影响并继续影响编程和设计技术，还将影响新的语言特征的方向。因此，STL在这里被看成是第一位的，而且比标准库的其它组件有更详细的阐述。

4.1 STL

STL是主要的创新并变成标准的一部分，许多关于编程技术的新的思考就是从那时起开始出现的。基本上，STL是一种革命，与以前C++社团对的容器及容器的使用的想法不同。

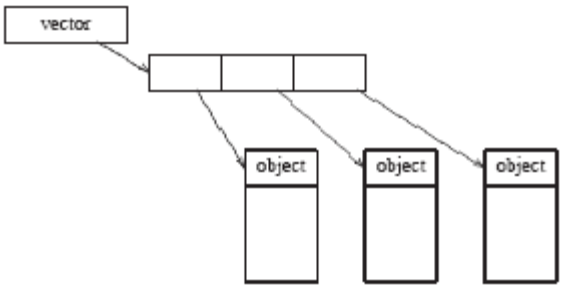
4.1.1 STL之前的容器

从Simula的最早期，就已经有了容器（例如list）：当且仅当它的类（显式或隐式）派生自特定的Link并且/或Object类时，对象能够放到一个容器中。这个类包含管理容器中的对象所需要的链接信息并为元素提供了一种公用的类型。基本上，这样的容器是存储指向对象的引用（指针）的容器。我们可以用图来表示这么的一个list：



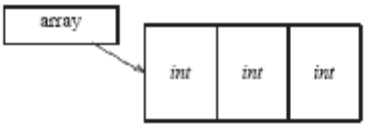
其中link来自基类Link。

类似的，一个“面向对象”的vector基本上就是一个数组，存储对象的引用。用下面的图来表示这样的一个vector：



在vector数据结构中的指向对象的引用，指向的就是Object基类的对象。这意味着基础类型的对

象，比如int和double，不能直接放到容器中，直接支持基本类型的数组类型必须与其它容器区分开：



而且，对于简单类的对象，比如complex和point，如果我们要把它们放到容器中则没有办法在时间和空间上进行优化。也即是说，Simula风格的容器是侵入式的，依赖于插入元素类型中的数据字段，并且通过指针（引用）提供对对象的间接访问。而且，Simula风格的容器不是静态类型安全的。比如，Circle可以添加到一个list中，但当从list中抽取出来时，我们只知道它是一个Object，需要使用转换（显式类型转换）去重新获得它的静态类型。

因此，Simula容器提供了对内建类型和用户自定义类型（后者只有某些可以放在容器中）的不同对待。类似的，数组与用户定义的容器也被区别对待（只有数组可能保存基础类型）。这直接违背了C++中的两条最清晰的语言-技术理想：

- 对内建类型和用户自定义类型提供相同的支持；
- 你不需要为你不需要的东西付出代价（0开销原则）

Smalltalk对容器的处理方法跟Simula一样，尽管它让基类通用，从而不需要隐式转换。后来的语言也有这个同样的问题，比如Java和C#（尽管它们—象Smalltalk—利用通用类，C#2.0还应用类似于 C++的特化去优化整数容器）。许多早期的C++库（比如，NIHCL[50]，早期的AT&T库[5]）也是根据这样的模型。它拥有许多重要的实用性，许多设计者也很熟悉它。然而，我认为这既没有规律性也没有效率（时间和空间），对于真正的通用库来说，这是不可接受的（你能从我的TC++PL3[126] §16.2中找到我的分析总结）。

没有解决这些逻辑上和性能上的问题的方法，导致在1985年的C++中没有提供一个合适的标准库（参见D&E[121] §9.2.3），这就是那个“最大的失误”的基本原因：我不想绑定任何不能直接将内建类型作为元素和不是静态类型安全的东西。甚至在第一个“带类的C”的论文[108]中，就开始跟这个问题做斗争，不成功地使用宏解决了这个问题。而且，我特别不想提供covariant container。考虑一个可以存储某些“通用”对象的vector：

```
void f(Vector& p)
{
    p[2] = new Pear;
}
void g()
{
    Vector apples(10); // 用于存放apple的容器
    for(int i=0; i<10; ++i)
        apples[i] = new Apple;
    f(apples);// 现在apple里面包含了pear
}
```

g的作者假定apples是Apple的vector，然而，vector是一个完美的普通的面向对象的容器，因此它真的包含（指针指向）对象。因为Pear也是对象，f可以毫无问题地把Pear放到容器里面去。它需要运行时的检查（隐式或显式）去捕获错误/误解。我记得当别人向我解释这个问题时（80年代早期）时我是何等地震惊，这太拙劣了。我决定我不能写这样的容器，这将带给用户肮脏的诧异和严重的运行时检查代价。在现代C++（比如，1988年后），

这个问题通过使用由元素类型进行参数化的容器得以解决。特别地，标准C++提供vector<Apple>作为解决方法。注意到vector<Apple>不能转换到vector<Fruit>，尽管Apple可以隐式转换到Fruit。

4.1.2 STL的出现

在1993年末，我发现有一种用于容器的新方法和使用，那是由Alex Stepanov所开发的。他把他所建造的库称之为“STL”。Alex那时在HP实验室工作，但他早期在贝尔实验室干过几年，在那里他和Andrew Koenig在一起，也是在那里，我和他一起讨论过库的设计和模板机制。他曾激发我更加努力地在模板机制的一般性及效率上进行工作，但幸运的是，他没有使我信服让模板更象Ada的generics。如果当时他说服了我，那么他将不会去设计和实现STL！

Alex展示了他长达十年的在泛型编程技术上的研究的最新成果，泛型编程技术的目标是基于严格的数学基础的“最一般、最有效率的代码”。它是容器和算法的框架。他首先向Andrew解释它的想法，然后Andrew用了数天的时间向我展示STL的用法。我的第一反应是迷惑，我发现STL风格的容器及其使用方法非常奇特，甚至是丑陋的、冗赘的。比如，你对vector<double>根据它们的绝对值进行排序，代码象下面这样：

```
sort(vd.begin(), vd.end(), Absolute<double>());
```

这里，vd.begin()和vd.end()指向vector中元素序列的头部和尾部，Absolute<double>()比较绝对值。

STL将算法与数据分离，那正是传统的做法，而不是面向对象的做法。而且，它还分离算法的策略决策，比如算法上的排序标准和搜索标准。其结果是无比的灵活性并且—出人意外的—高性能。

象许多熟悉面向对象编程的程序员一样，我认为我只是粗略地知道怎样使用容器进行编程。我猜想那是某些类似于Simula风格的经过模板扩充的容器，用于静态类型安全，也许还用于迭代器接口的抽象类。STL代码看起来非常不同，然而，多年后我总结出一个清单，那些我认为对于容器来说是很重要的属性都列于其上：

1、单独的容器是简单和高效的
2、容器提供它的“自然的”操作（比如，list提供put和get，vector提供下标操作）
3、简单操作，比如成员访问操作，不需要函数调用用于它们的实现
4、提供公共函数（可以通过迭代器或基类）
5、容器默认是静态类型安全的，并且是homogeneous（即在同一个容器的所有元素的类型相同）
6、heterogeneous容器能够由存储指向共同基类的指针的homogeneous容器所实现
7、容器是非侵入式的（比如，要成为容器的成员，一个对象不需要有特殊的基类或链接的字段）
8、容器能够包含内建类型的元素
9、容器能够包含有外部强加布局的structs
10、容器能应用在一般的框架上（容器和在容器上的操作）
11、没有sort成员函数的容器也能排序
12、“公共服务”（比如persistence）能独立地提供给一组容器（通过基类？）

在[124]中可以找到一个稍微不同的版本。

令我吃惊的是STL满足列表上的所有要求除了最后一个，我曾经想过使用一个公共基类为所有派生类（比如所有对象或所有容器）提供服务（比如persistence）。然而，我以前不（现在也不）认为这样的服务对容器来说是重要的。有趣地，某些“公共服务”能由“concepts”（§8.3.3）来表达，特别是在表达期望能从一组类型中得到些什么时，所以C++0x（§8）很可能将STL容器与上面列表的理想靠得更近些。

它花了我一些时间—很多周—去习惯STL。在那之后，我担心将这么一个全新风格的库引入到C++社区是否太迟。考虑到让标准委员会的成员们在标准化进程的最后阶段接受一些新的和

创新性的东西的可能性，我决定（正确的决定）让那些奇特的东西降到最低。尽管这样，标准仍延迟一年—而C++社团急切地需要标准。还有，委员会从根本上就是保守的团体而STL又太具革命性。

因此，尽管机率是很低的，但我仍带着希望辛勤工作。毕竟，我真的觉得C++没有足够大和足够好的标准库[120]（D&E [121] § 9.2.3）是非常坏的。Andrew Koenig尽他最大的可能为我鼓气，增加我的勇气，Alex Stepanov把他所知道的都教给了Andy和我，幸运的，Alex还没有完全领会到获得大多数同意的困难就通过了委员会这一关。因此他没有气馁，在技术方面不断工作同时教我和Andrew。我开始向其它人解释隐藏在STL后面的理念。比如，D&E上§15.6.3.1上的例子来自STL，我引自Alex Stepanov的原话："C++是一门威力强大的语言—我们遇到的第一个语言—允许构造具有数学的精确、优美和抽象的泛型编程组件，具有跟手工写的代码一样的高效。"这通常是对泛型编程的引述，特别是对STL。

Andrew Koenig和我邀请Alex在1993年10月的标准委员会会议上做晚会致词，在San Jose，加利福尼亚："它以The Science of C++ Programming为标题，并处理许多axioms of regular types—与构造、赋值和相等有关。我也阐述了现在称之为前向迭代器的公理（axiom）。我没有提到任何容器，而只提到一个算法：find[105]"。那次谈话是大胆的，令我吃惊和高兴的是，基本上把委员会的态度从"在这个阶段不可能做这么大的事情"转变到"好吧，让我们看看"。

那正是我们需要的突破。在接下来的4个月中，我们（Alex，他的同事Meng Lee，Andrew和我）在一起实验、争论、沟通、教学、编程、重设计和文档化，这样Alex才能在1994年3月在San Diego，加利福尼亚州向委员会呈上一份完整的STL描述。1994年Alex又安排了一次会议，面向HP中的C++库实现者。与会者有Tom Keffer（Rogue Wave），Meng Lee（HP），Nathan Myers（Rogue Wave），Larry Podmolik（Anderson Consulting），Mike Vilot，Alex和我。我们就许多原则和细节取得一致意见，但STL的大小成为主要障碍。在是否需要STL中的大部分上并没有达成一致，有个（现实的）担心就是委员会没有时间去检查和更正式地说明这么大的一个东西，而且当有许多事情需要去理解、实现、文档、教育时，人们总是感到沮丧。最后，在Alex的催促下，我拿起笔砍掉大概全文的三分之二，对于每一个设施，我对Alex和其它库专家的解释是—非常简短地—为什么它不能被砍掉和为什么它将让大多数C++程序员从中受益。这是个非常可怕的锻炼。Alex最后宣称那样做让他心碎。然而，剩余的部分现在就成了STL[103]，并且在1994年10月在Waterloo，加拿大的会议中进入了ISO C++的标准—而这是原始的、完整的STL所不能做到的。即使是修改后的"精简的STL"，仍将标准延迟了一年多。库的实现者们对库的大小和复杂性非常担心，关注提供一个优质的编译器的代价。比如，我记得Roland Hartinger（代表Simens和德国）担心接受STL将花费他的部门100万马克。现在回想，我想我造成的伤害比我们预期的要小得多。

在关于是否接纳STL的讨论中，我记得：Beman Dawes平静地向委员会解释说，他认为STL对于普通的程序员来说太复杂了，但作为练习，他已经实现了差不多10%，所以他不再认为STL超出标准的能力。Beman以前是（现在仍是）委员会中的一名少有的应用程序建造者之一。不幸的是，委员会往往被编译器制造者、库和工具的建造者所主宰。

我把STL归功于Alex Stepanov。他在STL出现之前的10年就开始工作于基本的理想和技术，不成功地使用Scheme和Ada[101]。然而，Alex总是第一个坚持参加探索的人。David Musser（Rensselaer Polytechnic Institute的一位教授）和Alex在泛型编程上一起工作了差不多20年，Meng Lee和Alex在HP一起工作过，帮助编程实现初始的STL。Alex和Andrew Koenig之间的email讨论也是有帮助的。除了砍掉STL的那部分实践，我在技术上的贡献是很少的。我建议把与内存相关的各种信息放到一个单独的对象中—这变成了allocator。我还在在Alex的黑板上草拟了最初的需求表，因此创建了标准文档上说明STL模板需求的格式。这些需求表实际上就是语言表达能力不足的指示器—这样的需求应该是代码的一部分。参见§8.3.3

Alex命名他的容器、迭代器和算法为STL库，通常这是“Standard Template Library”的缩写。然而，库在有这名字之前就已经存在，并且标准库的许多部分依赖于模板。机智的人建议“STepanov and Lee”作为另一种解释。Alex最后的话是：“STL就是STL。”正如其它情况，缩写究其一生代表自己。

4.1.3 STL理想和概念

那么STL是什么？它来源于一种尝试，应用数学的一般性的理想去解决数据和算法的问题。对容器里面的对象进行排序并写一个算法去操作这些对象，对这类问题的思考也就是理想的方向，独立和组合地表达概念：

- 用代码直接表达概念
- 用代码直接表达概念之间的关系
- 用独立的代码直接表达独立的概念
- 组合代码自由表达概念，只要组合言之有理。

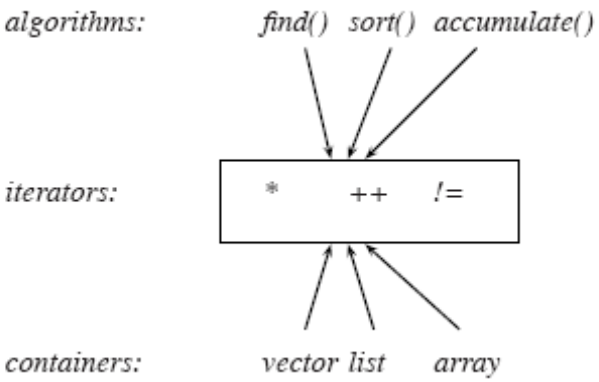
我们想要能够：

- 存储不同类型（比例，int，point和指向Shape的指针）的对象
- 存储不同类型的对象到各种容器（比如，list，vector和map）中
- 应用各种的算法（sort，find和accumulate）于容器中的对象
- 使用各种标准（比较、判断等）于算法之上

而且，我们需要使用的对象、容器和算法是静态类型安全的，尽可能地快，尽可能地紧凑而不冗赘的，并且是可读的。同时满足这些要求是不容易的。事实上，我花了超过10年的时间在寻找解决这一问题的方法，但是没有完全成功（§4.1.2）。

STL解决方法是基于用它们的元素类型参数化容器，并且将算法和容器完全分离。每一种类型的容器都提供一种迭代器类型，所有对容器元素的访问能通过只使用这种类型的迭代器完成。迭代器定义了算法和算法所要操作的数据之间的接口。一方面，算法能够使用迭代器实现，而不需要知道它所要应用到的容器的信息。每一种迭代器是完全独立的，除了对要求的操作，比如*和++提供相同的语义。

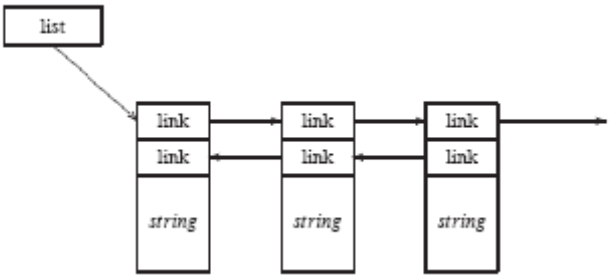
算法使用迭代器，容器的实现者实现他们的容器的迭代器。



让我们考虑一个著名的例子，就是那个Alex Stepanov最先向委员会展示的（San Jose，加利福尼亚州，1993）。我们需要从不同的容器中查找不同类型的元素。首先，这儿有两个容器：

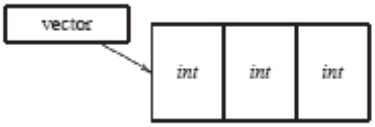
```
vector<int> vi; // int的vector
list<string> ls; // string的list
```

vector和list是标准库版本中的用模板实现的容器。一个STL容器是非入侵式的数据结构，你可以拷贝任何类型的元素。我们可以通过图来表达list<string>(双向链表)，如下：



注意到链接信息不是元素类型的一部分，STL容器（这里的list）为它的元素（这里的string）管理内存并提供链接信息。

类似的，我们可以表示一个vector<int>象这样：

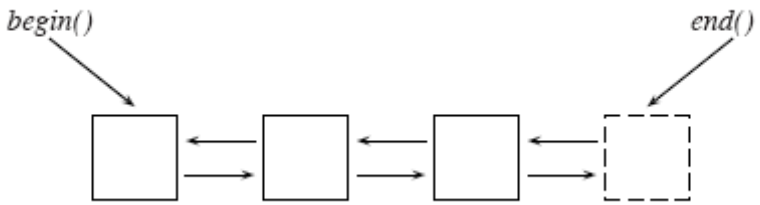


注意到元素是存储在由vector和list所管理的内存中的，这跟§4.1.1所提到的Simula风格的容器是不一样的，它最小化分配操作，最小化每个对象的内存，并且保存一个间接地访问元素方法。相应的代价就是当对象第一次插入到容器时的拷贝操作，如果拷贝的代价很高，那么程序员则倾向于使用指针作为元素。

假定容器vi和ls都被相应元素类型的值恰当地初始化了。那么从vi中查找第一个值为777的元素和在ls中查找第一个值为“Stepanov”元素则是有意义的：

```
vector<int>::iterator p = find(vi.begin(), vi.end(), 777);
list<string>::iterator q = find(ls.begin(), ls.end(), "Stepanov");
```

基本想法是你能把任何容器中的元素当成元素序列。容器“知道”它的第一个元素的位置，也知道它的最后一个元素的位置。我们把指向一个元素的对象称之为“迭代器”。我们可以使用一对迭代器来代表容器中的元素。begin()和end()，其中begin()指向第一个元素，end()是最后一个元素的下一个。通过图来表示这种一般的模型：



end()迭代器指向最后一个元素的下一个而不是最后一个元素，这样使得我们也可以使用相同的方法来表示空序列：

通过迭代器你能够做什么？你可以获得迭代器所指向的元素的值（象指针一样使用*），让迭代器指向下一个元素（象指针一样使用++），比较两个迭代器看它们是否指向同一元素（使用==或!=），令人意外的，这些操作对于实现find()已经足够了：

```
templae< class Iter, class T >
Iter find( Iter first, Iter last, const T& val ){
    while( first != last && *first != val )
        ++first;
    return first;
}
```

这是一个简单、非常简单的函数模板，习惯C和C++指针的人们发现这些代码非常容易读：first != last检查我们是否到达终点，*first != value检查我们是否找到我们正在查找的值（val）。如果没有，我们增加迭代器first，使得它指向下一个元素，然后再检查。当find()返回时，它的值要么指向第一个值为val的元素；要么指向最后一个元素的下一个（end()）。所以我们可以写：

```
vector<int>::iterator p = find(vi.begin(),vi.end(),777);
if (p != vi.end()) { // we found 777
    // ...
} else { // no 7 in vi
    // ...
}
```

这是非常非常简单的。它就象数学课本的前几页那么简单，能够快速翻过。然而，我知道我不是惟一花大量时间想搞清楚这儿到底发生了什么事，想搞清楚为什么这是一个好想法的人。象简单的数学，the first STL rules and principles generalize beyond belief。

首先考虑算法：在调用find(vi.begin, vi.end(), 777)，迭代器vi.begin()和vi.end()在算法中分别变成first和last。对于find()，first只是“某种指向int的东西”。vector<int>::iterator的明显实现就是一个指向int的指针int*。因此，*变成指针解引用，++变成指针递增，!=变成指针比较。就是说，find()的实现是显然和最优的。

请注意，STL没有使用函数调用去访问操作（比如*和!=），那对算法来说是有效的参数，因为他们依赖于模板的参数。在这儿，模板根本不同于大多数“generics”机制，“generics”机制依赖于间接的函数调用（象虚函数），正如由Java和C#所提供的一样。对于一个好的优化器，vector<int>::iterator可以是一个类，对于*和++操作就象提供内联函数一样没有开销，这样的优化器在现在是很普通的，使用迭代器类而不是指针，通过捕获没有根据的假设可以改进类型检查，比如用于vector的迭代器就是一个指针：

```
int *p = find(vi.begin(), vi.end(), 777); // 错误
// 冗赘的但是正确的写法如下
vector<int>::iterator q = find(vi.begin(), vi.end(), 777 );
```

C++0x将提供处理这种冗赘的方法，参见§8.3.2。

另外，不定义算法和它的类型参数之间的接口作为一组带独特类型的函数提供了很大的灵活性，这被证明是非常重要的[130]（§8.3.3）。比如，标准库算法copy能用于不同类型的容器之间的拷贝：

```
void f(list<int>& lst, vector<int>& v)
{
    copy(lst.begin(), lst.end(), v.begin());
    // ...
    copy(v.begin(), v.end(), lst.end());
```

```
}
```

为什么我们不抛弃迭代器，转而使用指针呢？一个理由是vector<int>::iterator可以是一个提供范围检查的类。查看另一个函数调用find()，我们可以得到另一个不是那么精妙的解释：

```
list<string>::iterator q = find(ls.begin(),ls.end(),"McIlroy");
if (q != ls.end()) { // we found "McIlroy"
    // ...
} else { // no "McIlroy" in ls
    // ...
}
```

这儿，list<string>::iterator不会成为string*。事实上，对于大多数普通的链表的实现，list<string>::iterator都会变成一个Link<string>*，其中Link是一个节点类型，象下面这样的：

```
template<class T> struct Link {
    T value;
    Link* suc;
    Link* pre;
};
```

那意味着*表示p->value（“返回值域”），++意味着p->suc（“返回下一个节点的指针”），而 !=则是指针比较（比较Link*）。同样的，实现是显然和最优的。然而，这与我们早些时候看到的vector<int>::iterator完全不同。

我们使用模板的组合和重载解析去选择根本上不同的，仍是优化的，find()的定义中使用到的这些操作的实现。注意到没有运行时的分派，没有虚函数调用。事实上，只是调用微不足道的内联的函数和基本的操作，例如*和++用于指针。在执行时间和代码量上，我们都取得了绝对的优化。

为什么不使用“序列”或“容器”作为基础概念而是使用“迭代器对”呢？部分原因在于“迭代器对”是比“容器”更一般的概念。比如，对于迭代器，我们可以对容器的前半部进行排序：sort(vi.begin(), vi.begin()+vi.size()/2)。另一个原因是STL遵守C++的设计规则，我们必须提供转变路径和统一地支持内建类型和用户定义类型。如果某人把数据放到普通的数组里面，那该怎么办？我们仍然能够使用STL算法，比如：

```
int buf[max];
// ... fill buf ...
int* p = find(buf,buf+max,7);
if (p != buf+max) { // we found 7
    // ...
} else { // no 7 in buf
    // ...
}
```

这里，find()中的*，++和!=真的就是指针操作！。象C++自身，STL也兼容老的概念如C数组。因此，STL满足总是提供转变路径的C++理想（§2）。它同样也满足对内建类型（本例中的数组）和用户定义类型（比如vector）提供统一对待的理想。

解释算法使用迭代器而不是容器或明确的序列抽象的另一个理由，是想要获得最优的性能：直接使用迭代器而不是通过获得指针或是来自另一个抽象的索引，消除一层间接性。

正如接受把容器和算法作为ISO C++标准库框架一样，STL由许多容器组成（比如vector，list和map）和数据结构（比如数组）都能当成序列使用。除此之外，有大概60种算法（比如find，sort，accumulate和merge），不能在这儿一一陈列。详细信息请参考[6，126]。

因此，我们使用简单算术来观察STL技术是怎么将算法从容器中分离出来的，从而减少了我们必须自己写和维护的代码量。有60*12（即720）种算法和容器的组合，但在标准中只有60+12（即72）种定义。这种分离将组合的乘变成了简单的加。如果我们考虑用于算法的元素类型和策略（policy）参数（函数对象，参见§4.1.4），我们会有更加深刻的印象：假定我们有N个算法和M个可选的准则（策略）和X个有Y个元素类型的容器，那么，使用STL的方法只需要N+M+X+Y个定义，而“手工来写”的话，则要求N*M*X*Y种定义。在现实设计中，这种差别不是太强烈因为设计者通常通过转换、类的派生、函数参数等的组合解决N*M*X*Y的问题。但是STL方法比以前的方法更加的清晰和系统化。

STL优雅和高性能的关键是—象C++自身—是基于直接的内存和计算的硬件模型。STL中序列的概念是从硬件的角度把内存看成是一组对象的序列。STL的基本语义直接映射到硬件指令，允许算法的优化实现。支持编译期的模板解析和优先使用内联，也是有效映射STL的高级表达到硬件层的关键。

4.1.4 函数对象

STL和泛型编程一般来说都欠了—直率的并且公认的（例如，[124]）—函数式编程的债。lambda和高阶函数在哪？C++没有直接支持任何象lambda和高阶函数那样的东西（尽管有提议添加嵌套函数、闭合、lambdas等，参见§8.2），作为替代，类定义应用操作符（application operator，即operator()），称之为函数对象（甚至仿函数），取代其角色并成为现代C++参数化中的主要机制。函数对象建立在一般C++机制的基础上，提供前所未有的灵活性和性能。

STL框架，到现在为止的描述，是有些苛刻。每个算法只做标准指定它做的一件事。比如，使用find()，我们找一个和我们给的参数相同值的元素。事实上查找一个具有某些属性的值是更加普遍的，比如不分大小写地匹配一个字符串或匹配一个允许非常细微的差异的浮点数。

比如查找一个满足某种判断的值，而不是7，比如查找小于7的值：

```
vector<int>::iterator p = find_if(v.begin(),v.end(),Less_than<int>(7));
if (p != vi.end()) { // element < 7
    // ...
} else { // no such element
    // ...
}
```

Less_than<int>(7)是什么？它是一个函数对象；即是一个类的对象，它具有operator()，定义来执行一个动作。

```
template<class T> struct Less_than {
    T value;
    Less_than(const T& v) :value(v) { }
    bool operator()(const T& v) const{ return v<value; }
};
```

比如：

```
Less_than<double> f(3.14); // f holds 3.14
bool b1 = f(3); // true: 3<3.14 is true
bool b2 = f(4); // false: 4<3.14 is false
```

从2005年来看，在D&E和TC++PL1中没有提到函数对象是奇怪的。对它们的描述应该需要完整一节。甚至用户定义operator()的使用也没有提到，尽管它在很早以前就已经出现了并且其功能很独特。比如，它是我在一开始允许重载的一组操作符中的一个（在=之后；参见D&E§3.6）[112]，这些操作符和其它东西用于模拟Fortran的下标符号。

我们使用STL算法find_if，使用Less_than<int>(7)到vector的元素上。find_if与find()的定义

的区别在于使用一个用户提供的判断而不是相等：

```
template<class In, class Pred>
In find_if(In first, In last, Pred pred)
{
    while (first!=last && !pred(*first))
        ++first;
    return first;
}
```

我们简单用!pred(*first) 代替 *first != value。函数模板find_if()将接受任何能把给定元素值作为它的实参的对象。特别地，我们可以调用find_if()，把一个普通的函数作为它的第三个参数:

```
bool less_than_7(int a)
{
    return a<7;
}
vector<int>::iterator p = find_if(v.begin(),v.end(),less_than_7);
```

然而，这个例子显示为什么我们经常选择函数对象而不是函数：函数对象能被1个或多个参数初始化，并能携带信息，用于后面的使用。函数对象能够带状态，这会产生更加一般和更加优雅的代码。如果需要，我们还可以在后面检查状态，比如:

```
template<class T>
struct Accumulator { // keep the sum of n values
    T value;
    int count;
    Accumulator() :value(), count(0) { }
    Accumulator(const T& v) :value(v), count(0) { }
    void operator()(const T& v){ ++count; value+=v; }
};
```

一个Accumulator对象能够重复地传递给算法，局部的结果是存储在对象中，比如

```
int main()
{
    vector<double> v;
    double d;
    while(cin>>d) v.push_back(d);
    Accumulator<double> ad;
    ad = for_each(v.begin(),v.end(), ad);
    cout << "sum==" << ad.value
        << ", mean==" << ad.value/ad.count
        << '\n';
    return 0;
}
```

标准库算法for_each把序列中的每个元素作为它的第三个参数的实参，并将第三个参数作为返回值返回。替代函数对象的方法可以是混乱地使用全局变量保持value和count。在多线程系统，使用全局变量不仅混乱，而且会给出不正确的结果。

有趣的是，使用简单的函数对象比相同的函数有更好的性能。原因在于它们通常是没有虚函数的简单类，因此当我们调用一个成员函数时，编译器知道我们调用的是哪一个函数。那样的

话，就算是一个头脑简单的编译器也能获得内联所需要的所有信息。另一方面，函数是通过指针传递进来的，优化器通常没有能力对与指针相关的东西进行优化。这是非常有效的（比如，在速度上可能是50倍）当我们传递一个对象或函数用于相同的简单操作，比如sort中的比较准则。特别的，当对某些带简单比较操作符（比如int和double）的类型进行排序 [125]，STL的sort()比传统的qsort()要强几倍，原因就在于内联函数对象。受到函数对象的刺激，一些编译器现在也能对函数指针进行内联只要在调用时把它作为常量传递进去。比如，今天的某些编译器能够内联compare的调用到qsort中：

```
int compare(double* a, double* b) { /* ... */ }
// ...
qsort(p,max,sizeof(double),compare);
```

在1994年，没有一个C/C++的编译器可以做到。

函数对象是C++的机制用于高阶构造，它不是最优雅的表达高阶的想法，但在通用语言中，它拥有惊人的表现力和与生俱来的高效。要从常规函数式编程设施的一般编译器中获得相同的效率（在时间和空间上）的代码，要求优化器要非常完备。作为具有强表现力的例子，Jaakko Järvi和Gary Powell展示如何提供和使用lambda类，使得下面的例子具有跟前面例子一样 [72]：

```
list<int> lst;
// ...
Lambda x;
list<int>::iterator p = find_if(lst.begin(),lst.end(),x<7);
```

注意重载解析使得我们可以让元素类型int隐式（推演得到）。如果你只是想让<工作，那么你不需要创建一个普通库，你在少于十几行的代码中增加Lambda的定义和<就可以完成。使用Less_than，上面的例子可以简单重写为：

```
class Lambda {};
template<class T>
Less_than<T> operator<(Lambda,const T& v)
{
    return Less_than<T>(v);
}
```

所以，在find_if调用中的参数x<7变成了调用operator<(Lambda, const int &)，而它又产生一个Less_than<int>的对象。那就是我们在这一节中使用的第一个例子。不同的是，我们获得了更简单和更直观的语法。这是一个展示C++强大的表达能力和展示库的接口能够比它的实现更简单的好例子。自然，运行时刻或空间的开销也不会比手工写的循环去查找一个小于7的值多。

C++与高阶函数最接近的地方是返回一个函数对象的函数模板，比如operator<返回一个有恰当类型和值的Less_than对象。好几个库已经扩展了这种想法，广泛地支持函数式编程（比如Boost的函数对象和高阶程序库[16]和FC++[99]）。

4.1.5 粹取

C++没有提供一般的编译期的查询类型属性的方法。在STL和许多其它库中，使用模板提供用于不同的类型的泛型类型安全设施，这成为一个问题。最开始，STL使用重载去解决这个问题，（比如，注意到类型int是通过x<7推演出来的§4.1.4），然而，使用重载是非系统化的，因此难以使用和错误检测。基本的解决方法是由Nathan Myers在模板化iostream和string[88]的工作中发现的。基本想法是提供辅助模板，“粹取”，去包含一组类型中想要的信息。考虑怎么找到由迭代器所指向的元素的类型，对于一个list_iterator<T>，它的元素的类型是list_iterator<T>::value_type；对于普通指针T*，它的类型就是T。我们可以表达如下：

```
template<class Iter>
```

```
struct iterator_trait {
    typedef Iter::value_type value_type;
};
template<class T>
struct iterator_trait<T*> {
    typedef T value_type;
};
```

就是说，迭代器的value_type是它的成员类型value_type。然而，指针是迭代器的普通形式，它们没有自己的成员类型。所以，对于指针，我们使用指针所指向的类型作为value_type。涉及到的语言构筑称之为偏特化（1995添加到C++，§5）。粹取会导致稍微的代码膨胀（尽管他们没有对象代码或运行时的代价），尽管这种技术具有很强的扩展性，但当添加新的类型时还是时常需要程序员的关注。然而，“concept”机制（§8.3.3）承诺提供直接的语言支持去表达类型属性的想法，那将减少粹取的使用。

4.1.6 迭代器分类

到目前为止所描述的STL模型的结果变成一团乱麻，因为每种算法依赖于特定容器所提供的迭代器的特性。为获得协同工作的能力，迭代器接口不得不被标准化。它可以简单地通过定义一组用于所有迭代器的操作符。然而，这样做则违反现实：从算法的观点看list、vector和输出流，它们有本质上不同的属性。比如，你可以通过下标来访问vector的元素，你可以增加一个元素到list中而无需扰乱相邻之间的元素，你可以从输入流中而不是从输出流读取数据。因而STL提供了迭代器的分类：

输入迭代器（输入流）

输出迭代器（输出流）

前向迭代器（我们可以反复读和写同一元素，单向list）

双向迭代器（双向list）

随机访问迭代器（vector和数组）

这样的分类对于那些希望让算法与容器协同工作的程序员来有指导的作用。它允许我们最小化算法与容器之间的耦合。不同的算法使用不同的迭代器种类，通过重载自动选择最恰当的算法（在编译期）。

4.1.7 复杂度要求

STL包含了所有标准库操作和算法的复杂度测量（使用大O符号）。这对于一个用于工业的语言的基础库来说是新奇的。过去的希望是，现在仍是，设置先例以更好地规范说明库。另一方面—不是那么的创新—在详细描述库时系统地使用前提和后置条件（postcondition，操作完成时必须满足的约束）。

4.1.8 Stepanov的看法

这儿对STL的描述集中在C++环境中的语言和库问题。为了获得补充看法，我询问了Alexander Stepanov的看法[106]：

在1976年10月，我发现到某些算法—可以并行减少—与monoids关联：一组元素与一个相关联的操作。这个发现让我相信存在这么一种可能性，把每一个有用的算法与数学理论相关联。而这种关联允许最大可能的使用和有意义的分类。正如数学家知道把理论提升到他们最一般的场景，我也想提升算法和数据结构。知道算法使用的数据的准确类型是罕见的需求，因为大多数的算法可以处理许多类似的类型。为了写一个算法，只需要知道要操作的数据的属性。我把在算法调用的一组具有类似属性的类型称之为算法的underlying concept。还有，为了选择有效率的算法，需要知道这些操作的复杂度。也即是说，对于概念来说，复杂度是接口的一个重要的部分。

在 70 年代末，我开始关注 John Backus 在 FP[7]上的工作。虽然他的函数式编程的想法从本质上震撼了我，但我意识到他尝试永久固定函数式形式的数字从根本上就是错误的。函数式形式的个数—或者，我现在称之为泛型算法的个数—总是随着我们不断地发现新的算法而不断地在增加。在 1980 年，我和 Dave Musser、Deepak Kapur 一起开始在 Tecton 语言上用代数理论去描述算法。语言本身是实用的，因为在那时我没有意识到内存和指针是编程的基本部分。我还花了许多时间学习亚里士多德和他的后继者的著作，这让我更好地理解在对象上的基础操作，象相等、拷贝和局部与整体之间的关系。

1984 年起我开始与 Aaron Kershenbaum 合作，他是图形算法方面的专家。他让我相信要认真对待数组。我认识序列是递归可定义的，因为它一般被认为“理论上健全的”方法。Aaron 向我展示许多基本依赖于随机访问的算法。我们在 Scheme 中制造了一大堆组件，它们能一般地实现一些复杂的图形算法。

Scheme 上的工作导致承诺在 Ada 上制造一个一般的库。Dave Musser 和我制造一个通用库，用于处理链接结构。由于那时的 Ada 编译器状况，我尝试去实现能够用于任何序列结构（list 和数组）上的算法失败了。我有许多相同的 STL 算法，但不能编译它们。基于这项工作，Dave Musser 和我发表了一篇文章介绍泛型编程的观点，坚持对有用的高效的算法进行抽象。我从 Ada 中学到的东西是静态键入的值（the value of static typing）是一种设计工具。Bjarne Stroustrup 从 Simula 中学到这一点。

1987 年在贝尔实验室，Andy Koenig 教我 C 的语义，在 C 后面的抽象机器被揭开了。我还读了大量的 UNIX 和 Plan 9 的代码：Ken Thompson 和 Rob Pike 的代码当然影响了 STL。无论如何，1987 年 C++还没有为 STL 做准备，我必须跟进。

在那时，我发现我对 Euler 的著作和对数学本质的理解发生了重大的转变。我是 de-Bourbakized，不再相信集合，从 Cantorian 的天堂中被驱逐出来。但我仍然相信抽象，但现在我知道，抽象只有目标而没有起点。我还学到，人只有适应现实中的抽象，没有其它路可走。对我来说，数学不再是一门科学理论，而是关于数字和形状的科学。

1993 年，在无关联的工程上工作 5 年之后，我回到泛型编程。Andy Koenig 建议我写一份把我的库加入到 C++标准中的提议。Bjarne Stroustrup 热情地赞同这项提议，然后在一年内，STL 被标准接纳。STL 是 20 年的思考的结果，但在两年之内形成。

STL 只是有限的成功。然而当它变成一个被广泛使用的库，它本质的东西并没有被理解。人们对使用（滥用）C++模板的泛型编程仍感到困惑。泛型编程是对算法和数据结构的抽象和分类。它从 Knuth 那儿得到灵感，而不是来自于类型理论。它的目标是增量构建系统分类的有用的、高效的和抽象的算法和数据结构。这个目标仍然是个梦想。

你能从www.stepanovpapers.com中找到导致STL产生的工作的参考。我对Alex的思想将产生的长远的影响比他本人还要乐观。然而，我们都同意，STL只是这个长途旅行的第一步。

4.1.9 STL的影响

STL对关于C++思考的影响是巨大的，在STL之前，我列出C++支持的三个基本的编程风格（“范例”）[113]：

- 过程编程
- 数据抽象
- 面向对象编程

我认为模板是对数据抽象的一种支持。在使用STL一段时间后，我总结出第四种风格：

- 泛型编程

这项技术是基于模板的使用，并且主要由函数式编程的技术所启发，与传统的数据抽象有质的不同。人们对类型、对象和资源的思考是不同的。新C++库已经使用模板完成—静态类型安全并且是高效的。模板是系统编程和高性能数值计算的关键[67]，对于高性能数值计算，资源

管理和正确性是关键的。STL本身并不总是适合那些领域。比如，它没有提供直接支持线性代数，它能巧妙地在禁止自由存储使用的硬实时系统中使用。然而，STL说明什么能够由模板完成并给出许多有效使用技巧的例子。比如，迭代器（和allocator）的使用，分离逻辑内存访问与实际物理内存是许多高性能数值计算技术的关键[86，96]，使用小的、容易内联的对象是许多嵌入式系统编程优化的关键。一些技术被标准委员会的关于性能的技术报告所记载（§6.1）。C++社团在90年代末和2000年初开始重视STL和泛型编程，以致在某种程度上是反应过度的—另外一方面—大的软件开发社团倾向于过度使用“面向对象”技术，而那依赖于类层次和虚函数。

显然，STL并不完美，也没有完美的东西。不管怎么样，它打开了一片新大陆并且它的影响超越了庞大的C++社团（§9.3）。它同样也激发了许多更守条例和更冒险的使用模板技术的方法。人们谈论“模板元编程”（§7.2.2）和generative programming [31]并试着将由STL领导的技术向前推进并超越STL。另一条战线是考虑C++怎么才能更好地支持有效的模板使用（concept，auto等，参见 §8）。

不可避免地，STL的成功同时也带来了它自身的问题。人们想把所有的代码都写成STL风格的。然而，象许多其它风格或技术，STL风格或泛型编程并不是所有问题的理想的解决之道。比如，泛型编程依赖于模板和在编译期对所有名字绑定完全解析的重载。它不支持在运行时进行解析的绑定的机制，那是类层次和它们相关的面向对象设计技术所支持的。象所有成功的语言机制和编程技术，模板和泛型编程变得流行起来甚至是被滥用。程序员基于模板实例和推演是完全图灵的，创建真实的过分装饰的和易损坏的结构，正如我早期观察到的C++的面向对象的设施和技术：“不能因为你能够做到，就意味着必须这么干”。发展一种综合使用C++支持的不同的编程风格的简单框架是未来几年的主要挑战。作为一种编程风格，“多范例编程”[121]没有完全得到发展，它通常提供许多更优雅和更好的替代方法[28]，但我们（还）没有一个简单和系统的方法去组合这些编程风格，甚至它的名字就暴露了它基本的弱点。

STL的另一个问题是它的容器是非侵入的。从代码清晰的角度和独立的观点看，非侵入有着巨大的优势。然而，它意味着我们需要拷贝元素到容器或插入具有默认值的对象到容器中，然后在后面再给它想要的值。有时这样做是低效和不方便的。比如，人们不愿意往vector中插入大对象就是基于这样的原因。作为替代方法，他们插入指向这些大对象的指针到容器中。类似的，标准容器提供的用于元素的隐式的内存管理是相当方便的，但有些应用（比如，在某些嵌入和高性能系统）中，这样的隐式内存管理必须避免。标准容器提供保证避免隐式内存管理的特性（比如reserve），但它们必须被理解，使用起来才能避免问题。

4.2 标准库的其它部分

从1994年起，STL所主宰了标准库的工作，并提供它的主要创新点。然而，STL并不是惟一的工作。事实上，标准库还提供其它的组件：

- 基本的语言运行时支持（内存管理 、运行时类型信息（RTTI）、异常等）
- C标准库
- STL（容器、算法、迭代器、函数对象）
- iostreams（由字符类型模板化，隐式的本地化）
- locales（objects characterizing cultural preferences in I/O）
- string（由字符类型模板化）
- bitset（有逻辑操作的一组bit）
- complex（由标量模板化）
- valarray（由标题模板化）
- auto_ptr（由类型模板化的用于管理对象的资源句柄）

有多种理由，其它库组件的故事比STL较无趣和较少启发性。大多数时间，在这些组件上的工作进展是与其它工作隔离的。没有全面的设计或设计哲学。比如，bitset是带范围检查的而string

则不是；而且，几个组件的设计（比如string、complex和iostream）是受到兼容性的制约。由于设计者尝试处理所有的要求、约束和已经存在的惯例，好几个组件（iostream和locale）遭受到“second-system effect”。基本上，标准委员会不包含“由委员会设计”，因此STL反映了一种明确的哲学和一致的风格，而大多数其它组件则不是这样的。每一个都代表它自己的风格和哲学，有些（比如string）同时表现出好几种风格和哲学。我认为complex在这儿算是例外，它基本上就是我原来的设计[91]，允许不同的标量类型进行模板化：

```
complex<double> z; // double-precision
complex<float> x; // single-precision
complex<short> point; // integer grid
```

它很难与数学混为一谈。

委员会的确有过多次关于标准库的范围的严肃讨论。讨论的背景是关于小的和被广泛接受的C标准库（C++标准采纳它，只对它做了微小的修改）和大公司的基础库。在标准化进程的早些年里，我明确表达了一些用于划定C++标准库范围的方针：

首先，现在被广泛使用的关键库必须标准化。这意味C++与C标准库之间的接口必须确定，iostream库也必须规定。除此之外，基本的语言支持也必须详细说明。

其次，委员会必须看它能否对满足“更有用的和标准的类”的公共要求。比如string，不会被委员会变成一个一团糟的设计并且不会与其它C++工业库竞争。任何超越C库和被委员会接纳的iostream的库必须是自然地由模块构建而不是野心勃勃的框架。标准库的重要角色是让独立开发的和更野心勃勃的库更容易沟通。

最后一句划定了标准委员会的工作范围。对标准库的需求的详尽规范，包含构建模块用于更加有雄心的库和框架，强调绝对的效率和极端的一般性。我经常把对容器元素的访问涉及到虚函数调用时将不可能有足够的效率和容器不能存储任意类型则不具有足够的一般性（参见§4.1.1）作为例子，去说明这些需求的重要性。委员会觉得标准库的作用应该是支持而不是取代其它库。

在1998年，标准化工作的最后，委员会普遍都觉得我们在库方面工作做得还不够，还有实验不够充分，我们对库的性能测试方面的关注太少。问题是在将来怎么强调这些问题。一方面—传统的标准进程—通过技术报告的方法（参见§6.2）。另一方面是由Beman Dawes在1998年启动的，称为Boost[16]。下面这段话引用自boost.org（2006年8月）：

“Boost提供免费的同行检查的可移植的C++源码库。我们强调库要跟C++标准库一起很好地工作。Boost库打算成为广泛地有用的，且对于广大范围的应用程序是可用的。Boost许可鼓励商业和非商业的使用。我们的目标是建立“已存在的实践”和提供参考实现，因此Boost库是适合最后成为标准库的一部分。C++标准委员会的库技术报告（TR1）中已经包含了10个Boost库，使之成为未来C++标准的一部分。更多的Boost库正被提议到即将来临的TR2。”

Boost繁荣发展，成为一个库的重要来源，并且通常是标准委员会和C++社团的想法的来源。

5、 语言特征：1991-1998

到1991年，用于C++98的最重要的语言特征已经被接纳：在ARM中详细描述的模板和异常已经正式成为语言的一部分。然而，对它们的详细规范持续了好几年。除此之外，委员会还在许多新的特征上进行工作，比如：

1992　协变返回类型（参见标准10.3.5）—对ARM中描述的特征的第一个扩展
1993　运行时类型标识（RTTI：dynamic_cast，typeid和type_info）§5.1.2
1993　在条件中的声明 §5.1.3
1993　基于枚举的重载
1993　名字空间 §5.1.1
1993　mutable

1993　新的转换（static_cast，reinterpret_cast和const_cast）
1993　布尔类型（bool） §5.1.4
1993　显式模板实例
1993　函数模板调用中的显式模板参数指定
1994　成员模板（嵌套模板）
1994　类模板作为模板参数
1996　类内成员初始化
1996　模板的分离编译（export）§ 5.2
1996　模板偏特化
1996　重载函数模板的偏序

在这儿我不会涉及到细节；关于这些特征的历史可以从D&E[121]找到，TC++PL3[126]中描述了它们的使用。显然，这些特征中的大多数是在被提议和讨论很久之后，才被投票允许收入到标准中。

委员会有用于接纳一个新的特征的全面的标准吗？不完全有，引进类、类层次、模板和异常（组合）代表了一种审慎的尝试，去改变人们关于编程和编写代码的思考方法。这样的主要的改变是我对C++的目标之一。然而，直到委员会可以被认为是在思考时，它还是没有跟它做的一致。个人提出提议，然后一些人获得委员会的通过，只有限定范围内的提议能够进行投票。委员会的成员都很忙，而主要的做实际工作的人们对于抽象的目标和具体细节没有多少耐心，而那些是可以通过详细的检查得以改正。

我的观点是增加的设施的总和对C++支持的编程风格一个更加完整和效率的支持，因此我们可以说这些提议的全面目标是“提供更好的对面向过程、面向对象和泛型编程的支持和对数据抽象的支持”。这是真的，但它不是用于从长长的列表中挑选出提议的具体的标准。某种程度上，该进程成功地选择出一些新的“次要特征”，它是基于提议-提议的个人决定的结果。那不是我的理想，但结果不是太糟。ISO C++（C++98）比早期的C++版本更接近我的理想，C++98是比“ARM C++”（§3）更灵活（强大）的编程语言。主要原因是修正的积累的结果，比如成员模板。

但是，在我看来不是所有被接受的特征都是一种改进。比如“在类内通过常量表达式初始化静态的具有整型类型的const成员”（由John “Max”Skaller提议，代表奥大利亚和新西兰的）和void f(T)和void f(const T)表示相同的函数的规则（由Tom Plum提议的，出于兼容C的理由）都有共同的可疑的差异，尽管我极力反对，但最后都通过了C++投票。

5.1 一些“次要特征”

当委员会在这些“次要特征”上工作时并不觉得这些次要特征次要，并且程序员在使用它们时也不会觉得这些特征次要。比如，我在D&E[121]中称名字空间和RTTI是“次要”的，然而，它们没有显著地改变我们考虑编写程序的方法，所以我将简要讨论几个这样的特征，许多被认为不是“次要”的。

5.1.1 名字空间

C提供一个单一的全局名字空间用于所有的名字，没有便利的地方存放一个单一函数，单一struct或单一编译单元。这会引起名字冲突的问题。我第一次与这个问题进行搏斗是在C++的早期设计中，对编译单元来说，默认所有名字都是局部的，需要显式使用extern声明才能让它们被其它的编译单元看到。这种想法既不能充分解决问题也不能被充分接受，因此它失败了。

当我设计出类型安全链接机制[121]时，我重新考虑这个问题，我观察到：

extern "C"{ /*...*/}

语法的很小的改变，因此实现技术将允许我们有

extern XXX{ /*...*/}

意味着在XXX中声明的名字是在独立的XXX范围内的，并且其它范围只能通过XXX::才能访问到这些名字，这跟从类外访问静态成员的方法一致。

由于各种的理由，大部分是由于没有时间，这个想法处于休眠状态直到1991年才浮现在ANSI/ISO委员会的讨论中。首先，来自Microsoft的Keith Rowe提出一个提议，建议如下符号：

bundle XXX{ /*...*/};

作为一种定义有名字范围的机制，一个操作符用于把所有bundle中的名字引进到其它域。这导致一场—不是非常精力充沛的—在扩展组中几个成员间的讨论，包括Steve Dovich，Dag Bruck，Martin O'Riordan和我。最后，由来自西门子的Volker Bauche，Roland Hartinger和Erwin Unruh提炼了这种想法并形成一份提议，不需要使用新的关键字：

::XXX::{/*...*/};

这导致扩展组中的一场严肃的讨论，特别地，Martin O'Riordan论证这个::符号会与用于类成员和全局名字的::发生歧义。

在1993年初，我已经—在标准会议的讨论中和许多email的帮助下—综合成一个一致的提议。我回想对名字空间有技术贡献的人有Dag Brück，John Bruns，Steve Dovich，Bill Gibbons，Philippe Gautron，Tony Hansen，Peter Juhl，Andrew Koenig，Eric Krohn，Doug McIlroy，Richard Minner， Martin O'Riordan，John "Max" Skaller，Jerry Schwarz，Mark Terribile，Mike Vilot和我。除此之外，Mike Vilot主张马上把这种想法通过明确的提议体现出来，这样这个设施将解决在ISO C++库中不可避免的名字问题。除此之外还有各种各样的普通的C和C++技术用于限制名字碰撞的破坏，Modula-2和Ada提供的设施也被讨论过。1993年7月，名字空间在Munich会议中投票通过。因此我们可以写如下代码：

```
namespace XXX{
    int f(int);
}
int f(int);
int x = f(1); // 调用全局的f
int y = XXX::f(1);// 调用 XXX中的f
```

在1993年11月的San Jose会议中，决定采用名字空间去控制标准C和C++库中的名字。

原始的名字空间设计包括一些其它设施，比如名字空间别名，允许对长的名字空间的缩写，using声明将把单独名字引进名字空间。三年后，增加了参数依赖的查找（ADL或"Koenig查找"），使得参数类型的名字空间隐式可见。

结局是一个设施是有用的并且被使用，但就是不被喜欢。名字空间完成了人们期望它们完成的工作，有时优雅，有时笨拙，有时它们做了太多人们希望它们做的事（特别是在模板实例化时的参数依赖的名字查找）。事实是C++标准库只使用单一的名字空间用于它的主要的设施，这是确立名字空间作为C++程序员的一种工具的一种失败的提示。在标准库中使用子名字空间意味着库实现的部分是标准的（也即是说，哪些设施是在哪些名字空间里面，哪些部分依赖于其它部分）。一些库提供商强烈反对限制他们作为传统的编译器实现者的自由—传统的C和C++库的内部组织本质上是不能被约束的。参数依赖的查找从名字空间中得到帮助，但它是在标准化的后期才引进的。还有，ADL受到模板的干扰，某些情况使得它宁愿意选择出人意料的的模板而不是明显的非模板。这里，"出人意料"和"明显"是由用户客气的评论。

这导致一个对C++0x的提议，加强名字空间，限制它们的使用，来自EDG的David Vandevoorde的提议是最有意思的，让某些名字空间进入到模块中[146]—那就是说，提供分离的编译的名字空间，象模块一样被加载。显然，这个设施看起来有点象Java和C#中类似的特征。

5.1.2 运行时类型信息

当设计C++时，我已经不考虑用于确定一个对象类型的设施（Simula的QUA和INSPECT类似

于Smalltalk的isKindOf和isA）。原因是我观察到频繁和严重的滥用严重地损害了程序的组织：人们使用这些设施去实现（慢且丑陋）switch版本的语句。

增加在运行时用于确定的对象类型的设施的原始动力来自于HP公司的Dmitry Lenkov。Dmitry的经验来自于创建主要的C++库，比如Interview[81]，NIH库[50]和ET++[152]。这些库所提供的RTTI机制相互间是不兼容的，所以它们是使用多个库的壁垒。还有，所有这些都要求基类设计者要有相当的远见。因此，需要一个由语言支持的机制。

我参与了这些机制的详细设计，与Dmitry一起，向委员会提出了初始的提议，然后作为委员会中的主要负责人对提议进行提炼[119]。该提议第一次呈给委员会是在1991年7月的London会议，然后在1993年3月在波兰的Oregon会议中被接受。

运行时类型信息机制包含三个部分：

- 操作符dynamic_cast，用于从指向派生对象的基类指针中获取一个指向派生类对象的指针，操作符dynamic_cast转换指针只有当它指向的对象类型真的是指定的派生类时，否则它返回0
- 操作符typeid，用于确定基类指针所指的对象的实际类型
- 一个结构type_info，作用相当于勾子，用以获得与类型相关的运行时的信息

假定一个库提供dialog_box类，它的接口用dialog_box表达。我使用dialog_box和我自己的Sbox

```
class dialog_box : public window {
    // library class known to "the system"
public:
    virtual int ask();
    // ...
};
class Sbox : public dialog_box {
    // can be used to communicate a string
public:
    int ask();
    virtual char* get_string();
    // ...
};
```

因此，当系统/库给我一个指向dialog_box的指针时，我怎么能知道它是否指向我的Sbox呢？注意到我不能修改库，让它知道我的Sbox类，甚至假定我可以，我也不会这么干，因为在那之后我将不得不修改每一个新版本的库。所以，当系统传一个对象给我时，我有时需要问它是不是我"自己类"中的一个。这个问题能够直接使用dynamic_cast得到解决：

```
void my_fct(dialog_box* bp)
{
    if (Sbox* sbp = dynamic_cast<Sbox*>(bp)) {
        // use sbp
    } else {
        // treat *pb as a "plain" dialog box
    }
}
```

dynamic_cast<T*>(p)转换p到目标类型T*，如果*p真的是一个T或者是T的派生类；否则，dynamic_cast<T*>(p)的返回值是0。dynamic_cast的使用对于GUI回调的一个十分重要的操作。因此，C++的RTTI可以看成是用于支持GUI的最小设施。

如果你不想显式测试，那你可以通过使用引用而不是指针：

```
void my_fct(dialog_box& br)
{
    Sbox& sbr = dynamic_cast<Sbox&>(br);
    // use sbr
}
```

现在，如果dialog_box不是期待的类型，那么异常将会抛出，能够在其它地方进行错误处理（§5.3）。

显然，运行时类型信息是最小的，这导致了要求最大限度的设施：一个完全元数据（meta-data）设施（反射）。到目前为止，对于编程语言来说，这被认为是不适宜的，其它东西将离开它的应用，带着最小的记忆轨迹

5.1.3 在条件中声明

留意我们在“box例子”中用到的转换、声明和测试方法：

```
if( Sbox *sbp = dynamic_cast<Sbox*>(bp)){
  // use sbp
}
```

这制造了一个简洁的逻辑实体，最小化了忘记测试的可能性，并把变量的范围限制到了最小。比如，dbp的域就是在if语句中。

这个设施被称之为“条件中的声明”并反映在“在for语句初始化中声明”，和“象语句一样声明”。允许在任何位置声明的思想是由Algol68的把语句作为表达式的定义所激发[154]。因此当Charles Lindsey向我解释说Algol68由于技术原因还不支持在条件中的声明时我感到非常吃惊，那是在HOPL-II会议上。

5.1.4 布尔类型

有些扩展真的是非常次要，但在C++社团中关于它们的讨论则不是。考虑最普通的一个枚举：

```
enum bool{ false, true};
```

每一个主要程序都会有一个或跟或者下面的其中一个：

```
#define bool char
#define Bool int
typedef unsigned int BOOL;
typedef enum { F, T } Boolean;
const true = 1;
#define TRUE 1
#define False (!True)
```

这种变化显然是没有尽头的，更坏的是，大多变形意味着语义上的微小的变异。并且大多数与其它的在一起使用时会发生冲突。

自然，这个问题已经存在多年，Dag Brück（代表瑞典和爱立信）和Andrew Koenig（AT&T）决定对该问题采取某些行动：“C++中关于布尔数据类型的想法属于宗教问题。一些人，特别是那些来自Pascal或Algol，认为C没有这样的类型，因此C++也不应该有的想法是不合理的；同样的，其它人，特别是那些来自C的，认为增加这样的类型到C++中是不合理的。

自然地，第一个想法是定义一个枚举类型。然而，Dag Brück和Sean Corfield（UK）测试了许多C++代码发现，大多数使用的布尔类型要求隐式bool到int之间的转换。C++没有提供隐式的int到枚举的转换，所以定义一个枚举作为标准的bool将破坏许多已经存在的代码。所以为什么还要一个布尔类型？

- 布尔数据类型是生活的一部分，不管它是否为C++标准的一部分

• 许多冲突的定义使得难以方便和安全地使用布尔类型
• 许多人想要基于布尔类型的重载。

对我来说，稍微有点意外，委员会接受了这些理由，所以bool现在是C++中独特的整数类型，带有文字常量true和false，非0值可以隐式转换成true，而true能够隐式转换到1。0能隐式转换成false，而false能隐式转换到0。这确保高度的兼容性。

许多年后，bool被证明是大家喜爱的。意料之中，我发现它在用于教那些没有编程经验的人们时很管用。bool在C++中取得成功之后，C标准委员会决定把它也加到C里面，不幸的是，他们通过不同的和不兼容的方法实现，所以在C99[64]中，bool是在头文件<stdbool.h>中用关键字_Bool定义的一个宏，在头文件中还有true和false的宏。

5.2 export论战

从最早的设计，模板原来是想允许在一个编译单元中通过声明就能够使用模板[117，35]。比如：

```
template<class In, class T>
In find(In, In, const T&); // no function body
vector<int>::iterator p = find(vi.begin(), vi.end(),42);
```

然后就是编译器和链接器的工作，去寻找和使用find模板的定义（D&E §15.10.4）。那是其它语言构建的工作的方法，比如函数，但对于模板，说起来容易，但实现起来却极端困难。

第一个实现模板的编译器是Cfront3.0（1991年10月），在编译期和链接期的代价耗费巨大。然而，当Taumetric和Borland实现模板时，他们引进了“包含所有东西”的模型：把所有模板的定义放到头文件中，然后当你在独立的编译单元中多次包含同一个文件时，编译器和链接器将会消除多份定义。第一个带有基本的模板支持的Borland编译器在1991年11月20日发布，然后马上推出3.1版本并在1993年11月推出更加鲁棒的4.0版本[27]。微软、Sun和其它厂商也采用（相互之间不兼容）“包含所有东西”的方法上。明显地，这种方法违反了实现（使用定义）通常要与接口（只展现声明）分离的规则，并使得定义对于宏、无意的重载解析等都是变得脆弱。考虑如下例子：

```
// printer.h:
template<class Destination>
class Printer {
    locale loc;
    Destination des;
public:
template<class T> void out(const T& x)
{ print(des,x,loc); }
// ...
};
```

我们可以在两个编译单元中这样使用Printer

```
//user1.c:
typedef int locale; // represent locale by int
#define print(a,b,c) a(c)<<x
#include "printer.h"
// ...
//user2.c:
#include<locale> // use standard locale
using namespace std;
```

```
#include "printer.h"
// ...
```

这显然是非法的，因为在user1.c和user2.c中的环境不一样，会导致Printer定义的不一致性。然而，编译器很难发现这样的错误。不幸的是，这种错误的可能性在C++中要比C要高一些，甚至在C++中，使用模板比不使用模板出错误的可能性也要高一些。原因是当我们使用模板时，在头文件中有许多关于typedef、重载和宏会干扰到模板。这导致防御式的编程实践（比如模板中定义的局部名字的命名以加密的方式以避免冲突，比如_L2）和在分离编译时尽量减少头文件中的代码量。

在1996年，在委员会中爆发了一场精力充沛的争论，关于我们是否应该接受“包含所有东西”的模型用于模板定义，实际上，“包含所有东西”的模型偏离了原来的在独立的编译单元中分离模板定义和声明的模型。双方的论据如下：

- 分离编译模板太难（不是不可能），这样的负担不应该加到实现者身上
- 分离模板编译是必要的，用于适当的代码组织（根据数据隐藏原则）

双方都有许多辅助的论据支持，Mike Ball（Sun）和John Spicer（EDB）领导“禁止分离模析编译”团队，Dag Bruck（爱立信和瑞典）领导“保留分离模板编译”团队。我坚持分离模板编译。甚至在混乱的讨论中，双方在他们的主要论点上都是正确的。在最后，来自SGI的人们—特别是John Wikinson—建议一个新的模型，作为一种妥协被接受了。这个妥协是关键字export被引入用于提示模板可以被分离编译。

分离模板编译在现在变得更坏：export特征仍然被禁上甚至在一些支持分离编译的编译器上，因为支持它将破坏ABI。直到2003年，Herb Sutter（代表微软）和Tom Plum（Plum Hall）提议对标准进行修改，通过这样，没有实现模板分离编译的编译器仍是遵守标准的，那就是把export变成一种可选的语言特征。给出的理由仍然是编译器实现的复杂度和基于一个客观事实，那就是在标准通过的五年后，仍然只有一个编译器能做到模板分离编译。这个提议被80%的人所反对，部分是因为已经有一个支持export的编译器存在；抛开技术上的争论，许多委员会成员认为对一些已经花了大量时间和工作去实现分离模板编译的编译器制造商来说，把export作为一个可选特征是不公平的。

这个伤感传奇的真正主角是EDG编译器的实现者：Steve Adamczyk，John Spicer和David Vandevoorde。他们强烈反对分离模板编译，最后为达到最佳妥协投赞成票，然后花了超过一年的时候完成了他们所反对的东西。这就是专业！编译器的实现正如他的竞争对手所预测的那样困难。但它成功了并带来了一些好处，正如它的支持者所承诺的那样。不幸的是，由于一些对分离模板编译的必要的约束，因此承诺以因没能提供它们预期的利益并且让编译器变得更加复杂而告终。一如往常，在技术问题上的政客般的承诺导致了“warts”。

我怀疑分离模板编译的一个更好的解决方法的主要组成部分是concepts（§8.3.3），另一个就是David Vandevoorde的模块[146]。

5.3 异常安全

在详细说明STL的工作过程中，我们遇到一个奇怪的现象：我们不能很好地谈论模板与异常之间的相互作用。很少有人因为这方面的模板问题而被责怪。其它人开始认为由于没有自动废料收集，异常基本上是缺陷的（比如[20]）或至少是有缺陷的。然而，当库的实现者（Nathan Myers，Greg Colvin和Dave Abrahams）观察这个问题时，他们发现基本上我们有一个语言特征—异常—那是我们还不知道怎么很好地使用它。问题在于资源和异常间的相互作用（或更通常的，invariant（使对象的状态良好定义的属性称之为invariant。）与异常之间的相互作用）。如果抛出一个异常，那会导致资源不可访问，没有优雅地恢复的希望。当我设计异常处理机制时我当然考虑过这一点并提出从构造函数中抛出异常的规则（正确地处理部分构建的组合对象）和“在初始化时获取资源”技术（§5.3.1）。然而，那只是一个好的开始和一个十分重要的基础。我们需要的是概

念性的框架—一个更系统的方法去思考资源管理。和许多人一起，Matt Austern开发了这样的一个框架。

关键是Dave Abrahams在1997年早期所阐述的三个保证：

- 基本保证：保持成员的invariant不变，没有资源泄露
- 强保证：操作要么完全成功，要么抛出异常，让程序维持该操作之前的状态
- no-throw保证：操作不会抛出异常

注意到强保证基本上是数据库的“commit”或“rollback”规则。使用这些基本的概念，库工作组阐述标准库和制造出高效和鲁棒的标准库。标准库对所有操作提供了基本保证，我们不应该通过在构造函数中抛出异常来退出。除此之外，库对关键操作提供强保证和no-throw保证。我发现这个结果是很重要的，应该加到TC++PL的附录中[124]，但由于其它原因没能加进来，后面在[126]添加进来。对于标准库异常保证的细节和使用异常的编程技术，参考TC++PL的附录E。

Matt Austern的SGI STL和Boris Fomitchev的STLPort[42]是第一个使用这些概念去获得异常安全的STL，在1997年春天出现的标准容器和算法得到Dave Abraham的异常安全的支持。

我认为这儿的关键教训是：只是知道一个语言特征是怎么表现是不够的。为了写出好的软件，对需要用到特征的问题我们必须有一个清晰的表达设计策略。

5.3.1 资源管理

异常典型地—并且是正确的—被看成是一种控制结构：throw将控制权丢给了其它的catch语句。然而，操作的顺序只是该场景中的一部分：使用异常的错误处理大多都是跟资源和invariant有关。对于C++类来说异常就是类的一部分，异常原语提供一种必要的用于保证和标准库设计的基础。

当抛出一个异常时，每一个在从throw到catch路径上的已经构造的对象都会被销毁，而部分构造的对象的销毁则不会被调用。如果没有这两条规则，异常处理将会变得难以管理（在没有其它支持的情况下）。我（笨拙地）命名这个基本的技术为“在初始化时获取资源”—通常缩写为“RAII”。经典的例子[118]是

```
// naive and unsafe code:
void use_file(const char* fn)
{
    FILE* f = fopen(fn,"w"); // open file fn
    // use f
    fclose(f); // close file fn
}
```

这看起来很合理，然而，如果在调用fopen之后，fclose之前出错，异常会使得use_file在没有调用 fclose的时候退出。请注意同样的问题同样发生在不支持异常处理的语言上。比如，调用标准C库函数longjmp也会有同样的后果。甚至一个错误引入的return，都会使得程序泄露一个文件句柄。如果我们想要支持容错系统，我们必须解决这个问题。

通常的解决方法是把资源（这里的文件句柄）作为某些类的对象，类的构造函数请求资源，然后类的析构函数释放它。比如，我们可以定义一个File_ptr类，其作用跟FILE*一样，如下

```
class File_ptr {
    FILE* p;
public:
    File_ptr(const char* n, const char* a)
    {
        p = fopen(n,a);
        if (p==0) throw Failed_to_open(n);
```

```
    }
    ~File_ptr() { fclose(p); }
    // copy, etc.
    operator FILE*() { return p; }
};
```

我们可以通过fopen返回的参数去购造一个File_ptr对象。File_ptr对象将在它的域的最后被析构，析构函数则关闭文件。我们的程序现在缩短成这样：

```
void use_file(const char* fn)
{
    File_ptr f(fn,"r"); // open file fn
    // use f
} // file fn implicitly closed
```

析构函数被调用与函数是由于正常退出或由于抛出一个异常而退出没有关系。问题的一般形式看起来是这样的：

```
void use()
{
    // acquire resource 1
    // ...
    // acquire resource n
    // use resources
    // release resource n
    // ...
    // release resource 1
}
```

这可以应用任何“资源”，任何你获取然后必须释放给系统，以便使其正确工作的东西。例子有文件、锁、内存、iostream状态和网络连接。请注意到自动废料收集不是它的替代：及时释放资源是是很重要的，并且对性能有益的。这是一种对资源管理的系统的方法，修改后的代码比原来有错误的和原始的方法的代码更短和更简单。程序员不用记住去释放资源。这跟以前曾经流行的finally方法相比也不一样，程序员提供一个try块去释放资源。C++处理这种问题的解决方法象下面这样：

```
void use_file(const char* fn)
{
    FILE* f = fopen(fn,"r"); // open file fn
    try {
        // use f
    }
    catch (...) { // catch all
        fclose(f); // close file fn
        throw; // re-throw
    }
    fclose(f); // close file fn
}
```

这种解决方法的问题是它很冗赘的、乏味和可能代价巨大。通过在语言中提供 finally语句，它能变得不那么冗长和不那么乏味，正如Java和C#以及早期的语言所做的那样[92]。然而，缩短

这样的代码没有涉及到基本的问题，即程序员必须记住去写释放每个已经获得的资源的代码，而不是只获取资源，象RAII方法所做的那样。当我发现“在初始化时获得资源”是对finally方法的一种系统的和更小错误倾向的替换方法后，半年之后把异常引入到ARM并在会议论文中提到异常[79]。

注意到RAII技术—象其它强大的C++习惯用语，比如智能指针—依赖于析构函数的使用。析构函数是1980年第一批被引入到带类C中的特征（§2.1）中的一个。它们的引入是由于对资源管理的关注所激发。构造函数申请资源并初始化；析构函数则执行相反的动作：它们释放资源和做清理工作。因为释放和清除必须被保证，对于分配在栈上（静态分配的变量也是）的对象，析构函数被隐式调用。因此，现代C++资源管理是在带类C的第一个版本的基础上发展起来的，而这又根源于我早期的在操作系统上的工作[121]。

5.4 自动废料收集

1995年的某个时候，我逐渐理解到委员会中的大多数人认为把废料收集器加入到C++编程是不符合标准的，因为收集器会不可避免地执行某些动作，而那将违反标准规则。更坏的是，他们在自动废料收集器将打破规则这一点上是正确的。比如：

```
void f()
{
    int* p = new int[100];
    // fill *p with valuable data
    file << p; // write the pointer to a file
    p = 0; // remove the pointer to the ints
    // work on something else for a week
    file >> p;
    if (p[37] == 5) { // now use the ints
        // ...
    }
}
```

我对上面代码的看法—不管口头上的还是书面上的—都是相当粗暴的：“这样的程序应被扔掉”、“使用保守的废料收集器对C++来说非常好”。然而，那不是草稿标准说的。废料收集器在我们读从文件中获取数据的指针和再次使用整数数组之前无疑会回收内存。然而，标准C和标准C++，绝对不允许内存在没有程序员的显式的动作下被回收！

为解决这个问题，我做了个提议，允许“可选的自动废料收集器”[123]。这会把C++带回到我曾经考虑过的C++该是什么样子的境地并让已经在被使用的废料收集器遵守标准[11，47]。在标准中明确的提出这个将会鼓励GC的恰当的使用。不幸地，我严重低估了委员会对废料收集器的厌恶，并且错误地提交了提议。

我的GC提议的致命失误是混淆关于“可选择的”的含意。“可选择”真的意味着编译器不是必须提供一个废料收集器？它意味着程序员能够决定是否打开或关闭废料收集器？是否要在编译时和运行时做出选择？如果我要求激活废料收集器去工作但编译器不提供时发生什么事？我能否在运行时请求进行废料收集？我怎么确定废料收集器没有在关键操作的时候运行？那时对于这样的问题的混乱的讨论爆发了，不同的人们发现许多的冲突的答案，提议被否决。

事实上，就算当时没有被搞糊涂，废料收集器也不会在1995年获得通过。部分委员会成员：

- 出于性能的原因对GC强烈地不信任
- 厌恶GC因为它看起来与C不兼容
- 没有感觉到他们理解接受GC所带来的含义（我们也没有）
- 不想创建一个废料收集器

• 不想为废料收集器付出代价（金钱、空间和时间等）
• 要GC风格的另一种替代
• 不想花费委员会宝贵的时间在GC上

基本上，在标准进程中引进某些主要的东西是太迟了。为了让与废料收集器有关的东西能被接受，我应该提前一年开始。

我的关于废料收集器的提议反映了后来在C++中主要使用的废料收集器—即保守的收集器，没有关于哪些内存位置包含指针和决不在内存中移动对象的假设[11]。变通办法包括创建一个类型安全的C++子集，这样就能知道每个指针指向哪里，使用智能指针[34]和提供一个单独的操作符（gcnew或new(gc)）用于分配在废料收集堆的对象，这三种方法都是可行的，各带不同的好处，并且都有各自的支持者，进一步让C++中标准化废料收集器的工作变得复杂。

多年的一个共同的问题是：为什么你没有增加GC到C++？通常，这意味着（后续评论）C++委员会怎么无视这本该被做而尚未开始做的工作。首先，我观察到在我以前的思考中有这样的看法，当C++第一次设计时如果它依赖于废料收集器则C++将胎死腹中。废料收集器的时间开销，在可供使用的硬件上，在靠近硬件和性能关键的领域都排除废料收集器的使用，而那是正是C++的面包和黄油。带有废料收集的语言，比如Lisp和Smalltalk，人们对那些适合他们的应用程序有理由感到高兴。我的目标不是在它们已经确立的领域上让C++去取代它们。C++的目标是让面向对象和数据抽象机制在那些技术已被认为不现实的领域变得可以使用。C++的主要使用领域涉及任务，比如器件驱动，高性能计算和硬实时任务，而废料收集器在这些领域要么不可行要么不常被使用。

一旦C++被制定为没有废料收集并带有一组使得废料收集变得困难（指针、转换、联合等）的语言特征，那么将很难在不造成较大损害的情况下去重新改造。而且，C++提供的许多特征使得废料收集在许多领域没有必要（域对象、析构函数、定义容器和智能指针的设施等）。这些都让废料收集器变得不紧迫。

所以，为什么我想看到C++支持废料收集器呢？实际的理由是许多人写代码时随意地使用自由存储，在一个数十万行代码的程序中，满世界都是new和delete，我看不到避免内存泄露和限制非法指针访问的希望。我对那些开始一个工程的人们的建议是简单的：“不要那么做”。通过使用容器（STL和其它§4.1)、资源句柄（§5.3.1和智能指针(§6.2)）和智能指针（如果需要的话）就能避免那些问题，写出正确且高效的C++代码的方法是相当容易的。然而，我们中的许多人不得不处理老的代码，它们对内存的处理方法是随意的，对于这样的代码，增加废料收集器通常是最佳选择。我希望C++0x要求每个C++编译器都要有一个废料收集器，它能够被激活或关闭。

我怀疑废料收集器最终会变成必需品的另一个理由是，我还没有看到怎么获得完美的类型安全而不需要—至少不需要普遍深入的测试指针合法性或有害的兼容性（比如，通过使用两个字的非局部指针）。并且改进类型安全（比如，“消除每一个隐式类型违例”[121]）总是C++的一个长期的基本的目标。显然，这与通常的“当你有一个废料收集器，编程将变得简单因为你不需要考虑回收”理由很不同。与之对比，我的观点总结为“C++是一个可以应用废料收集的语言，因为它创建许多小的废料需要被收集。”许多关于C++的思考已经关注到一般的资源（比如锁、文件句柄、线程句柄、和自由存储内存），正如§5.3所备注的，这个关注从语言本身转移到标准库，再到编程技术。对于大的系统，对于嵌入式系统，和对于安全关键的系统，一种系统的对待资源的方法对我来说比关注废料收集器更有前途。

5.5 没有完成的工作

选择做什么比怎么完成那些工作更重要。如果有人—在本文是指C++标准委员会委员—决定在一个错误的问题上进行工作，工作完成的质量大多是不相关的，给定有限的时间和资源，委员会只能处理少数主题并且只能在少数的主题上努力工作而不是更多的主题。大体上，我认为委员

会的选择是正确的，并且C++98已经被证明是一个明显的比ARM C++好的语言。自然，我们可以做得更好，但是回顾起来，仍是难以知道怎样才能做得更好。那时候所做的决定，都带有许多我们可以获得的信息（关于问题、资源、可能的解决方法、商业现实等），那今天的问题又有哪些呢？

尽管有些问题很显然：

- 为什么我们没有增加对并发的支持？
- 为什么我们没有提供更多有用的库？
- 为什么我们不提供一个GUI？

这三个问题都被认真考虑过，并且前两个是通过投票决定的。投票的结果基本上是一致通过的。考虑到我们已经决定不再追求并发或提供一个更大的库，那么关于GUI库的问题就没有讨论的意义了。

我们中的许多人—也许是大多数的委员会成员—想要某种并发的支持。并发是基本的和重要的。在其它建议中，我们有一个漂亮的坚固的提议，关于以micro-C++的形式支持并发，提议来自University of Toronto[18]。我们中的某些人，比如Dag Bruck依赖于来自爱立信的数据，察看该事件并向委员会提出不处理并发，理由如下：

- 我们发现许多替代的方法用于支持并发
- 不同的应用领域显然有不同的需求和完全不同的传统
- Ada就是直接支持并发的语言，它的经历让人失望
- 许多（而不是全部）能够由库来完成
- 委员会没有足够的经验去完成一个坚固的设计
- 我们不想有一组对并发方法的支持超过了其它方法的原语
- 我们估计工作—如果完全可行—将耗费多年，考虑到我们短缺的资源
- 我们没有能力去决定应该抛弃用于改进并发工作的其它想法

我估计那时“并发跟其它扩展一样是一个大的主题，我们考虑把它们放到一起”。我不记得我听说过我认为是“辩论终结者”的东西：任何充分的并发支持都会牵涉到操作系统；因为C++是一个系统编程语言，我们需要映射C++的并发设施到系统提供的原语上。但对于每个平台的所有者都坚持C++程序员能够使用每一个他们提供的设施。除此之外，作为多语言环境的基本部分，C++不能依赖于并发模型，因为这与其它语言支持的明显不同。设计问题受到太多约束以致没有解决方法。

缺少并发支持并没有伤害到C++社团，因为人们处理并发是相当现实的，够通过库的支持来实现，并且有少数的语言扩展（非标准的）。不同的线程库（§8.6）和MPI[94][49]提供了许多例子。

今天，折衷方案变得不同了：硬件门数的持续增长，但硬件时钟速度没有增长，这两个是利用低层次的并发的一个强烈的诱因。除此之外，多处理器的增长和集群要求其它风格的并发支持，广域网的增长和web从本质上说也是另一种风格的并发系统。支持并发的挑战比以往都要紧急，C++世界中的主要玩家们看起来对适应改变的要求很开放。C++0x上的工作就能反映出这一点（§8.2）。

对“为什么我们没有提供更多有用的库”的回答是简单的：我们没有资源（时间和人力）去做比我们所能做到的更多的工作。甚至为了获取与其它部分的一致性，STL因此延迟1年，比如iostream，就让委员会劳累很久。明显的捷径—采用商用的基础库—也被考虑过。在1992年，TI提供他们的非常漂亮的库用于考虑，在一个小时内，有5个主要公司的代表清楚地表达了他们的观点：如果这个库被认真考虑的话，那么他们将推荐他们自己公司的基础库。这不是委员会所能接受的。对一致性要求不是很高的委员会也许会通过从商用库中选择一个来达到目的，但对于C++委员会来说，这是不可能的。

1994年同样也是必须被记住的，许多人已经认为C++标准库太大了。Java还没有改变程序员的观点，即他们可以从一门语言中免费得到某些东西。反而，C++社团中的许多人使用微小的C标准库作为库大小的比较单位。一些国家代表（荷兰和法国）反复表达他们对C++标准库的膨胀的担忧。象委员会中的大多数人一样，我也希望标准会对C++工业库有所帮助而不是取代它们。

考虑到那些对库的一般的关注，对“为什么我们没有提供一个GUI库”的答案也就很显然了：委员会不能完成它。甚至在处理GUI特定的设计之前，委员会必须处理并发和决定容器设计。除此之外，许多成员并没有准备。GUI被看成是一个又大又复杂的库—使用dynamic_cast（§5.1.2）—就能由自己来完成（特别地，那正是我的观点）。事实上他们也确实这样做了。今天的问题不是没有C++的GUI库，而是有太多库，大约有25种类似的库在被使用（比如，Gtkmm[161]，FLTK[156]，SmartWin++[159]，MFC[158]，WTL[160]，wxWidgets（前身是wxWindows）[162]和Qt[10]）。委员会可以在GUI库上工作的，在那些被GUI库作为基础的库设施上工作，或者在用于公共GUI功能的标准库接口上工作。后两种方法也许会产生重要的结果，但那都没有被采纳，因此我不认为委员会有在那个方向上取得成功的必需的才能。相反，委员会在更精细的流IO的接口上花了大量工夫。那也许是一个死胡同，因为用于多字符集和locale的施设在传统数据流的环境中不是很有用。

6、 标准维护：1997-2005

在标准通过之后，ISO进程进入了“维护模式”差不多5年。C++委员会决定这样做因为：

- 委员会的成员累了（对于某些成员来说，工作了10年），想去做其它事情
- 社团还没有理解新的特征
- 许多实现者还不支持新的特征、库和工具
- 没有让委员们或社团感到紧急的新的伟大的想法。
- 对于许多成员，用于标准的资源（时间和金钱）太少
- 许多成员（包括我）认为ISO进程需要一个“冷却期”

在“维护模式”，委员会主要是对缺陷报告做出回应。大多数缺陷通过文字得以澄清或解决矛盾。只有少数的新的规则被引入，真正的创新被避免。稳定就是目标。在2003年，所有的这些小的修正以“技术勘误表1”的名义发表了。同时，英国委员会把握这次机会去矫正一个长期存在的问题：他们说服Wiley去发行修正版本的标准[66]。最初的和许多艰难的工作都由Francis Glassborow和Lois Goldthwaite完成，委员会的项目编辑Andrew Koenig，他创作了现行的文本，也给予了技术支持。

直到2003年修正标准出版，公众所能获取的惟一的标准是非常昂贵的（大约$200）来自ISO的纸质拷贝或来自INCITS（前身是ANSI X3）的另一种pdf版本。pdf版本在当时是一种新奇的事物。标准机关通过贩卖这些标准得到部分的财政支持，因此他们最不情愿去使得标准免费或很便宜。另外，他们没有零售的渠道，所以你在当地的书店中找不到国家或国际标准—当然除了C++标准。象C++一样，C标准现在也可得到了。

许多最好的标准工作对于平常的程序员来说是看不到的，并且是相当的深奥，当陈述出来时又是无聊的。许多工作用在寻找更清楚的表达方法和“每个人都知道，但就是用书面语言表达不出来”和解析模糊不清的事情—至少理论上—不会影响到大多数程序员。维护本身大部分就是“无聊和深奥”的事情。更进一步，委员会需要关注标准在某处自身矛盾或看起来是自身矛盾的事情。然而，这些事情对于实现者来说是十分重要的，确保一个特定的语言使用被正确地处理。反过来，这些事情对程序员来说也是十分重要，因为尽管最小心地编写代码，大的程序也会故意地或意外地依赖于某些特征，对某些人来说，那会变得难解和深奥。除非编译器实现者同意，程序员要获得可移植性将变得困难并且容易成为单一编译器提供商的人质—而那有背于我对C++应该是什么的观点。

给一个这项维护工作量的数字（不明确地开展下去）：从1998年标准完成到2006年，core和库工作组每个处理超过600个“缺陷报告”。幸运的是，不是所有的报告都是有缺陷的，但要确定那真的不是问题或这个问题只是由于在标准中没有表述，需要时间和注意。

维护并不是委员会从1997到2003的全部工作，有一个大小适度的计划用于将来（考虑C++0x），但主要工作是写一个关于性能问题[67]的技术报告和一个关于库的报告[68].

6.1 性能技术报告

性能技术报告（“TR”）[67]是由一个建议标准化C++的分支用于嵌入式系统编程而促进产生的。提议称之为嵌入式C++[40]或简单的EC++，源于日本嵌入式系统工具开发者协会（包括Toshiba，Hitachi，Fujitst和NEC），有两个主要的关注点：移除对性能有潜在伤害的语言特征和移除对于程序员来说太过复杂的语言特征（这被认为是一个潜在的效率和正确性危害）。一个没有明确阐明的目标是去定义一些比标准C++更容易实现的短术语。

在这个C++子集中禁止的特征包括：多继承、模板、异常、运行时类型信息（§5.1.2）、新风格的转换、名字空间。对于标准库，STL和locale也被禁止，并且提供了iostream的替代版本。我认为这个提议误入歧途并且是倒退的。特别地，性能代价是假想出来的，比如，模板的使用已被多次证明是对嵌入性系统的性能（时间和空间）和正确性是关键的。然而，在1996年EC++被第一次提议时，并没有多少的数据能够说明这一点。讽刺的是，今天少数使用EC++中的大多数使用“扩展EC++”[36]，即是EC++加模板。类似的，名字空间（§5.1）和新风格的转换（§5）主要用于澄清代码，并能用于降低维护的难度和验证的正确性。最好的证明（并常常被引用的）是完整的C++与EC++的比较是iostream。主要原因是C++98的iostream支持locale而老的iostream不支持。这就有些讽刺意味因为locale被加进来是为了支持不同于英语的语言（比如日语），能用于优化掉其它没有使用到的环境（[67]）。

在认真的考虑和讨论后，ISO委员会决定坚守长期以来的不支持方言的传统—尽管那只是分支。每个方言都导致使用者社团的分裂，甚至正式定义这样分支也会造成这样的破坏，当它用户开始去发展一种单独的技术、库和环境的文化。不可避免地，关于与语言相关的子集的失败故事开始出现。因此，我建议反对EC++的使用，而是使用完整的ISO标准C++。

显然，那些提议EC++的人们在想得到一个有效率的、容易实现和相对容易使用的语言，这些想法都是正确的。委员会需要示范ISO标准C++就是这样的一个语言。特别地，对于委员会来说，对被EC++出于性能关键、资源受限、安全关键的任务的原因而反对的特征的使用进行文档化看起来是一项合适的任务。所以决定去写一份关于性能的技术报告，它的实行总结如下：

“这份报告的目标是：

- 给读者一个使用各种C++语言和库特征所带来的时间和空间开销的模型
- 揭穿流传广泛的有关性能问题的流言
- 展示用于关注性能的应用程序中的C++技术
- 展示用于实现C++标准语言和库设施的技术，用于产生高效的代码

至于运行时和空间性能的关注，如果你能够在应用程序中使用C语言开发，那么你也能够用C++去开发。”

在我们需要某些高性能和嵌入式应用的场合中，不是每一个C++特征都是高效和可预测的。如果我们可以预先容易和精确地确定每个特征使用所需要的时间，那么这样的特征是可预测的。在嵌入式统的环境中，我们必须考虑如果我们能够使用：

- 自由存储区（new/delete）
- 运行时类型信息（dynamic_cast和typeid）
- 异常（throw/catch）

要完成这些操作的时间依赖于代码的环境（比如，在throw到达跟它匹配的catch之前有多少个栈需要unwinding）或程序的状态（比如，new的时序和在new之前的delete）。

用于嵌入式或高性能应用的编译器都有编译选项用于禁止运行时类型信息和异常。自由存储的使用也很容易避免。所有其它C++语言特征是可预测的并且能够被优化掉（根据0开销原则§2）。甚至异常跟其替代品[93]相比，也是高效的，并且应该能被所有的系统所使用，除了最严格的硬实时系统。TR讨论这些事情并定义一个接口到硬件的最底层（比如寄存器）。TR的工作是由一个工作组—主要由那些关注嵌入式系统，包括EC++技术委员会的成员们所组成的—编写完成的。我在性能工作组中也很积极，并起草了相当部分的TR文档，但主席和编辑首先是Martin O'Riordan，然后是Lois Goldthwaite。致谢表中有28个人，包括Jan Kristofferson，Dietmar Kühl，Tom Plum和Detlef Vollmann。在2004年，关于TR的投票被一致通过。

2004年，在TR被定下来后，来自Lockheed-Martin Aero的Mike Gibbs发现一个算法，允许dynamic_cast能够在常量时间内实现并且速度很快[48]。这就给dynamic_cast最终在硬实时编程中变得可用带来了希望。

TR的工作是少数的几个地方之一，在嵌入式系统中的广大的C++用法变成了公开的可访问的著作。使用范围从高端系统比如电信系统到底层系统，在那里完全和直接访问特定硬件特征是十分重要的（§7）。为服务后者，TR的工作包含一个“硬件寻址接口”，还有它的使用方针。这个接口主要是由Jan Kristofferson（代表Ramtex Internal和丹麦）和Detlef Vollmann（代表Vollmann Engineering Gmbh和瑞士）来完成的。为了让大家先一睹为快，这儿给出的代码的功能是拷贝一个称为PortA2_T的端口所指向的寄存器缓冲的内容：

```
unsigned char mybuf[10];
register_buffer<PortA2_T, Platform> p2;
for (int i = 0; i != 10; ++i)
{
    mybuf[i] = p2[i];
}
```

本质上以下的操作也能实现块的读操作：

```
register_access<PortA3_T, Platform> p3;
UCharBuf myBlock;
myBlock = p3;
```

注意到使用模板和使用整数作为模板的实参；对于库来说这是十分重要的，用于在时间和空间上对性能的优化维护。这对许多乐于听到由于模板所导致的代码膨胀故事的人来说是一个意外。模板通常会对每一种特化产生一份代码拷贝，即对于每一种模板参数的组合。因此，显然，如果你的代码是由模板产生的，并且占用了很多内存，那么你可以使用这些内存。然而，许多现代模板是基于这样的一个观察而写的，即内联函数有可能跟函数的前导一样小，甚至更小。这样的话，你就可以同时节省时间和空间。除了好的内联之外，从模板代码那里还能得到其它好处：没有用到的代码不会被产生和避免假指针。

标准要求除非成员函数被一组特定的模板参数所调用，否则类模板不会产生这样的成员函数。这就自动消除了所有可能变成的“死代码”。比如：

```
template<class T> class X {
public:
    void f() { /* ... */ }
    void g() { /* ... */ }
    // ...
};
int main()
{
```

```
    X<int> xi;
    xi.f();
    X<double> xd;
    xd.g();
    return 0;
}
```

对于这个程序，编译器必须产生X<int>::f()和X<double>::g()，但不会产生X<int>::g()和X<double>::f()。在我的坚持下，这个规则在1993年被添加到标准中，特别是为了减少代码膨胀，那是早些时候使用模板所观察到的结果。我看到该规则在嵌入式系统得到广泛使用，“所有类必须是模板，甚至只用到单一实例时也必须这么做”。在这种情况下，死代码就自动被消除了。

另一个需要遵循的简单的规则是为了从模板中获得好的内存性能：“不要使用指针如果你不需要它们”。这个规则保存完整的类型信息，允许优化器去更好地执行（特别是当使用到内联函数时）。这意味着把简单对象作为参数的函数模板应该是通过传值而不是传引用。注意到指向函数和虚函数的指针会破坏这个规则，会给优化器带来问题。

要获得代码膨胀，你只需要这么做：

1、使用大的函数模析（这样产生的代码就会很大）

2、使用许多指向对象的指针、虚函数和指向函数的指针

3、使用“特征非富”的类层次（产生许多潜在的死代码）

4、使用差劲的编译器和差劲的优化器

我设计模板专门是为了让它更容易地为程序员避免（1）和（2）。基于经验，标准也能避免（3），除非你违反（1）或（2）。在90年代初，（4）变成了一个问题。Alex Stepanov命名它为“抽象的惩罚”问题。他定义“抽象的惩罚”为一种比例，即运行该模板操作所需要的时间与实例化该模板操作所需要时间的比例（比如，find在一个vector<int>和一些trivial非模板的相同的操作（比如在一个int数组上的循环））。编译器处理这些简单和明显的优化得到的比例是1，差劲的编译器的抽象惩罚系数是3，尽管好的编译器实现真的做得很好。在1995年10月，为了鼓励编译器制造商们做得更好，Alex写了“抽象惩罚评测”，简单测量了抽象惩罚的数据[102]。编译器和优化器的作者们不喜欢看到他们产品这么弱，所以今天的比率大约是1.02左右。

C++支持嵌入式系统编程的其它方面—也是同样重要的方面—是它的计算和内存模型是现实世界中硬件的反映：内建类型直接映射到内存和寄存器，内建操作直接映射到机器操作，用于数据结构的组合机制并没有增加间接性或内存的开销[131]。在这一点上，C++与C一样。参见§2。

把C++看成是一种更接近机器语言并带有抽象设施，可用于表达可预测的类型安全的低层的设施，已经变成了Lockheed-Martin Aero用于安全关键硬实时代码的编码标准[157]。我帮忙起草了这个标准。通常，使用C和汇编语言程序员理解将语言设施直接映射到硬件，但那通常不能满足抽象机制和强类型检查的需求。反过来，由高级“面向对象”语言过来的程序员常常看不到需要与与硬件靠得更近的需求，并且期望一些未指定的技术去帮他们传达他们的想要的抽象而又不需要付出不可接受的开销。

6.2 TR库

1997年当我们完成标准时，我们都觉得标准库就是我们认为最迫切需要的也是我们可以提供的简单的库。还有几个也很想要的库，比如hash table、正则表达式、目录操作和线程，都缺失了。由Matt Austern（最开始和Alex Stepanov一起在SGI工作，然后在AT&T和我一起工作，现在在google工作）带领的库工作组马上为这样的一个库开展工作。在2001年，委员会开始了在库技术报告的工作。在2004年，TR库中指定的库被人们认为最迫切需要而被一致投票通过。

尽管标准库和它的扩展的重要性是非常巨大的，我在这里只能简要地列出新的库：

- 多态函数对象封装器

• Tuple类型
• 数学的特殊函数
• 类型粹取
• 正规表达式
• 增强的成员指针适配器
• 通用的智能指针
• 引用封装器
• 用于计算函数对象返回类型 的统一的方法
• 增强的binder
• hash table

在投票的当时，每一个库已经有相应的原型或相应的具有工业强度的实现。他们希望在每一个新的C++编译器中都带有这样的库。很多新库的设施很明显是“技术的”，即，它存在主要是为了支持库的创建者。特别地，它们的存在用于支持STL传统中的标准库设施的创造者。这儿，我只是强调为大多数应用构建者也喜欢的三个库：

• 正则表达式
• 通用智能指针
• hash table

正则表达式匹配是脚本语言和许多文本处理的支柱。最后，C++有一个标准库用于它。主要的类是regex，提供正则表达式匹配，与ECMA脚本模式兼容并有其它流行的符号。

“智能指针”的主体是一个引用计数指针，shared_ptr，用于需要共享所有权的代码。当指向一个对象的最后一个shared_ptr被销毁时，由它所指向的对象也就被删除。智能指针是流行的，但不普遍，因此对于它们性能的关注和可能的滥用就使得智能指针的祖先counted_ptr被排除在C++98之外。智能指针也不是万能药，特别地，它们比使用普通指针的代价高得多，对象的析构函数由一组shared_ptr所拥有，并且将在不可预测的时候运行。并且如果由于最后一个shared_ptr被删除导至许多对象立刻删除，那么你会招致“废料收集器延迟”，就好象你正在运行一个一般的收集器。这个代价主要跟计数对象所分配的自由存储有关，特别是在线程系统中对使用读数器所加的锁（“lock-free”在这儿能有所帮助）。如果它就是你想要的废料收集器，那么你最好能够简单地使用现在可用的废料收集器[11，47]或等到C++0x（§5.4）。

对于hash table则没有这样的忧虑，它们本应该在C++98中出现，如果我们有时间去做一个恰当的详细的设计和规范说明的话。毫无疑问，在用于大的表时，hash_map是map的另一种替代，其中key是字符串，我们可以设计出好的hash函数。在1995年，Javier Barreirro，Robert Fraley和David Musser尝试及时准备一份提议用于标准化，他们的工作变成了后来的hash_map[8]的基础。委员会没有足够的时间，最后在TR库中，unordered_map（和unordered_set）是8年的实验和工业应用的结果。一个新的名字，选择“unordered_map”是因为有许多hash_map正被使用。unorder_map是与hash_map实现者保持一致的结果。unordered_map是无序的是指遍历元素的迭代器并不需要保证以任何特定次序：hash函数没有定义map的<应该是以什么次序进行。

开发者们对这些扩展的最普通的反应是：“那是时间的问题，为什么它花了你们这么久？”和“我现在需要更多”。那是可以理解的（我现在也想要更多—但是我知道我没有办法获得它），但这样的陈述反应了对ISO委员会是什么及能做什么缺少理解。委员会是靠自愿者才得以运作，并要求规范中达到一致性和不寻常等级的精确的要求（参见D&E §6.2）。委员会没有足够的钱象商用提供商那样为他们的客户花在“免费”和“标准”库。

7、 C++在现实世界中的使用

关于编程语言的讨论典型地都集中在对语言特征的讨论：语言有些什么样的特征？它们的效率怎么样？更多具有启迪性的讨论集中在更难的问题上：语言怎么使用？能被怎么用？谁用它？

比如，在早期的OOPSLA，Kristen Nygaard（Simula和OOP）察觉：“如果我们创建了一门需要MIT的PhD才能使用的语言，那么我们是失败的”。在工业环境中，第一个—并且经常只有一个—问题是：谁使用这个语言？为了什么？谁支持它？可替代的语言是什么？本段从C++使用的角度展示C++和C++的历史。

C++在哪里使用？考虑到用户的数量（§1），显然它在很多地方被使用。但由于大多数使用是商业的，这很难有记录。这也是由于C++社团中缺少中央组织带来的缺陷之一—没有人系统地收集关于C++的使用信息，也没有人有接近完整的信息。为了对C++的应用范围有个印象，下面是几个例子，在主要系统和应用中，C++作为关键性的组件：

- Adobe- Acrobat, Photoshop, Illustrator, ...
- Amadeus- airline reservations
- Amazon- e-commerce
- Apple - iPod interface, applications, device drivers, finder, ...
- AT&T-1-800 service, provisioning, recovery after network failure, ...
- Games- Doom3, StarCraft, Halo, ...
- Google- search engines, Google Earth, ...
- IBM - K42 (very high-end operating system), AS/400,

...

- Intel- chip design and manufacturing, ...
- KLA-Tencor- semiconductor manufacturing
- Lockheed-Martin Aero-airplane control (F16, JSF), ...
- Maeslant Barrier- Dutch surge barrier control
- MAN B&W-marine diesel engine monitoring and fuel injection control
- Maya- professional 3D animation
- Microsoft - Windows XP, Office, Internet explorer, Visual Studio, .Net, C# compiler, SQL, Money, ...
- Mozilla Firefox- browser
- NASA/JPL - Mars Rover scene analysis and autonomous driving, ...
- Southwest Airlines - customer web site, flight reservations, flight status, frequent flyer program, ...
- Sun-compilers, OpenOffice, HotSpot Java VirtualMachine,

...

- Symbian-OS for hand-held devices (especially cellular phones)
- Vodaphone - mobile phone infrastructure (including billing and provisioning)

更多的例子，参见[137]。一些最广泛使用和最有利可图的软件产品也曾在这个列表上。无论是C++的理论的重要性和影响，它都满足了它的大多关键设计目标：它成为一个非常有用的现实的工具。它带来面向对象编程，而且最近又把泛型编程引入了主流。在应用程序的数目和应用程序的领域上，C++的使用超过了任何个体的专业范围。那证实我在D&E[121]（第四章）所强调的一般性。

这是最不幸的事实，应用程序没能触动研究者、教师、学生和其它应用程序建造者的意识，其记录是不完整的。有许多没有记载或访问不到的经验，这不可避免地歪曲了人们对现实的感觉—关于什么是新的（或认为是新的）的方向上和在出版的文章、学术论文和教科书的所描述的方向。许多可以见到的信息容易受到商业操作。这导致许多“重新发明轮子”，不理想的实践和虚构的故事。

一个普通的虚构故事是“大多C++代码只是使用C++编译器编译的C代码”，对于这样的代

码来说没有什么不对—毕竟C++编译器比C编译器能找到更多的bug—并且这样的代码是普遍的。然而，从许多商业的C++代码和与无数的开发者的交谈及从开发者的交谈中，我知道许多主要的应用软件，比如上面所提到的，使用C++是有些“冒险的”。许多开发者都提到在局部代码使用到设计类层次，STL和使用来自STL的想法。参见§7.2。

我们能否根据C++的使用领域进行分类？下面是其列表：

- 使用系统组件的应用程序
- 银行和财务（转帐、经济模型、客户交互、柜员机等）
- 经典的系统编程（编译器、操作系统、编辑器、数据库系统...）
- 传统的小的商业应用（物品清单系统、客户服务..）
- 嵌入式系统（仪器、照相机、手机、唱片控制器、飞机、电饭煲、医疗系统...）
- 游戏
- GUI—iPod,CDE桌面、KDE桌面、Windows,...
- 图形
- 硬件设计和检验[87]
- 底层的系统元件（设备驱动、网络层...）
- 科学和数值计算（物理、工程、仿真、数据分析...）
- 服务器（web服务器，大应用程序的骨干、计费系统...）
- 符号操作（几何学模型、视觉、语音识别...）
- 电信系统（电话、网络、监控、计费、操作系统...）

这只是一个列表而不是分类。编程世界抵抗有用的分类。不管怎么样，通过这个列表，有一个事实是明摆的，C++不能成为所有东西的最后目标。事实上，从早期的C++，我就已经认定在定义明确的应用领域，通用语言至多是第二好的。而C++是一门偏向系统编程的通用编程语言。

7.1 应用编程vs系统编程

考虑这么一个似是而非的论点：C++有一点偏向系统编程但大多数的程序员都在写应用程序。显然，数以百万的程序员是在写应用程序，许多用C++写他们的应用程序。为什么？为什么他们不用应用程序编程语言写应用程序呢？对于“应用程序编程语言”的定义有许多种方法，我们可以把“应用程序编程语言”定义为一个我们不能直接访问硬件，而对重要的应用提供直接和特殊的支持的概念（比如数据库访问，用户互动或数值计算）。然后，我们可以把这些应用程序认为是“应用程序语言”：Excel、SQL、RPG、COBOL、Fortran、Java、C#、Perl、Python等，不管是好是坏，当需要比较专业的领域（更安全、更容易使用、更容易优化等），C++被当作通用编程语言。

由于还不至于那么无知，我没有声称C++是接近完美（那将会是可笑的），也没有说它不需要改进（毕竟我们正在C++0x上工作，参见§9.4）。然而，C++有一个合适的职务—一个非常大的职务—那是当前其它语言无法担任的：

- 带系统编程的应用程序；通常是受资源限制
- 应用程序由不同应用领域的组件组成，以致没有一个单一的特定的应用语言能支持全部

应用程序通过它们的特殊性获得获得它们的优势，通过增加便利和降低使用的难度或潜在的危险特征。通常，那都有运行时或空间的代价。通常，简化是基于执行环境的假设。如果你正好需要一些基本的东西，那些是在设计中被认为是不必要的（比如直接访问内存或完全普遍的抽象机制）或不需要“增加的便利”（并且不能负担起它们强加的开销），那么C++就变成了一个候选者。对许多由C++创建的应用程序来说，它们都有许多组件，对这些组件的基本的猜想都是这种情况：“某些时候，我们中的大多数做与众不同的事”。

正确陈述的方法是一般机制打败了大多数应用程序的特殊用途的特征，这些应用程序不能进行独特的分类，必须与其它应用程序合作或从它们原来狭窄的小生境中演化。这是每一个语言

增加通用语言特征的原因之一，无论语言原来的目标是什么以及它所要阐述的哲学是什么。

如果从应用程序语言到通用语言的转变是容易且代价低的话，我们将会多一个选择。然而，很少有这样的情况，特别是当有性能要求或需要机器级的访问时。特别地，使用C++你可以（而不是必须）打破一些基本的那些应用程序建立起来的假设。实践的结果是当你在应用程序中需要用到系统编程或需要C++的性能关键的设施时，在应用程序的大部分中使用C++会让这些变得很方便，然后C++的高层（抽象）设施也很也用。C++提供几乎提供了能够直接在应用程序中应用的高层特征，它所提供了用于定义这些设施成为库的机制。

从历史的观点看这个分析不需要是正确的或是事实的惟一的可能解释。许多人喜欢从另外的角度看待问题。不同的观点对语言和公司的成功有不同的定义。然而，C++的设计是基于这种观点，并且这种观点引导了C++的演化。比如，参见第9章[121]。我认为这是C++在主流中开始成功的理由，并且这个理由让C++持续稳定增长，尽管在市场上不断地出现良好设计和有更好资金支持的替代品。

7.2 编程风格

C++支持好几种编程风格，有时它们也被称之为“范例编程”。不管怎么称呼，C++通常被指为“面向对象编程的语言”。但只有当歪曲“面向对象”的定义时这才是真的。我从来没有这么说“C++是一个面向对象语言”[122]，我更喜欢“C++支持面向对象编程和其它编程风格”或“C++是一个多范例编程语言”。因此，人们提到语言问题是因为它设置了期望并影响了人们看法。

C++几乎把C（C89）作为一个“子集”，支持在C中普遍使用的编程的风格。有争议的，C++通过提供更多的类型检查和更多的符号支持，比C更好地支持了那些风格。因此，许多C++代码是用C的风格编写—更普遍的—以C的风格的代码、类和类层次混在一起不会影响整体设计。这些代码从根本上是面向过程的，使用类去提供一组更丰富的类型。有时被称之为“带类的C风格”。那种风格比纯C要好得多（用于理解、调试和维护等）。然而，从历史的观点看，这样比那些使用C++设施去表达更多高级编程技术的代码无趣得多，而且通常效率比较低下。由纯粹的“C风格”完成的C++代码在过去的15年已经越来越少。

C++的抽象数据类型和面向对象的风格的使用已被讨论得够多了，这儿不再需要进一步解释（比如，看[126]）。它们是许多C++应用程序的支柱。然而，对于这些实用程序也有它们的限制。比如，面向对象编程会导致过度依赖严格的等级和虚函数。还有，当你需要在运行时选择执行某个动作时，虚函数的调用是基本还是有效率的，但它仍然是一个间接的函数调用，因此同单一机器指令相比，仍然是开销巨大的。这导致在要求高性能的地方，泛型编程变成了C++的标准选择。

7.2.1 泛型编程

有时，C++只是简单地把泛型编程定义为“使用模板”。那最多也只是一种简化。从一个编程语言特征的观点来看，更好的描述是“参数多态”[107]和基于参数选择动作和构建类型的重载。模板是一种基本的编译期机制，用于产生基于类型参数 、整数参数等的定义（类和函数）[144]。

在模板之前，C++中的泛型编程通过宏[108]、 void*和转换或抽象类实现。自然，有些仍然在被使用，有时这些技术也有优势（特别是当与模板化接口结合时）。然而，泛型编程的当前主宰形式依赖于用于定义类型的类模板和用于定义操作（算法）的函数模板。

基于参数化，泛型编程天生就比面向对象编程更规则。从多年的对主要泛型库的使用中，比如STL，可以得出这么一个主要结论，当前C++对泛型编程的支持是不够的，C++0x关注这部分的问题（§8）。

按照Stepanov（§4.1.8）的意思，我们可以定义泛型编程而不需要提及语言特征：从具体的例子提升算法和数据结构到它们最一般的和抽象的形式。这就暗示着把算法和它们对数据的访问

用模板表示，正如对STL的描述中所展示的那样。

7.2.2 模板元编程

C++模板实例机制是（当编译器限制可以被忽视，正如它们经常被忽视一样）图灵完全（比如，参见[150]）。在模板机制的设计中，我已经把目标定为完全的一般性和灵活性。一般性是由Erwin Unruh的早期的标准化模板所说明。1994在扩展工作小组会议上，他展示了一个在编译期用于计算质数的程序（使用错误信息作为输出机制）[143]，让人感到惊讶，我和其它人认为那是不可思议。模板实例实际上是一个小的编译期函数式编程语言。早在1995年，Todd Veldhuizen展示了通过使用模板怎么定义编译期的if语句和怎么使用这样if语句（和switch-语句）在多种数据结构和算法中进行选择[148]。下面是编译期的if语句还有它的一个简单使用：

```
template<bool b, class X, class Y>
struct if_ {
    typedef X type; // use X if b is true
};
template<class X, class Y>
struct if_<false,X,Y> {
    typedef Y type; // use Y if b is false
};
void f()
{
    if_<sizeof(Foobar)<40, Foo, Bar>::type xy;
    // ...
}
```

如果类型Foobar的大小小于40，那么变量xy的类型是Foo，否则是Bar。当模板参数与<false,X,Y>匹配时，if_的第二个定义是一个偏特化。这真的相当简单，但非常聪明，我记得当得Jeremy Siek第一次向我展示时我感到非常神奇。在不同的包装下，它已经被证明对于产生可移植的高性能的库是非常有用的（比如，许多Boos库[16]依赖于它）。

Todd Veldhuizen还贡献了表达式模板（expression template）的技术[147]，最初作为实现他的高性能数值库Blitz++[149]的一部分。核心思想是通过一个操作符返回一个函数对象，该对象包含最后要求的值的参数和操作，去获得编译期解析和延迟估计。在§4.1.4中operator<产生一个Less_than对象就是一个简单的例子。David Vandevoorde独立发现了该技术。

这些技术和其它利用模板实例化时的计算能力的技术，为给定情形产生完全匹配的源代码的想法提供了技术基础。它可能导致恐怖的含糊和较长的编译时间，但对于艰难的问题，同时也是优雅和高效的解决方法，参见[3，2，31]。基本上，模板实例依赖于重载和特化去提供合理的完整的函数式编译期编程语言。

关于泛型编程与模板元编程之间的差异的定义并没有达成普遍的共识。然而，泛型编程倾向于强调每一个模板参数类型必须有一个可以枚举的一组属性；即是说，它必须能够定义一个概念用于每一个参数（§4.1.8，§8.3.3）。模板元编程则并不总是那么做，比如，以if_例子来说，模板定义的选择是基于一组有限的类型参数。比如它的大小。因此，两种编程风格不是完全不同。随着对参数的要求越来越多，模板元编程混合到泛型编程中。通常，这两种风格被组合使用。比如，模板元编程能用于在一个程序的通用部分中去选择一个模板定义。

当对模板使用的关注非常强烈地放在组合和其替代方法的择抉上时，这种编程风格有时也称之为“generative programming”[31]。

7.2.3 多范例编程

对于程序员来说，由C++支持的不同的编程风格是很重要的。通常，最优的代码要求

使用4种基本“范例”中的多种。比如，我们用C++可以写经典用SIMULA BEGIN[9]实现的“画出在同一个容器中所有图形”的例子：

```
void draw_all(vector<Shape*>& v)
{
    for_each(v.begin(), v.end(), // sequence
            mem_fun(&Shape::draw)); // operation
}
```

这儿，我们使用面向对象编程风格通过Shape类层次获得运行期的多态。我们使用泛型编程用于参数化（标准库）容器vector和参数化（标准库）算法for_each。我们使用普通过程编程风格的两个函数draw_all和mem_fun。最后，调用mem_fun的结果是一个函数对象。一个类不是类层次的一部分并且没有虚函数，则可以被划分进抽象数据类型编程。注意到vector，for_each，begin，end和mem_fun是模板，当使用它时会产生最恰当的定义。

我们能概括出任何序列能一对ForwardIterator定义，而不仅仅是vector，并且使用C++0x的概念可以改进类型检查（§8.3.3）：

```
template<ForwardIterator For>
void draw_all(For first, For last) requires SameType<For::value_type,Shape*>
{
    for_each(first, last, mem_fun(&Shape::draw));
}
```

我认为这是对适当地表现多范编程一个挑战，以便它足够易用，从而为大多主流的程序员所使用。这将涉及找到一个更能描述它的名字。也许它将会从增加的语言支持中获益，但那将是C++1x的任务。

7.3 库、工具包和框架

所以，当我们冲击一个C++语言显然不胜任的领域时，我们能做什么？标准的答案是：创建一个库支持应用程序的概念。C++不是一个应用程序语言；它是一个有许多设施支持设计和实现优雅和高效库的语言。许多关于面向对象编程和泛型编程的谈话都是关于创建和使用库。除了标准库（§4）和TR库的组件（§6.2）外，广泛使用的C++库还包括：

- ACE—分布计算
- Anti-Grain几何学—2D图形
- Borland Builder（GUI builder）
- Blitz++[149]—vector库，“认为自己是一个编译器”
- Boost[16]—基于STL、图形算法、正则表达式匹配、线程、...
- CGAL[23]—计算几何学
- Maya—3D动作
- MacApp和PowerPlant—Apple的基础框架
- MFC—微软窗口基础框架
- Money++—银行
- RogueWave 库 (前STL基础库)
- STAPL[4]，POOMA[86]- 并行计算
- Qt [10]，FLTK[156]，gtkmm[161]，wxWidgets[162] -GUI库和创建者
- TAO[95]，MICO，omniORB- CORBA ORBs
- VTK[155] - 可视化

在这里，我们使用“工具包”去描述由编程工具支持的一个库。在这种意义上，VTK是一个工具包因为它包含产生在Java和Python中使用的接口工具，Qt和FLTK是工具包因为它们提供GUI

创造器。库和工具的组合是对方言和特殊用途语言的一种重要的替代[133，151]。

一个库（工具包、框架）能支持一个巨大的用户社团。这样的用户社团可能比许多程序语言的用户社团还要大，一个库能够完全主宰它们的用户的世界观。比如，2006年，Qt[10]是一个有大约7500个付费客户和150,000个开源版本[141]用户的商业产品。两个挪威程序员，Eirik Chambe-Eng和Haavard Nord在1991-1992年开始，并且在1995发布第一个商业版本，这随后变成了Qt。这是流行的桌面KDE（用于Linux、Solaris和FreeBSD）和著名的商业产品，比如Adobe Photoshop Element，Google Earth和Skype（VOIP）的基础。

对于C++的名声是个不幸，好的库大家看不到；它只是在它的用户后面默默无闻地干活。这常常让人们低估了C++的使用。然而，“当然有视窗基础类（MFC），MacApp和PowerPlan—大多数Mac和微软商用软件是由这些框架所创建的” [85]。

除了这些一般和特定领域的库之外，还有许多用途更特殊的库。它们的功能被限定到一个特定的组织或特定的应用程序。即：他们把使用库作为应用程序设计的哲学：“首先通过库扩展语言，然后用扩展的语言完成应用程序”。由Celera Genomics使用的“string库”（部分大系统称之为“Panther”），成为人类基因序列[70]工作的基础，就是这种方法的一个伟大的例子。“Panther”只是许多在生物学和生物学工程的一般领域中使用的许多库和应用程序中的一个。

7.4 ABI和环境

在更大的范围使用库并不是说没有问题，C++支持源码层的兼容性（但只提供弱链接时刻的兼容保证）。如果满足以下条件，这可以很好地工作：

- 你有所有的源代码
- 你的代码只用你自己的编译器编译
- 你的源代码中的不同部分是兼容的（比如，关于资源使用和错误处理）
- 你的代码全都是用C++写的

对于一个大的系统，上面的假设没有一个是站得住脚的。换句话说，链接就是问题的来源。这个问题的根源在于基本的C++的设计决定：使用已有的链接器（D&E §4.5）。

它并不保证符合语言定义的两个C编译单元在经过不同的编译器编译后，能够被正确地链接。然而，对于每一个平台，都达成一个ABI协议（应用程序二进制接口），对于所有的编译器，寄存器的使用、调用惯例和对象布局是一样的，因此C程序员就能正确地链接。C++编译器同样也使用这些函数调用的惯例和简单的结构布局。然而，传统的C++编译器提供商抵制用于类层次的布局、虚函数的调用和标准库元件的链接标准。这导致的结果是一个合法的C++程序可以正常工作的前提是每一部分（包括所有的库）必须被同一个编译器所编译。在同一平台，比如Sun和Itanium(IA64)[60]，C++的ABI标准是存在的，但只有“使用单一编译器或通过C函数和structs进行通信”是惟一的一条能生存的规则。

坚持只用一个编译器能保证在同一个平台上的链接兼容性，在提供多平台的可移植性上，它也是一个有价值的工具。通过一致地使用同一编译器，你会得到“bug兼容性”和向所有由提供商所支持的平台移植程序。比如微软平台，微软C++提供这样的一个机会；对于更多的平台，GNU C++是可移植的；对于不同的平台，使用者会惊喜地发现，当他们的本地的编译器使用一个EDG（Edison Design Group）前端时，这会让他们的源代码具有可移植性。当每一个C++编译器都通过一系列升级，部分地增加与标准的一致性，去适应平台的ABI，去改进性能、调试和通过IDE进行集成等，在这些时期内，会让问题变得模糊不清。

与C的链接兼容性产生许多问题，但也产生了一些重大的好处。Sean Parent（Adobe）观察到：“我看到C++成功的一个理由是它与C“足够近”，平台提供者能够提供单一与C++兼容的C接口（比如Win32 API或Mac Carbon API），许多库提供了与C++兼容的接口因为这样的代价是不大—相反，如果提供一组用于Eiffel或Java语言的接口，那将是一个巨大工作，提供语言兼容性的工作量远远超过了保持与C一样的链接模式的工作量。

显然，人们尝试过很多种方法去解决链接器的问题。平台ABI就是解决方法之一。CORBA是一个与平台和语言无关（或几乎无关）的、到目前为止被广泛使用的解决方法。然而，看起来C++和Java才是被CORBA使用的语言。COM是与微软平台无关并且是与语言无关的解决方案（或几乎无关）。Java是基于这样的一种观察而产生的：获得平台与编译器的独立性；JVM通过消除语言无关和完全的特定链接解决了这个问题。微软的CLI（普通语言框架）通过要求所有的语言支持类似于Java的链接、元数据和可执行模型，从而解决了语言无关的问题。基本上所有的这些解决方法都是通过提供一个统一平台而达到提供平台无关性：在一台新机器上或操作系统上使用，你需要移植一个JVM或ORB等。

C++标准没有直接提出平台不兼容的问题。使用C++，平台无关性是通过提供与平台相关的代码（典型的是依赖于条件编译—#ifdef）。这通常是杂乱和ad hoc。但高度的平台无关性是可以实现的，通过把与平台相关的依赖局部化—也许是放在单一的头文件中[16]。在任何情况，实现平台独立的服务是有效的，你需要一种语言，能够利用不同硬件和操作系统环境的独特性。通常这个语言是C++。

Java、C#和许多其它语言依赖于元数据（即，数据的类型是跟语言有关的）和依赖于这些元数据所提供的服务（比如把一组对象传送到另一台计算机）。此外，C++只有一个最低要求。惟一的“元数据”就是RTTI（§5.1.2），用于提供类的名字和它的基类列表。当讨论RTTI时，一些我们梦寐以求的工具会提供更多系统需要的数据，但这些工具并没有变得公共、通用或标准。

7.5 工具和研究

从1980年代后，C++开发者得到C++世界中存在的许多分析工具、开发工具和开发环境的支持，比如：

- Visual Studio（IDE:微软）
- KDE（桌面环境；自由软件）
- Xcode（IDE；Apple）
- lint++（静态分析工具;Gimpel软件）
- Vtune（多层性能优化;Intel）
- PreFAST（静态源码分析；微软）
- Klocwork（静态代码分析；Klocwork）
- LDRA（测试；LDRA Ltd）
- QA.C++（静态分析；编程研究）
- Purify（内存泄露查找器;IBM Rational）
- Great Circle（废料收集器和内存使用分析器；Geodesic,然后是Symantec）

然而，工具和环境对于C++来说都是其弱项。问题的根本在于分析C++很困难。语法对于任意的N不是LR(N)。那显然是很荒谬的。问题产生是因为C++是直接基于C的（我从Steve Johnsom那里拿来了基于YACC的C句法分析器），C是“已知的”不能由LR(1)语法表达的，直到1985年Tom Pennello发现怎么去实现它。不幸的是，到那时起我已经定义C++以这样的方式工作，Pennello的技术不能应用到C++，因为有太多的代码依赖于非LR语法。句法分析的另一个方面问题是在C预处理器中的宏，它们保证当看到一行代码时，程序员所看到的—经常—显著地与编译器进行句法分析和类型检查时所看到的不同。最后，高级工具的创造者同样也面临着名字查找、模板实例和重载解析的复杂性。

通过组合，这些复杂性挫败了许多去建造用于C++的工具和编程环境的尝试。这个结果就是太少软件开发和源代码分析工具被广泛使用。还有，创造工具的代价通常很高。这导致学术实验比相对容易分析的语言实施要少一些。不幸的是，许多人都带有这么一个态度“如果它不标准，那么存在就没有意义”或者是“如果它需要很多金钱，那么它就没有存在的价值”。这导致了对已存在的工具缺乏了解和充分利用，导致更多的挫折和浪费更多的时间。

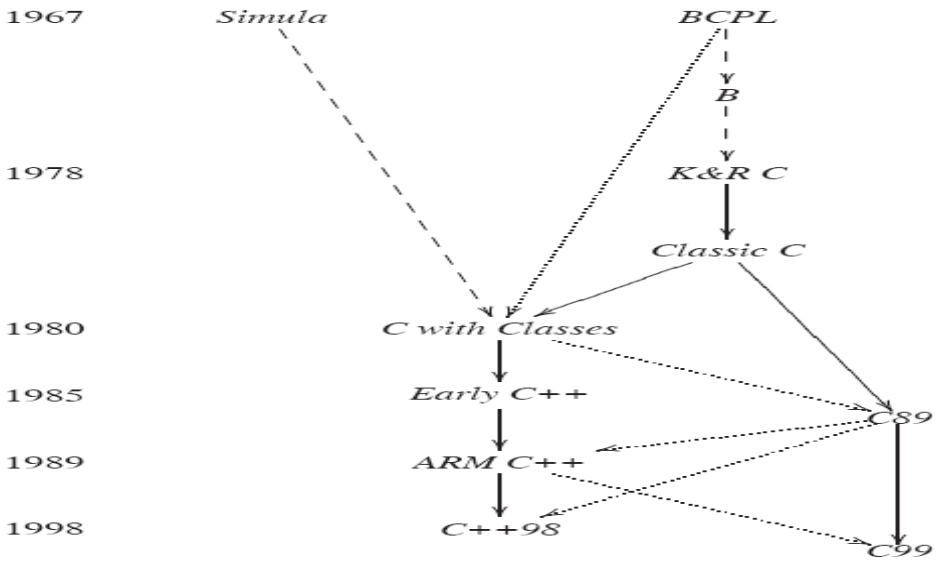
间接地，句法分析问题已经让C++在那些依赖于运行时刻信息的领域（比如GUI创造者）变得软弱，由于语言没有要求它，编译器一般不会产生任何形式的元数据（超出RTTI的最小要求；§5.1.2）。我的观点（在原来的RTTI论文和D&E上有阐述）是工具能产生用于特定应用程序或特定领域所需要的类型信息。不幸的是，句法分析问题妨碍了它。工具产生元数据方法已经成功地用于数据库访问系统和GUI，但代价和复杂性让这种方法不能广泛地使用。特别地，由于学生（或教授）没有时间去进行重大的基础建设，使得学术研究再次遭受打击。

对性能的关注也降低了工具的使用范围和数量。象C，C++是被设计用于保证最小的运行时和空间开销。比如，标准库vector默认的就没有范围检查，因为你可以基于不带范围检查的vector创建一个优化的带范围检查的vector，但你不能在一个带范围检查的vector上创建一个不带范围检查的vector（至少不可移植）。然而，许多工具依赖于额外的动作（比如数组的范围检查或确认指针合法）或额外的数据（比如用于描述数据结构布局的元数据 ）。C++社团比C社团更加关注于性能，通常这是一件好事，但它对工具创建有一定的限制作用，通过强调最小化，甚至当没有严重性能问题时也是如此。特别地，没有理由C++编译器不能提供好的类型信息用于工具创建者[134]。

最后，C++是自己成功的一个受害者。研究者不得不与公司（有时是正确的）竞争，认为公司有钱去制造研究者想要制造的各种各样的工具。关于性能有一个奇怪的的问题：C++太有效了以致那些真的想获得性能的研究，没能获得成果。这导致许多研究者转战到那些没有效率的语言上去，消除语言中没有效率的东西。消除虚函数的调用就是一个例子：通过对其它面向对象编程语言进行改进，就可以获得大的性能提升。原因是C++的虚函数调用是非常快，一般C++在时间关键的操作中都使用非虚函数调用。另一个例子是废料收集器。这个问题是一般C++程序员不会产生太多的废料和基本操作是很快的。那使得修复一个废料收集器的开销看起来比较没有吸引力，而对于比较没有效率的基础操作和更多废料的语言，则会有大比例的性能提升。再一次，其后果是C++在研究和工具方面较弱。

7.6 C/C++兼容性

C是C++最接近的语言，对C具有高度的兼容性总是C++的设计目标。在早期，兼容性的主要原因是去共享基础组织和保证完全一致（§2.2）。稍后，兼容性被认为很重要是因为在应用领域和程序员群体与C有巨大的交叠。在80、90年代 对C++作了许多小的改变，目的是把C++与ISO C靠得更近[62]（C89）。然而，在1995-2004，C同样也在演化。不幸的是，C99[64]比C89对C++的兼容性差一些，并且更难同存。参见[127]，对C/C++关系的有详细讨论。下图是C各种版本与C++的关系图：



“古典C”就是大多数人认为的K&R C，但由于由[76]定义的C缺少结构拷贝和枚举。ARM C++是由ARM定义的C++，那是大多C++标准之前的C++的基本。到目前为止基本观测报告是C（C89或C99）和C++（C++98）是兄弟（带类的C为他们共同的祖先），而不是传统的观点，认为C++是C的某种不兼容的方言。这是一个重要事件因为人们一般宣布每一种语言都有单独演化的权力，除了ISO C++，所有人采纳了ISO C所采纳的特征—而不管C的单独演化和C委员会接纳的那些与C++已经提供的相似但不兼容C++的特征的趋势。从长长的列表[127]中挑出几个例子：bool、inline函数和complex数字。

7.7 Java和Sun

我不愿意拿C++跟其它编程语言进行比较。比如，在D&E[121]中“致读者”中，我写到：几个审稿人都要我做一些C++语言与其它语言的比较。关于这个我已经决定不做了。在此我重申自己长期的且强烈持有的一个观点：语言的比较很少是有意义的，更少是公平的。对于重要语言做一个很好的比较需要付出许多精力，实际上大大超出了大部分人所愿意付出的，超出了他们所具有的在广泛应用领域的经验。为此还需要严格地维持一种超然的不偏不倚的观点和一种平和的理性。我没有时间，而且作为C++的设计者，我的公正将永远不能得到足够的信任...更坏的是，当一个语言比另一个更出名时，会发生这种看法：知名的语言的缺陷被认为是次要的，并且有简单的解决方法可以弥补，然而另一个语言的类似的缺陷，则被认为是基础的。通常，在不那么知名的语言中的解决方法不为人所知且被认为是不能满足要求的，因为那不能在他们熟悉的语言中工作...因此，我不对C++之外的其它语言进行一般和特定的评论。

然而，已经有许多关于C++/Java关系的说法，Java在市场中的存在已经影响到了C++社团。因此，一些评论不可避免，尽管恰当的语言比较不在本文的考虑范围之内，甚至在C++的定义中也没有Java的踪影。

为什么不？从Java早期，C++委员会总是包含那些有许多Java经验的人：使用者、实现者、工具构建者、和JVM实现者。我认为在基础层面上，Java和C++是在传达想法时有很多不一样的东西。特别地：

- C++依赖于直接访问硬件资源，去获取它的许多目标，然而Java依赖于虚拟机，从而让它远离硬件
- C++只带有限的运行时刻的支持，而Java依赖于许多元数据
- C++强调与用其它语言写的代码的互动，并且共享同一系统工具（比如链接器），而Java的目标是通过将Java代码与其它代码隔离，从而达到简化的目标。

C++和Java的“基因”很不一样。Java和C++之间的语法相似总带有欺骗性。作为一个类比，我注意到其它与英语很近的语言，比如法语和德语，很容易采用“结构化元素”，但对于日语和泰语来说，那就要困难得多。

Java在许多公司支持和市场（更多的目标是非程序员）中，进入到编程领域。根据Sun的关键人物（比如Bill Joy），Java是作为C++的改进和简化。“如果还有Bjarne没有设计的东西，那么都必须与C兼容”—仍然很让人惊异—一句经常被听到的语句。Java却不必那么做；比如，在D&E §9.2.2，我概述出用于C++的基本设计准则：一个比C++更好的语言将意味着什么？考虑第一位的决定：

- 使用静态类型检查和Simula类似的类
- 分离语言与环境间的关系
- C源代码兼容性（“尽可能地近”）
- C链接和布局兼容（“真正局部变量”）
- 不依赖于废料收集

我仍认为静态类型检查是好的设计和运行时效率的本质。如果让我重新设计一种新的语言用于今天C++语言所完成的工作，我还是会遵从Simula的类型检查和继承模型，而不是Smalltalk或Lisp

的模型。正如我多次所说的那样，“如果我想模仿Smalltalk，我会创建一个更好的仿制品，Smalltalk是周围最好的Smalltalk，如果你想要Smalltalk，使用它”。

我认为今天我能够更清楚地表达它，但本质是一样的；这些标准是定义C++使之成为一个系统编程语言，我不会放弃。对于那段话，今天看起来跟Java更有关系。有内建数据类型和操作符直接映射到硬件的设施，并能从本质上利用所有的机器资源就是隐式地与“C兼容”。

C++并不满足Java的设计准则；它也不想满足。类似的，Java也不满足C++的设计准则。比如，考虑一对语言-技术准则：

- 对用户定义类型提供与内建类型一样好的支持
- 不给C++以下的底层语言生存空间（除了汇编）

许多不同能被归结为C++为了使自己成为系统编程语言，并在最底层有能力以最低开销处理硬件。Java把自己的目标定为成为一种应用程序语言（对于某些定义的“应用程序”）。

不幸的是，Java支持者和他们的市场机器不是宣扬Java的本质，而是对Java与其它语言（大多指C和C++）做了许多不公的比较。正如2001年晚期，我听到Bill Joy声称（可能是在技术介绍时口头上说的）说“可靠的代码不能用C/C++写，因为它们没有异常”（参见§5，§5.3）。我认为Java是Sun用于对付微软的商业武器，但却误伤了旁观者：C++社团。它同样伤害了许多更小的语言社团；比如Smalltalk、Lisp、Eiffel等。

尽管有许多Java不会取代C++的承诺（“Java会在两年内完全干掉C++”是1996我参加一个图形发布会时反复听到的）。事实上，C++社团在Java第一次出现之时到已经翻了三倍。Java没有真的干掉C++，然而，通过转移需要用于工具、库和技术工作的精力和资金，伤害了C++社团。另一个问题是Java鼓励一个狭隘的“纯面向对象”的编程观点，过分强调运行时的解析和轻视静态类型系统（只有在2005年Java引进了“generics”）。这导致许多C++程序员模仿写了许多不优雅、不安全和效率低下的代码。当这种狭隘的观点影响到教育，并在学生中引起对不熟悉的事物的恐惧时，这个问题就变得特别严重。

正如当我第一次听到有关Java的简单和性能的吹嘘时所预测的那样[136]，Java迅速添加新特征—有些是与C++一样的。新的语言总自称是“简单”并对现实世界的应用程序是有用的，然后它们在大小和复杂度上不断增加。不管是Java还是C++都不能摆脱这些效应。显然，Java已经在效率上取得了很大的进步—考虑到它的最开始的缓慢，它当然会取得很大的进步—但当抽象被严重使用时，Java的对象模型抑制了它的性能（§4.1.1，§4.1.4）。基本上，C++和Java在目标、语言结构和实现模型上的不同比大多数人所认为的要大得多。有一种观点认为Java是另一种受限的Smalltalk版本，在它的运行时刻模型之后是一个类似于C++的语法和静态的类型检查接口。

我的猜测是Java能与C++进行比较的真正力量在于它的二进制兼容， Sun事实上已经通过对Java虚拟机的定义，控制了链接器。这给了Java链接兼容性，那是C++社团所没有的，因为使用基本的系统链接器和C++提供者没有在关键平台上达成一致的协议（§7.4）。

在1990初期，Sun基于Mike Ball和Steve Clamage的Taumetric编译器，开发了一个良好的C++编译器。当Java发布后，Sun大声宣布支持Java方针，其中C++代码被称之为“合法的代码”但由于“污染”需要重写。C++受到一些打击。然而，Sun从不动摇它对C++标准工作的支持，并且Mike、Seve和其它人都做了重大的贡献。从技术上说，人们必须接受技术现实—比如事实上许多Sun用户和许多Sun工程依赖于C++（全部的或者部分地）。特别的，Sun的Java的实现者，HotSpot，本身就是一个C++程序员。

7.8 微软和.Net

微软是当前软件开发中的巨无霸，它与C++有着各种各样的关系。他们在80年代后期第一次尝试一个基于面向对象的C，而不是C++。并且我对它在标准一致性摇摆的两种矛盾的情感有深刻印象。微软通过设置事实上的标准而不是严格遵守的正规的标准而更为人所知。然而，它们在很早之前就制造出一个C++编译器。它的主要设计者是来自以色列公司Glockenspiel的Martin

O'Riordan，他是Cfront专家，制造和维护许多移植版本。他曾经制造出一个能把错误信息以浓重的丹麦口音通过语音合成器输出的Cfront，娱乐自己和他的朋友。今天，有许多前Glockenspiel公司的以色列人在微软的C++团队。

不幸的，第一个发布并不支持模板和异常。虽然最后，这些关键特征也被很好地支持了，但那是在多年之后。第一个提供与完整ISO C++特征相近的微软编译器是1998年7月发布的VC++6.0。它的前辈，VC++5.0是在1997年2月发布，已经带了很多关键特征。在那之前，微软管理员使用高度可视化平台，比如conference keynotes，对这些特征（由他们的竞争对手所提供的，比如Borland）进行打击，说它们是“昂贵且无用的”。更坏的，微软的内部工程不能使用模板和异常，因为它们的编译器不支持这些特征。这确立了一个坏的编程实践并造成了长期的损害。

除此之外，微软是C++社团的一个负责人和常驻成员。微软过去、现在都向委员会会议派出其代表—有点迟—但提供了一个非常好的符合标准的C++编译器。

为有效地使用微软把其作为视窗特点的.Net框架，语言不得不支持象Java那样的一组设施。这暗示支持许多语言特征包括许多元数据机制和继承—由可变数组完成（§4.1.1）。特别地，一个语言习惯于产生组件，假定其它语言必须能够产生或接受这些元数据。微软的C++方言支持所有的ISO C++加“对ISO C++的CLI扩展”，口头上指的就是C++/CLI[41]，会在未来的C++世界扮演重要角色。有趣的，带C++/CLI扩展的C++是惟一一种提供访问每一个.Net特征的语言。基本上，C++/CLI是对ISO C++的一组巨大扩展，并在一定程度上提供与Windows集成，那使得如果一个程序依赖于C++/CLI特征，那将会在非Microsoft平台变得不可移植，因为它依赖于Windows平台提供的许多C++/CLI的巨大的基础设施。跟以前一样，系统接口可以被封装，并且必须被封装以保持可移植性。除了ISO C++之外，C++/CLI提供了自己循环结构，重载机制（indexer），“属性”，事件机制，废料收集堆、不同的类对象实例化语义，新形式的引用，新形式的指针，泛型，一组新的标准容器（.Net）等。

在.Net的早期（大约2000之前），微软支持一种称之为“managed C++”[24]的方言。它通常被认为是“漂亮但不中用的”（对微软的某些用户来说），并且好象是一个停止演化的东西—没有任何清晰的迹象表明它将给它的用户带来些什么样的特征。它被更复杂和更小心设计的C++/CLI所取代。

我在标准委员会工作的主要目标之一是就是防止C++分裂成各种方言。显然，对于C++/CLI，我和委员会的努力都失败了。C++/CLI被ECMA[41]标准化，作为C++的一个捆绑。然而，微软的主要产品和他们的主要客户，仍然是使用C++。在可以预见的将来，在Windows平台下，这确保有好的编译器和工具支持C++—ISO C++。那些关注可移植性的人们可以在微软的扩展周围编程去确保平台独立（§7.4）。微软编译器默认会对那些使用C++/CLI扩展进行警告。

C++标准委员会的成员很高兴看到C++/CLI是ECMA的一个标准。然而，尝试将ECMA标准提升到ISO标准引起了洪水般的不满。一些国家标准机关—比如UK的C++小组—公开表达严重关注[142]。这使得微软的人们去更好地文档化他们的基本设计原理[140]和更小心地分清ISO C++和微软自己的C++/CLI。

7.9 方言

显然，不是每个人都认为通过漫长的ISO标准化进程去让他们的想法融入到主流是一个好主意。类似的，一些人认为兼容性被高估了，甚至是一个坏主意。在以上两种情况中，人们感到他们能够加快这个进程或者觉得通过简单定义和实现他们自己的方言更好。一些希望他们的方言最终会变成主流的一部分；一些认为生活在主流之外对于他们的目的来说也是好的；还有少数的目标就是制造一种短期的语言用于实验用途。

对于我来说，那已经有太多的C++方言了（许多打）。这还没有坏事—尽管我不赞同方言因为他们在分化使用者社团[127，133]—但也反映了它们在限定的社团之外的影响非常有限。到

现在为止，没有一个主要语言特征是从方言中引入到C++主流的。然而，C++0x将会有所改变，引入有点象C++/CLI的恰当的域enum[138]，有点象C++/CLI的for语句[84]和关键字nullptr[139]（很奇怪，那是我对C++/CLI的建议）。

并发看起来是个极度受欢迎的语言扩展领域。下面是一些C++方言，通过语言特征从某种程度上支持并发：

- Concurrent C++[46]
- Micro C++[18]
- ABC++[83]
- POOMA[86]
- C++//[21]

基本上，计算机科学世界中的每一次潮流和衰落都会产生许多C++方言，比如Aspect C++[100]，R++[82]，Compositional C++[25]，Objective C++[90]，Open C++[26]和许多其它的。这些方言的主要问题是它们的个数。方言总是分裂一个子用户社团，使之没有足够大的用户社团，从而不能提供一个可以支撑的底层建筑[133]。

当提供商往语言中添加某些（典型的很少的）“巧妙的特征”到它们的编译器去帮助他们的用户去实现某些特定任务（比如，OS访问和优化）时，另一种方言就出现了。这些都成为移植的障碍，尽管它们受到一些使用者的喜爱。它们同时也受到一些管理者和平台狂热者的赏识，作为一种占据（lock-in）机制。几乎每个编译器提供者都这么干过；比如Borland（Delphi风格的属性），GNU（变量和类型属性）和微软的（C++/CLI§7.8）。当这些方言特征在头文件中出现时，它们是肮脏的。引爆提供商之间的为了能够处理他们竞争者的代码的战争。同时他们还提供非常重要的编译器、库和建设工具。

当一种方言变成一种完全独立的语言是不明显的，许多语言从C++那里借用了许多东西（有些承认，有些不承认）而没有任何兼容性的目的。一个近期的例子是“D”语言（最近的有着最流行的名字的语言）：“D是在1999年12月由Walter Bright构想，作为对C和C++的重新设计”[17]。

甚至Java的出发点（非常短暂）就是作为C++的一种方言[164]，但它的设计者马上决定维护与C++的兼容性限制了他们的需求。因为他们的目标包括对Java代码的100%的可移植性，限制代码风格为面向对象编程风格。在这个方面他们的确是正确的，达到任何程度的有用的兼容性是非常难的，正如C/C++的经验所表明的那样。

8、 C++0x

从1997年晚期到2002年，标准委员会小心谨慎地避免关于语言扩展的严肃的讨论。这允许编译器、工具和库实现者能够赶上标准，使得使用者能够吸收由标准C++所支持的编程技术。我第一次使用“C++0x”术语是在2000年之前，并通过出席委员会会议和其它2001年之前的会议，开始讨论“C++0x的方向”。到2003年，委员会再次考虑语言扩展。“扩展工作组”是由“演化工作组”构成的。名字的改变 （由Tom Plum建议的）是对语言特征和标准库设施的集成的重视的反映。依旧，我是工作组的主席，希望能够帮助确保一个C++视觉的连续性和最后结果的一致性。类似的，委员会成员关系显示大部分人和组织的持续参与。幸运地，也许多新面孔为委员会带来了新的利益和新的专长。

显然，C++0x的工作仍在进行，通过多年的工作，已经产生了许多投票。这些投票很重要，因为它们代表了C++社团对C++98的问题和对C++程序员所面临的当前挑战的回应。委员会原打算对语言本身的改变保持谨慎和保守，并强烈强调兼容性。这个规定的目标是去引导对标准库扩展的主要工作。在标准库，目标是更具侵略性和机会主义的。它正在被证明很难达到那个目标。像平常一样，委员会没有充足的资源。同时，人们对语言扩展太过兴奋，希望花更多的时间去和反对的人进行疏通。

进程的速度与会议的次数大致成比例。从1998年标准完成起，每年都有两次会议，还有许多的网络会议。当C++0x的最终期限到来时，这些大会议还会有几个“附加的”会议，关注于紧迫主题，比如概念（§8.3.3）和并发（§5.5）。

8.1 技术挑战

在C++0x开始构思时，C++社团所面对的技术挑战是什么？从高层次上，这个问题可以这么回答：

- 基于GUI的应用程序构建
- 分而式计算（特别是web）
- 安全

大的问题是怎么将这些转换成语言特征，库（ISO标准或非标准），编程环境和工具。在2000-2002年，我没有能够让标准委员会把注意力放在分布式计算和微软的代表经常地提起的安全问题上。然而，委员会没有考虑到那些。为了获得进步，问题被表达成一些具体用于改变的具体提议，比如增加一个（特定）语言特征去支持回调或通过一个（特定）变通办法替代臭名昭著的不安全的C标准库函数gets()。还有，C++处理挑战的传统是通过间接地改进抽象机制和库，而不是提供对特定问题的解决方法。

在我给出许多关于原则和方向的谈话后，委员会的第一个具体行动是在2001年Redmond会议中的头脑风暴。这儿我们做出了对C++0x需要的特征的“期望列表”。第一个建议是用于表达对齐限制（由P.J.Plauget和其它几个成员支持的）的可移植方法。尽管我强烈表达了对库的关注，但在大约100个建议中还是有差不多有90个是关于语言特征的。可取之处就是这些特征会让设计、实现更容易，并使用更优雅、更可移植和更有效率的库。这个头脑风暴提供了维护“期望列表”的种子，用于C++0x语言特征和标准库设施[136]。

从许多讨论和陈述中，我已经有了一个简短的用于C++0x的一般目标和设计规则的总结，看起来能被广泛接受，目标是：

- 让C++成为系统编程和库构建的更好的语言—而不是提供特别的设施用于支持特定的子社团（比如，数值计算和视窗风格的应用程序开发）
- 让C++更容易教与学—通过增加统一性，更强的保证和设施支持新手（新手永远比专家多）

经验法则：

- 提供稳定性和兼容性
- 优先使用库，然后才是语言扩展
- 只做那些会改变人们思考方式的改变
- 一般性优先，然后才是特化
- 同时支持专家和初学者
- 增加类型安全
- 改进性能和直接在硬件上工作的能力
- 适应现实世界。

这些列表作为用于提议扩展的框架基本原理是有用的。许多委员会的技术论文中已经用到它们，然而，它们只提供一小组方向和对抗关注细节的广泛趋势的微薄的平衡力。GUI和分布式计算的挑战在这儿没有直接陈述，但值得注意的“库”和“直接在硬件上工作”（§8.6）的特征就是对它的回答。

8.2 建议的语言扩展

为了对委员会2005年的工作有个一认识，考虑下面一个不完整的建议扩展列表（简短的！）：

1、decltype和auto—从表达式中推演出类型（§8.3.2）

2、模板别名—绑定某些但不是所有模板参数的方法并命名由模板偏特化所产生的结果为模板别名

3、外部模板—一种抑制在编译单元中隐式实例化的方法
4、移动语义（右值引用）—一种一般的机制用于消除多余的值拷贝
5、静态断言（static_assert）
6、long long和许多其它C99特征
7、>>（没有空格）去结束两个模板特化
8、unicode数据类型
9、可变（variadic）模板（§8.3.1）
10、概念—用于C++类型和整数值的类型系统（§8.3.3）
11、归纳常量表达式（generalized constant expression）[39]
12、初始化列表作为表达式（§8.3.1）
13、域和强类型枚举[138]
14、对齐控制
15、nullptr—空指针常量[139]
16、带范围的for语句
17、委托构造函数
18、继承构造函数
19、原子操作
20、线程局部存储
21、默认和禁止公共（common）操作
22、Lambda函数
23、由程序员控制的废料收集 [12]（§5.4）
24、类中的成员初始化
25、允许局部类作为模板参数
26、模块
27、动态库支持
28、整数子范围
29、多方法（multi-methods）
30、类名字空间
31、Continuations
32、Contract编程—直接支持前提、后置条件和其它
33、用户定义操作符.（点）
34、在string上的switch
35、简单的编译期反射
36、#nomacro—一种用于保护代码免受无意的宏扩展影响的域
37、GUI支持（比如，slot和信号）
38、反射（比如，数据结构描述用于运行时使用的类型）
39、语言层面的并发原语（而不是在库中）

依旧，提议太多，超出了委员会能够处理或语言能够吸收；依旧，接受所有好的提议不是切实可行；依旧，有许多人认为委员会通过引进无必要的复杂的特征让语言变坏，也有许多人抱怨说委员会通过拒绝接受许多必须的特征在谋杀语言。如果你把争论中夸张的成分去掉，两边都有相当程度的理由支持他们。委员会面临的平衡是很重要的。

在2006年10月，条款1-7被通过。基于提议的状态和初步的工作组投票，我的猜测是条款10-21也将被接纳。在那之外的则很难猜。建议22-25正在被发展，目标是在2007年7月投票，提议26（模块）被推迟到技术报告中。

这些条款中的大多数是在前面列出的“经验法则”的指导下工作的。那个列表比委员会收到的建议（以一种强迫（至少简要地）去考虑它们的形式）的一半还少。我从email和与用户会议搜集到的建议是它的好几倍。在我力促下，委员会在2005年春天在挪威Lillehammer会议通过投票决定停止接收新的提议。在2006年10月，又进行了一次投票决定在2007年末提交一个标准草案，让C++0x进入C++09。然而，尽管从根源上截止了新提议的来源，对入选的特征进行一致性地筛选同样也是一个实际的挑战，就跟技术挑战一样。

给一个技术挑战的量级描述，考虑一篇论文作为概念问题的一部分被首要的学术会议接受POPL[38]，分析与概念相关的其它论文则被提交给OOPLSA[44，52]和PLDI[74]。我个人认支持与并发（包括内存模型）相关的技术问题更难—而且更加本质。用于处理并发的C++0x设施将在§8.6中简短地讨论。

实际的挑战是提供正确和一致的对所有这些特征（和标准库设施）的详细规范说明。这涉及产生和检查好几百页的高水平的技术标准文档。通常，实现和实验是确保一个特征能够被恰当详述和与语言的其它特征和标准库良好互动的工作的一部分。标准规范不仅是一个实现（象提供商的文档），还是一组可能的实现（不同的提供商将提供不同的特征实现，每一个都具有相同的语义，但具有不同的工程折衷）。这意味着接受任何新的特征都是一种冒险—尽管接受的特征已经被实现和被使用。委员会成员对冒险的理解各不相同，什么样的冒险是值得的，什么样的冒险能被最好处理。自然，这是许多艰难的讨论的来源。

所有提议和所有委员会的会议记录都可以在委员会的网页[69]上获得。超过2000份文档—有些很长。1995年之前的少许论文在网上不能查到因为委员会直到1994年才开始依赖于论文。

8.3 支持泛型编程

对于语言本身，我们看到对支持泛型编程的特征的强调，因为泛型编程是使用C++所取得了由语言支持的进步最大的领域。

语言扩展支持泛型编程的总体目标是提供更加一致的设施，这样使得它有可能直接通过普通的形式去表达更大的问题。可能的最重要的扩展是：

- 通用初始化列表（§8.3.1）
- auto（§8.3.2）
- 概念（§8.3.3）

这些提议中，只有auto已经被正式投票通过。其它提议都已成熟，最初的民意投票已经开始支持它们。

8.3.1 一般初始化列表

“对用户定义类型提供跟内建类型一样好的支持” 是C++的语言-技术设计原则中（§2）的突出部分。C++98中有一个违背这条原则的东西没有被修正：C++98提供符号支持用于初始化一个（内建）数组，通过一组初始值。但对于vector（用户定义）则没有这样的支持。比如，我们能够定义一个由3个int 1、2、3初始化的数组：

```
int a[] = {1,2,3};
```

定义一个等价的vector则笨拙得多，还需要引入一个数组：

```
// 方法一
vector<int> v1;
v1.push_back(1);
v1.push_back(2);
v1.push_back(3);
// 方法二
int a[] = {1,2,3};
vector<int> v2(a,a+sizeof(a)/sizeof(int));
```

在C++0x中，这个问题通过允许用户对于一个类型定义一个“序列构造函数”，从而定义了一个初始化列表去初始化而得解决。通过对vector增加一个序列构造函数，我们允许：

```
vector<int> v = {1,2,3};
```

因为初始化的语义定义了参数传递的语义，这也将允许：

```
int f( const vector<int>&);
// ...
int x = f({1,2,3});
int y = f({3,4,5,6,7,8,9,10});
```

就是说，C++0x获得类型安全机制，用于变长的同类参数表[135]。

除此之外，C++0x将支持类型安全的可变模板参数[54][55]。比如：

```
template<class ... T> void print(const T& ...);
// ...
string name = "World"'
print("Hello, ",name,'!');
int x = 7;
print("x = ",x);
```

对于每一次的给定的一组参数类型的调用，其代价是产生惟一的实例化（通过一个模板函数），可变模板允许处理任意的非同类的参数列表。这对于tuple类型和类似的抽象特别有用，用于处理不同的列表。

我是同一初始化列表机制的主要支持者。Doug Gregor和Jaakko Jarvi设计了可变模板机制，它是用于简化库的实现而被开发的，比如Jakko Jarvi和Gary Powell的lambda库[72]和tuple库[71]。

8.3.2 auto

C++0x将支持通过变量的初值[73]推演得到它的类型。例如，我们可以改写§4.1.3的冗赘的例子成为下面这样：

```
auto q = find(vi.begin(), vi.end(), 7);// ok
```

这儿，我们推演q的类型就是find的返回值的类型，而该类型就是vi.begin()的类型；即是vector<int>::iterator。我第一次实现使用auto是在1982年，但由于与C的兼容性问题，被强迫将它从“带类的C”中拿掉。在K&R[76]（和后来的C89和ARM C++）中，我们可以在声明在省略类型，比如

```
static x;// 意思就是 static int x
auto y; // 意思就是 stack分配的 int y
```

在Dag Bruck提议后，C99和C++98都禁止“隐式int”。由于现在对于“auto q”来说没有合法的代码与之不兼容，因此我们允许它。不兼容性总是在理论上讲比较严重而实际上不是那么回事（通过对大量代码的审查证明了这一点）。但使用（未用过）的关键字auto使得我们不用引进一个新的关键字。另一个明显的选择（从许多语言，包括BCPL）是let，但每一个简短而有意义的单词已经被许多程序所使用，而长的和神秘的关键字广泛不受喜欢。

如果委员会—正如计划的那样—接受基于概念（§8.3.3）的重载并调整标准库去利用这个优势，则我们可以这样写：

```
auto q = find(vi,7);//ok
```

对于更一般的，但罗嗦地写法：

```
auto q = find(vi.begin(), vi.end(), 7);// ok
```

不令人吃惊地，考虑到它的超长久的历史，我是auto机制的主要设计者，同每一个C++0x提议一样，许多人都做出了贡献，有Gabriel Dos Reis，Jaakko Jarvi和Walter Brown。

8.3.3 概念

D&E[121]中对模板的讨论包含三整页（§15.4）用于模板参数的约束。显然，我觉得需要一个更好的解决方法—许多人也这么认为。在使用模板上，细微的错误所导致的错误信息，比如标准库算法，能惊人的冗长且缺少帮助信息。问题在于模板代码对模板参数的期望是隐式的。再次考虑find_if：

```
template<class In, class Pred>
In find_if(In first, In last, Pred pred)
{
    while (first!=last && !pred(*first))
        ++first;
    return first;
}
```

这儿，我们对In和Pred类型做了许多假设。从代码中，我们看到In必须支持带有恰当语义的!=，*和++，并且我们必须能够拷贝作为实参的In对象并返回值。类似地，我们看到我们调用的Pred带有一个参数，其类型是对In解引用的类型，并且Pred的返回值能够使用操作符!，以便能当成一个bool。然而，这些期望那在代码中都是隐式的。注意到泛型编程风格的许多灵活性来自隐式转换，用于让模板参数类型满足这些要求。标准库小心地记录这些需求用于输入迭代器（In）和预测（Pred），但编译器并不能读取这些需求。尝试这个错误并看你的编译器会说些什么：

```
find_if(1,5,3.14);// error
```

在2000的C++编译器，错误信息是都是惊人的差劲。

• 约束类

一个部分但通常相当高效的解决方法是基于我老的想法，让一个构造函数去检查关于模板参数（D&E §15.4.2）的假设，这种方法现在被广泛使用。例如：

```
template<class T> struct Forward_iterator {
    static void constraints(T a) {
        ++a; a++; // can increment
        T b = a; b = a; // can copy
        *b = *a; // can dereference
                 // and copy result
    }
    Forward_iterator()
    { void (*p)(T) = constraints; }
};
```

这定义了一个类，并且仅当T是一个前向迭代器[128]才能被编译通过。然而，Forward_iterator对象并没有做任何事情，因此编译器能够（所有的都可以）优化掉这样的对象，我们可以在定义中这样使用Forward_iterator：

```
template<class In, class Pred>
In find_if(In first, In last, Pred pred)
{
    Forward_iterator<In>(); // check
    while (first!=last && !pred(*first))
        ++first;
    return first;
}
```

Alex Stepanov和Jeremy Siek做了许多发展和普及这些技术。它们在Boost库[16]中被大量使用。

错误信息的质量的差别是惊人的。然而，对于所有编译器来说，写出约束类并给出好的错误消息是相当不容易的。

• 概念作为一个语言特征

约束类最多只是解决部分问题的方法。特别地，类型检查是在模板定义中完成。对于适当的独立的关注，检查应该依赖于声明中呈现的信息。那边，我们会遵守用于接口的通常规则，并且开始考虑真正的分离模板编译的可能性。

让我们告诉编译器我们希望从一个模板参数那里得到些什么：

```
template<ForwardIterator In, Predicate Pred>
In find_if(In first, In last, Pred pred);
```

假设我们能够表达ForwardIterator和Predicate是什么，编译器现在就能检查一个find_if的调用而不需要管它的定义。我们在这儿所做的就是去创建一个用于模板参数的类型系统。在现代C++环境中，这样的“类型的类型”被称之为概念（参见§4.1.8）。有许多方法详细说明这些概念；眼下，把它们认为是由语言直接支持的约束类，并且是一个更好的语法。概念说明了一个类型必须提供的什么样的设施，但对怎么提供这些设施一无所知。使用概念作为模板参数的类型（比如<ForwardIterator In>）是非常接近“对于所有类型In，In能够增加，解引用和拷贝”的数学详细描述，正如原来的<class T>就是数学上的“对于所有类型T”。

对于只给出find_if的声明（而不是定义），我们可以写

```
int x = find_if(1，2，less_than<int>(7) );
```

这个调用将失败因为int不支持单目运算*（解引用）。换句话说，调用将编译失败因为int不是一个ForwardIterator。重要的是，让编译器以用户的可以理解的语言去报告错误变得容易，并且是在编译期第一次看到这个调用时报告。

不幸的是，知道迭代器参数是ForwardIterator和预测参数是Predicate并不足以保证对find_if的调用可以成功编译。这两个参数类型相互作用。特别地，预测带了一个参数，那是对一个迭代器的解引用： pred(*first)。我们的目标是隔离调用对模板的完全检查和不需要查看模板定义就能对每次调用的完全检查。因此，“概念”必须有充分的表现力去处理与模板参数之间的相互作用。解决参数化概念的方法就是把模板也参数化。例如：

```
template<Value_type T,
         ForwardIterator<T> In, // sequence of Ts
         Predicate<bool,T> Pred> // takes a T;
                                 // returns a bool
In find_if(In first, In last, Pred pred);
```

这儿，我们要求ForwardIterator必须指向一个类型为T的元素，与Predicate的实参类型相同。然而，这导致模板参数类型间的过分严格的相互作用和非常复杂的模式和参数化的冗余[74]。

当前概念的提议[129，130，132，97，53]关注于直接表达参数间的关系：

```
template<ForwardIterator In, // a sequence
         Predicate Pred> // returns a bool
    requires Callable<Pred,In::value_type>
In find_if(In first, In last, Pred pred);
```

这儿，我们要求In必须是一个ForwardIterator，带value_type，能被Pred作为参数，而Pred必须是一个Predicate。

概念是编译期对一组类型和整数值的预测。单一类型参数的概念提供了一个类型系统，用于C++类型（内建和用户定义类型）[132]。

通过概念，把模板的需求放到它的参数上，同样允许编译器去捕获模板定义自身的错误。考虑这种看似有道理的pre-concept的定义：

```
template<class In, class Pred>
In find_if(In first, In last, Pred pred)
{
    while(first!=last && !pred(*first))
        first = first+1;
    return first;
}
```

使用vector或数组测试它，它会正常工作。然而，我们指定find_if应该工作在ForwardIterator，即find_if对于这类迭代器必须能工作，因此我们能够把它用在list或输入流（输入流是InputIterator吧？）。这些迭代器不能简单地向前移动N个元素（通过p = p+N）—甚至对于N==1的情况也不行。我们应该只假设++first，这不仅更简单，而且是正确的。经验表明这类错误是非常难以捕捉的，但概念让它变得微不足道，编译器能够发现它。

```
template<ForwardIterator In, Predicate Pred>
    requires Callable<Pred,In::value_type>
In find_if(In first, In last, Pred pred)
{
    while(first!=last && !pred(*first))
        first = first+1;
    return first;
}
```

operator+不是ForwardIterator所指定的操作中的一个。编译器可以检测到它并简洁地报告它。

关于对模板参数的编译器信息的影响，是使得基于参数类型属性的重载变得容易。再次考虑find_if。程序员经常抱怨当他们需要从一个容器中找东西时，必须指定一个序列的开头和结尾。另一方面，对于通用性，我们需要能够在算法中通过迭代器对表达序列。明显的解决方法是提供两种版本：

```
template<ForwardIterator In, Predicate Pred>
    requires Callable<Pred,In::value_type>
In find_if(In first, In last, Pred pred);
template<Container C, Predicate Pred>
    requires Callable<Pred,C::value_type>
In find_if(C& c, Pred pred);
```

有了它们，我们能够处理象§8.3.2的例子，也能处理依赖于参数类型的更精细的不同的例子。

这儿不是展示概念设计细节的地方，然而，正如上面所陈述的，设计呈现出一种致命的僵化：类型要求的属性的表达式通常是根据成员类型。然而，内建类型没有成员。例如，int*怎么才能成为ForwrdIterator，当用于上面的ForwardIterator被假设具有成员value_type时？更一般地说，我们怎么能够写对参数类型带精确和详细的需求的算法，并期望类型的定义者会定义类型具有所需要的属性？我们不能。我们也不能期望程序员当他们发现对类型的一个新的使用时，去改变老的类型。现实世界中，类型通常定义在对它们的使用范围未知的情况下。因此，它们通常不能满足详细需求，甚至当它们有用户需要的所有基本属性时亦是如此。特别地，int*是在30年前定义的，没有考虑过C++或STL的迭代器概念，对这些问题的解决方法是（非入侵式的）映射一个类型的这些属性到一个概念的需求上。特别地，我们需要表达“当我们使用一个指针作为前向迭代器时我们应该认为指向的对象的类型就是value_type”。在C++0x句法中，那可以表达如下：

```
template<class T>
```

```
    concept_map ForwardIterator<T*> {
        typedef T value_type;
}
```

concept_map提供从一个类型（或一组类型；这儿是T*）到一个概念（这儿是ForwardIterator）的映射，以便概念的使用者对于一个参数不会看到它的真实类型，而是映射类型。现在，如果我们使用int*作为一个ForwardIterator，则那个ForwardIterator的value_type将会是int。除这提供成员外， concept_map能够用于提供用于函数的新名字，甚至对类型的对象提供新的操作。

Alex Stepanov可能是第一个提议“概念”作为C++语言特征[104]（在2002年）的人—我数不清有多少“对模板的更好的安全检查”愿望。最初我不想采用语言的方法，是因为我害怕它会引向严格的接口（禁止独立开发的代码的组合，并且有严重的性能伤害）的方向，那跟早期的“语言支持会限制通用性”的思想一样。在2003年夏天，Gabriel Dos Reis和我分析这个问题，概述了设计理念，并且把解决办法的基本路径[129][130]都记录下来。所以，当前的设计避免那些坏的影响（例如：参见[38]），并且现在有许多人参与到C++0x的概念设计，有Doug Gregor，Jaakko Järvi，GabrielDos Reis，Jeremy Siek，Andrew Lumsdaine和我。对概念的一个实验性的实现已经由Doug Gregor完成，同时还有一个使用概念的STL版本[52]。

我期望概念成为所有C++泛型编程的中心。它们已经是许多使用模板设计的中心。然而，已存在的代码—没有使用概念—当然可以继续工作。

8.4 阻碍

我的其它优先权（与更好地支持泛型编程一起）是更好地支持初学者。提议的显著的趋势是偏向老练的使用者，他们提议并评估这些提议。帮助新学者在几个月内变成老手的简单的东西通常被忽视。我认为那是一个潜在的致命的设计偏见。除非新学者得到充分支持，只有少数人会成为老手。步步为营是最重要的。此外，许多人—十分合理的—不想成为内行；他们是并且希望是“临时的C++用户”。例如，一个物理学家使用C++用于物理计算或控制实验设备，通常对成为一个物理学家感到开心，并且只有有限的时间用于学习编程技术。作为计算学科学家，我们可以盼望人们花更多的时间在编程技巧上，但不是希望，我们应该去除采用好技术所遇到的不必要的障碍。自然，许多需要的去除“障碍”的改变是微不足道的。然而，它们的解决方法被对兼容性的关注和对语言规则的一致性关注所约束。

一个非常简单的例子是：

```
vector<vector<double>> v;
```

在C++98，这是语法错误因为>>是一个词法的标记，而不是两个>，每个关闭一个模板参数列表。正确的v的声明应该是这个样子的：

```
vector< vector<double> > v;
```

我认为这就是障碍。对于当前规则有非常好的语言-技术理由，并且扩展工作组两次拒绝了我的这个值得解决的问题的建议。然而，这些理由是语言-技术的并且是初学者（对于所有背景—包括其它语言的行家）所不感兴趣的。不接受第一个并且大多数明显的对v的声明浪费了使用者和教师的时间。对 “>>”的一个简单的解决方法由David Vandevoorde[145]提议并在2005的Lillehammer会议投票，我希望许多起小的“障碍”不会在C++0x中出现。然而，我，Francis Glassborow还有其它的人，尝试系统地消除许多频繁出现的“障碍”，看起来没有结果。

“障碍”的另一个例子是使用默认拷贝操作（构造函数或赋值）去拷贝一个带有用户定义的析构函数的类的对象是合法的。需要用户定义的拷贝操作将会消灭许多跟资源管理有关的肮脏的错误。例如，考虑一个简化的string类：

```
class String {
public:
    String(char* pp); // constructor
```

```
    ~String() { delete[] pp; } // destructor
    char& operator[](int i);
private:
    int sz;
    char* p;
};
void f(char* x)
{
    String s1(x);
    String s2 = s1;
}
```

在构造s2后；s1.p和s2.p指向同一块内存。这块内存（由构造函数分配）将通过析构函数被删除两次，可能带有灾难性后果。这种问题对于有经验的C++程序员来说是明显的，他们将提供恰当的拷贝操作或禁止拷贝。从C++的最早期起，这个问题已经被很好地记录了；在TC++PL和D&E中有两个明显的解决方法。然而，这个问题会严重地阻碍一个新学者并破坏其对语言的信任。Lois Goldtwaite，Francis Glassborow，Lawrence Crowl和我[30]提出语言层面解决方法的提议，一些版本将把它带入到C++0x中。

我选择这个例子去说明由于兼容性所强加的约束。通过对具有指针成员的类的对象不提供拷贝，问题能被解决；如果人们需要拷贝，他们将补充一个拷贝操作符。然而，那将破坏大量的代码。通常，矫正长期以来存在的问题是很困难的，特别是还有C兼容性需要考虑。然而，对于这种情况，析构函数的存在就是一种强指示器，说明默认拷贝可能是错的，所以string的例子能够可靠地被编译器捕获。

8.5 标准库

对于C++0x标准库，已规定的目标是去打造一个用于系统编程的更大的平台。2006年6月版的“标准库期望列表”，由Matt Austern维护，列出的64条提议包括：

- 基于容器的算法
- 对文件的随机访问
- 安全STL（完全的范围检查）
- 文件系统访问
- 线性代数（vector,matrice等）
- 日期和时间
- 图形库
- 数据压缩
- unicode文件名
- 无限精度整数算术
- 在库中统一的使用std::string
- 线程
- 套接字
- 对unicode的综合支持
- XML解析器和产生器库
- 图形用户接口
- 图形算法
- web服务（SOAP和XML绑定）
- 数据库支持

• 词法分析和分解

在这些库上已经进行了大量的工作，但许多被推迟到post-C++0x库技术报告。可惜，这让许多广泛想要和广泛使用的库组件没有标准化。另外，观察到人们与C++98的斗争，我同样高度希望委员会会对许多新的C++程序员心存同情，并提供支持具有不同背景（不仅仅是程序员新手和来从C转过来的人）的新学者的库设施。对最频繁要求添加到标准库的请求，我有一个低的期望：标准的GUI（图形用户接口；参见§1和§5.5）。

C++0x标准库的第一批新组件是来自TR库（§6.2）。在2006年的柏林会议中除了一个外，其它都被投票通过了。特殊的数学函数被认为对于绝大多数的C++程序员来说太过专业化了，被抽出来变成一个单独的ISO标准。

除了这些增加新库组件之外，库工作组的许多工作把关注点放在对已存在的组件的细小的改进上，改进已存在的组件的规范说明。这些细微改进的积累的效果是巨大的。

2007的计划包括检查标准库，使之能用上新的C++0x的特征。第一个也是最明显的例子是增加右值初始化[57]（主要由Howard Hinnant，Peter Dimov和Dave Abrahams完成）去有效的地改进由于广泛使用组件的性能，比如vector，有时必须移动对象。假设这些概念（§8.3.3）能够进入到C++0x，他们使用将引发库中STL部分（和其它模板组件）的规范说明的变革。类似的，标准容器，比如vector应该被增强，通过增加允许接受初始化列表的序列构造函数（§8.3.1）。归纳常量表达式（constexpr，主要是Gabriel Dos Reis和我[39]在工作）将允许我们去定义简单函数，比如在numeric_limits和bitset操作符中定义的类型的属性，那在编译期是有用的。可变模板机制（§8.3.1）显著地简化了与标准组件的接口，比如tuple，能带许多不同的模板参数。这就暗示在性能关键领域中将可以使用这些标准库组件。

在那之外，好象只有一个线程库得到足够的支持，将变成C++0x的一部分。其它组件可能会变成另一个TR库（TR2）的一部分：

• 文件系统库—平台无关的文件系统操作[32]

• 日期和时间库[45]

• 网络—套接字，TCP，UPD，组播，TCP之上的iostream等[80]

• numeric_cast—转换检查[22]

这儿提到的导致提议并很可能被投票的工作是库工作组2003-06的主要工作。网络库已被商业使用于许多地方有些时候了。库工作组刚开始由Matt Austern（首先是AT&T，然后是Apple）主持；现在是由Howard Hinnant（首先是Metrowerks，然后是Apple）来主持这项艰难的工作。

除此之外，C委员会也在采纳技术报告，这些必须被考虑（尽管从官方的ISO政策来说，C和C++是两种不同的语言）并且将最终被C++库接受—尽管他们是C风格的，不能从C++语言特征的支持中受益（比如用户定义类型、重载和模板）。这样的例子有带关联操作和函数的十进制浮点数和unicode内建类型。

所有加在一起，对于库工作组中的志愿者们来说是大量的工作，但它不是2001（§8）年我期望的“有雄心和机会主义的”政策。然而，那些尖声要求更多的人们（比如我）应该注意到上面列出的已经差不多是标准库的两倍大。TR库进程是未来的希望。

8.6 并发

并发不能单独通过库得到完美支持。另一方面，委员会仍然考虑基于语言的方法去解决并发，正如Ada和Java中的情况一样，不够灵活（§5.5）。特别地，C++0x必须支持当前操作系统的线程库，比如POSIX线程和Windows线程。

因此，在并发上的工作是由一个ad hoc工作组来完成，基于库-语言进行划分。采用这种方法去小心地说明用于C++的机器模型，考虑现代硬件体系结构[14]并提供最小限度的语言原语：

• 线程局部存储[29]

• 原子类型和操作[13]

其它的则是由线程库完成。目前的线程库草稿是由Howard Hinnant完成[58]。

“并发的工作”是Hans Boehm（HP）和Lawrence Crowl（以前在Sun，现在在Google）领导的。自然，编译器实现者和硬件提供商对这些最感兴趣和最支持这部分。比如，来自Intel的Clark Nelson重新设计了C++从C那里继承来的sequencing的概念，使得它能够让C++更加适应现代硬件[89]。线程将仿效传统的Posix线程[19]，Windows线程[163]和基于这两个库的其它库，比如boost::thread[16]。特别地，C++0x线程库不会尝试去解决跟线程有关的所有问题，相反，它会提供一组可移植的核心设施，但是存在与系统相关的技巧问题。与以前的库保持兼容性和与下层的操作系统保持兼容性是必要的。除了典型的线程库设施（比如lock和mutex）外，库将提供线程池和“futures”版本[56]作为一个更高层次的和更容易使用的交流设施：future能用于描述用于被其它线程计算的值的占位符；当futher读时潜在地发生同步。原形实现已经存在并且已经被评估[58]。

不幸地，我不得不接受，我对直接支持分布式编程的希望超出了C++0x现在能做的范围。

9、 回顾

这个回顾希望能从过去吸取教训，这在将来也许是有用的：

- 为什么C++成功了？
- 标准化进程怎么服务C++社团？
- 什么对C++产生了影响并带来什么样的冲击？
- 处于C++之下的理想有没未来？

9.1 为什么C++成功了？

这是一个考虑周全的人们每年都要问我几次的问题。没有考虑那么多的人们理所当然地认为答案就在网络上或其它地方。因此，它值得简短地回答和详细回答，去解除许多流行的虚构的故事。更长版本的答案能够在D&E[121]，在我的第一篇HOPL论文[120]和本文（比如，§1，§7，和§9.4）找到，本节是一个基本的总结：

C++成功因为：

- 底层访问加抽象：C++的初始的概念，“C类似的用于系统编程加Simula类似的用于抽象”是最强大和有用的想法（§7.1，§7.9）。
- 一个有用的工具：成为一门有用的语言对于它的使用者来说必须是完整的并能适应工作环境，它没有必要也不足以称之为是世界上最好的东西之一（§7.1）。
- 时机：C++不是第一个支持面向对象编程的语言，但从它面世的第一天起，就作为一种人们用于解决现实世界的问题的有用工具。
- 非专有的：AT&T没有独占C++，在早期，编译器是相当便宜的（比如，对于教育机构只需要$750），在1989所，语言的所有权力被移交给标准机构（ANSI和后来的ISO）。我和其它来自贝尔实验室的人积极帮助非AT&T的C++编译器开发者去开发自己的编译器。
- 稳定：高度的（不是100%）与C兼容在今天看来被认为是必不可少的。保持早期实现和定义的高度兼容性（仍然不是100%）仍在继续，标准进程在这儿是很重要的（§3.1，§6）。
- 演化：新的想法被吸引到C++里面（比如，异常、模板和STL）。世界在演化，活的语言也必须演化。C++并不是仅仅跟随潮流；有时，它领导（比如泛型编程和STL）潮流。标准进程在这儿是很重要的（§4.1，§5，§8）。

个人而言，我尤其喜欢C++的设计，不以提供清晰的可枚举的一组的解决方法去解决清晰地可枚举的一组的问题为目标：我从不设计一个工具只能做我想要做的事情[116]。开发者往往不会领会这个终极目标或不重视抽象，直到他们的需要发生改变。

尽管频繁地宣称，C++成功的理由不是因为：

- 仅仅是好运气：我常常听到这种论调。我不能理解人们怎么想象我和其它人在C++上的工作—只是纯粹靠运气—就能提供为百万计的系统构造者所喜欢的服务。显然，任何努力都需要运气

的成分，但相信只靠运气就能够说明C++ 1.0，C++ 2.0，ARM C++和C++98的成功是太过牵强了。25年来一直都靠运气也太不容易了。是的，我没有猜测到模板的使用或STL的使用，但我的目标一直都是一般性（§4.1.2）。

- AT&T的营销能力： 这很好笑！所有其它与C++进行商业竞争的语言都有很好的财政支持。AT&T在1985-1988（包括1988）用于C++营销的预算是$5000（而我们只能管理其中的$3000，参见D&E）。在第一次OOPSLA，我们不能负担起使用一台电脑的费用，我们没有广告，没有一个市场人员（我们使用一个粉笔板，上面是研究论文和他的研究者的名字），在那之后的几年，情况变得更糟
- 它是第一个：Ada，Eiffel，Object C，Smalltalk和各种List方言在C++之前都是可通过商业获取。对于许多程序员，C，Pascal和Modula-2是强烈的竞争者。
- 与C兼容：与C兼容的确是有帮助的—正如它应该的，因为这是它的一个设计目标。然而，C++从来就不仅仅是C，因为这个，它从许多C热爱者那里得到了许多敌意。为达到C++的设计目标，引进了一些不兼容性。C后来也采用了一些C++特征，但时间上滞后一些，确保C++因为那些特征（§7.6）得到比赞赏多得多的批语。C++从来不是一个“C的改头换面的继任者”。
- 便宜：在早期，一个C++编译器对于教育机构来说是相当便宜的，但对于商业使用，则跟它的竞争语言所提供的编译器一样昂贵（比如，商业源码授权需要$40000）。然后，提供商（比如Sun和微软）倾向于对C++加价而不是对他们自己的语言和系统加价。

显然，成功的理由是复杂和多样的，正如采用C++的个体和组织一样。

9.2 标准进程的影响

回顾‘90年代和21世纪的前几年’，关于C++和标准委员会对语言的工作，我们能提出什么样的问题呢？

- C++98比ARM C++更好吗？
- 最大的失误是什么？
- 什么使得C++恢复正常？
- C++能被改进吗？如果可以？怎么改进？
- ISO标准进程能被改进吗？如果可以？怎么改进？

是的，C++98（有一个更好的库）是一个比ARM C++更好的C++，被设计用来解决各种问题。因此，委员会的工作必须被认为是成功的。除此之外，C++程序员的人数在委员会的服务期间的增长已经超过了一个数量级（从1990年的差不多150,000到2004年的超过3百万; §1）。委员会能做得更好吗？毫无疑问，但具体怎么做就很难说了。当时做出决定的确切条件现在很难回忆得起来（更新你的记忆，参见D&E[121]）。

那么最大的失误是什么？对于早期的C++来说，这个问题很容易回答（“早期的AT&T发布没有带一个更大/更好的函数库”[120]）。然而，对于1991-1998期间，没有特别突出的事情—至少没有如果我们是现实主义者的话。如果我们不用成为现实主义者，我们可以梦想对分布式编程的支持，一个主要的库包含GUI支持（§5.5），标准平台的ABI，消除只用于向后兼容性的过分精细设计的瑕疵等。如果必须找一个出来做替罪羊的话，我认为我们应该离开语言特征和库设施的技术领域并考虑社会因素。C++社团没有真正的中心。尽管ISO C++委员会的成员的主要和持续不变的工作，委员会还是不为人知的；或对于数量巨大的C++用户来说，委员会是令人难以相信的遥不可及。其它编程语言通过维持会议、网站、合作开发论坛、FAQ、分布站点、期刊、工业协会、学术会议、教学点、授权机构等进行交流，在这些方面做得比C++好。我做了几种尝试去促进一个更加有影响力的C++社团，但我必须总结，我和任何人都不能取得充分的成功。C++世界的离心力太过强大了：每一个实现、库和工具供应商有他们自己的成果，通常帮了他们的用户一个大忙，但同时也隔离了那些用户。由于经济问题和在C++社团中没有独断的领导者，会议和期刊也没能成功。

什么使得C++恢复正常？首先，ISO标准在合理的时间内及时完成并且结果是真正的一致。那是当前所有C++使用的基础。标准加强C++作为一个多范例编程语言，带有强硬的支持用于系统编程，包括嵌入式系统编程。

其次，泛型编程被发展并出现了使用语言设施用于泛型编程的有效方法。它不仅要求与模板工作（比如，概念），还要求对资源管理（§5.3）和泛型编程技术的更好的理解。从长远的观点看，将泛型编程带到主流中可能是这个时期C++的技术贡献。

C++能被改进吗？如果可以？怎么改进？显然，技术上C++能够被改进，我也认为它能在现实世界的应用环境中改进。后者更难，因为强加了严峻的兼容性约束和文化上的需求。C++0x的工作就是在那个方向（§8）上的主要尝试。从更长远的观点看（C++1x?），我希望完善类型安全和用于并行的普通高层次模型。

关于改进标准进程的问题难以回答，一方面，ISO进程是缓慢的、官僚主义的、民主的、一致性驱动的，服从来自非常小的社团（有时就是一个个体）反对，缺少关注（“远见”），不能对工业或学术潮流的改变做出快速的反应。另一方面，ISO进程是成功的。我有时总结我的作用通过改写邱吉尔话：“The ISO standards process is the worst, except for all the rest.”。标准委员会看起来缺少一个“秘书处”，那里应该有许多技术人员，他们能全职地工作，检查需求，进行解决方法之间的比较，实验语言和库特征，并且制定出对提议的详细的确切表述。不幸的，长时间在这样一个“秘书处”服务对于第一流的学术和工业研究者来说是容易造成职业死亡。另一方面，这样的秘书工作如果由二流的人员担任会导致许多灾难。也许让技术人员在秘书处服务几年，支持象ISO C++委员会这样的基本的民主机构就是一种解决方法。那要求稳定的投资，尽管标准组织没有多余的现金。

商业提供商已经让用户习惯了许多“标准库”。ISO标准进程—至少由C和C++委员会所实践的—没有办法满足用户的期望的数以万计行的免费的标准库代码。Boost的工作（§4.2）是在强调这一点的一个尝试。然而，标准库不得不展现一个合理的一致的计算和存储的观点。这些库的开发者也不得不把大量的工作放在重要的工具库上，尽管这些工作没有多少有意义的智力挑战：只是提供“每个人所期望的东西”。还有许多俗世的规范说明需要去做，尽管商业提供者通常不提供—但标准必须提供因为它的目标是支持许多实现。一个松散地由自愿者组织起来的组织不能处理这个问题。对于一致性和已经取得的“普通开发工作”来说，多年机制为指导是必要的。

9.3 影响和冲击

我们会以多种方式去察看C++，其中一个就是影响：

1、对C++的主要影响是什么？

2、什么语言、系统和技术被C++所影响？

识别影响C++语言特征的影响是相对容易的，识别影响C++编程技术、库和工具就困难些，因为在那些领域里有太多的东西，并且都不需要通过标准委员会的同意。识别C++对其它语言、库、工具和技术的影响几乎是不可能的。这个任务被一种智力和商业竞赛所导致的缺乏强调记录来源和影响的趋势变得更加困难。市场份额的竞争和思想分享并不遵循理想的学术出版的规则。特别地，许多主要的想法有许多不同的版本和来源，并且人们倾向于把来源称之为竞争者—并且对于许多新语言来说，C++就是“要击败的对手”。

9.3.1 对C++的影响

对早期C++的主要影响是编程语言C和Simula。然后进一步的影响来自Algol68，BCPL，Clu，Ada，ML和Modula-2+[121]。语言-技术借鉴同时还伴随着与那些特征相关的语言技术。许多设计想法和编程技术来自经典的系统编程和UNIX。Simula不仅贡献语言特征，还有与面向对象编程和与数据抽象相关的编程技术的想法。我在C++中强调强静态类型检查作为用于设计、早期错误检测和运行时性能的工具主要来自Simula，对于泛型编程和STL，主流C++编程接收了许多

函数式编程技术和设计思想的良药。

9.3.2 C++的影响

C++的主要贡献以前是，现在也是，即许多系统构建使用到它（§7[137]）。编程语言的主要目的是帮助构建好的系统。C++已经成为我们的文明所依赖（比如，电信系统，个人计算机，娱乐，医疗和电子商务）的基础建筑的十分重要的组成部分，并且是这个时期的某些最鼓舞人心的成就的一部分（比如，火星流浪者号和人类基因序列）。

C++是卓越的，将面向对象编程引进主流。为完成这个，C++社团必须克服两个最普遍的反对：

- 面向对象编程天生就没有效率的
- 面向对象编程对于“普通程序员”太过复杂而不能被使用

C++的独特的OO变种是继承自Simula，而不是来自Smalltalk或Lisp，并强调静态类型检查的作用和与之相关的设计技术。较少被公认的是C++还带有非OO的数据抽象技术和对非OO设施的静态的类型接口（比如，对简单的老的C函数的适当的类型检查；§7.6）。当意识到它之后，它通常被批评为“杂种”、“短时间的”或“静态的”。我认为它是一种有价值、并且通常是对关注类层次和动态键入（dynamic typing）的面向对象编程观点的一种必要的补充。

C++把泛型编程引入主流。这是早期C++强调静态类型接口的一种自然演化，并引进许多函数式编程技术，转换自列表和递归到一般序列和迭代器。函数模板加函数对象通常占据了高阶函数的角色。STL是一个关键的趋势。另外，模板和函数对象的使用导致了强调在高性能计算（§7.3）中的静态类型安全，高性能计算至今仍在现实世界的应用程序中缺失。

C++对其它语言特征的影响，最明显的是在C中，C从C++“借”到：

- 函数原型
- const
- inline
- bool
- complex
- 声明作为语句
- 在for语句的初始化中声明
- // 注释

可惜，除了//（我从BCPL中借过来的），这些特征被C以不兼容的形式引进。还有void*，这是Larry Rosler，Steve Johnson和我在贝尔实验室在1982年一起实现的（参见D&E[121]）。

泛型编程、STL和模板同样影响到了其它语言的设计。在80和90年代，模板和与模板相关联的编程技术频繁地被蔑视为“非面向对象”，太复杂或代价太高；工作区（workarounds）被称赞为“简单”，跟早期的社团对关于面向对象编程和类的评论类似。然而，到2005年，Java和C#都获得了“generics”和静态键入的容器。这些“generics”看起来很象模板，但Java的主要是用于抽象类型的语法的糖衣，并且比模板严格得多。尽管仍不如模板般有效率或灵活性，C# 2.0的“generics”被更好地集成到类型系统，甚至（那就是模板通常做的）提供特化的形式。

C++在现实世界中的成功使用也导致了一些比较不好的影响。在现代语言中频繁使用的丑陋的和不整齐的C/C++风格的语法并不是我感到自豪的东西。然而，它又是一个优秀的反映C++影响的指示器—没人会从第一原则中提出这样的句法。

C++明显地影响到了Java、C#和各种脚本语言。对Ada95、COBOL和Fortran的影响就没有那么明显，但肯定有。例如，有一个Ada版本的Booch组件[15]，能够在代码量和性能上与C++版本进行对比就是一个普通的引证目标。C++甚至影响到函数式编程社团（比如[59]）。然而，我认为这些影响是很小的。

一个被影响的重要领域是用于双向通信的系统和组件，比如CORBA和COM，这两个都有一

个基本的接口模型，继承自C++的抽象类。

9.4 Beyond C++

C++在未来的许多年都会是许多系统开发的支柱。在C++0x之后，我期望看到语言和它的社团所采用的C++1x能适应新挑战。然而，C++跟当前语言的哪一个哲学相吻合？换句话说，不管兼容性，C++的本质是什么？C++是对理想的一种近似，但它本身并不是理想。什么样的关键的属性、想法和理想可以成为一门新语言和编程技术的种子？C++是

- 一组低级语言机制（直接处理硬件）
- 与强大的组合抽象机制结合
- 对运行环境的最小依赖

在这些方面，它是很独特的，许多它的力量来自这些特征的结合。C是到目前为止最成功的传统的系统编程语言，是许多流行和有用的语言中的主要幸存者。C以C++的方法靠近机器（§2，§6.1），但C没有提供重要的抽象机制。大多“现代语言”（比如Java，C#，Python，Ruby，Haskell和ML）提供抽象机制，但故意在它们与机器之间设置障碍。最流行的是依赖于虚拟机器，增加了一个巨大的运行时支持系统。当你构建一个应用程序需要提供的服务，但只局限于一定的应用范围时 (§7.1)，这是一个巨大的优势。它还确保每一个应用程序是一个巨大系统的一部分，这使得它难以完全理解，从而不可能正确完成和良好的隔离。

与之对比，C++和标准库能用C++实现（有少数几个例外是由于兼容性要求和次要的规范说明过失）。C++是完整的，作为一种处理硬件和靠近硬件层编程的有效抽象的机制。有时，最理想的使用一个硬件特征要求编译器内在的或一个内联汇编器插入、而不破坏基本的模型。实际上，这能看成是模型的一个完整部分。

这样一个模型能否应用于未来的硬件、未来系统的要求和未来应用程序的要求？我认为在没有兼容性要求下，它是可以的，基于这种模型，新的语言可以比C++更小、更简单、更具表达力和更可容易使用，没有性能损失或应用领域的限制。多小呢？比如只有C++定义的10%并且与前端编译器的大小类似。在D&E[121]的回顾章节中，我表达了那样的想法“在C++内部，有更小和更干净的语言要挣脱出来”。许多简化将来自一般化—消灭许多特殊用例，这些特殊用例使得C++难以处理—而不是限制或把工作从编译期转移到运行期。但一个与计算机靠近并且带0开销抽象的语言就能满足“现代要求”吗？特别地，它必须是：

- 完全的类型安全
- 有能力有效使用并发硬件

这不是一个给出技术论证的地方，但我对它能被实现有信心。这样一个语言将要求一个现实（与真实硬件相关）的机器模型（包括内存模型）和一个简单对象模型。这个对象模型将类似于C++的：直接映射基本类型到机器对象并简单组合作为抽象的基础。这意味着以指针和数组的形式去支持“对象序列”的基本模型（类似于STL提供的）。获得类型安全—即是说，消灭非法指针使用、范围错误等的可能性—带最小的运行时检查是一个挑战，但有许多对软件、硬件和静态分析研究，这给了我们乐观的理由。该模型将包含真正的局部变量（对于用户定义类型跟内建类型一样），意味着支持“在初始化时获取资源”风格的资源管理，它不能是“用于所有东西的废料收集”模型，尽管毫无疑问，在大多数语言中会有废料收集的一席之地。

这类语言将支持哪种编程？它将更好地支持什么样的编程呢？这类语言会是一个系统编程语言，适合于硬实时应用，设备驱动，嵌入设备等。我认为理想的系统编程语言就属于这种一般模型—你需要靠近机器的特征，可预测的性能，（类型）安全和强大的抽象特征。其次，这类语言将是所有资源受限的应用程序的理想选择，大多应用程序都满足这两条准则。这是一个巨大的并且不断增长的应用。显然，这个论据是对C++的力量分析的另一个变种§7.1。

什么将与当前的C++不同？大小、类型、安全和对并发的完整支持将是最明显的不同。更重要的是，这样的语言比C++更经受得起推理，有更多的工具可以使用，语言依赖于许多运行

时的支持。除那之外，还有广大的范围可以改进，在设计细节和规范说明技术上，从而超越C++。例如，核心语言和标准库特征间的整合能更加地平滑；C++理想地一致地支持内建类型和用户定义类型将完全达到，以便一个用户不能（未经实现测试）区分出类型究竟是什么类型。很明显，本质上所有的这样的语言都应该定义到一个抽象机器（从而消灭所有偶然的“编译器定义”行为），而不用对靠近机器的理想进行妥协：可以利用机器体系结构在操作和内存模型上的巨大的相似性。

现实世界的挑战通过详细说明和实现正确系统（通常在资源受限和通常在出现硬件失败的可能性的条件下）得到满足。我们的文明重要地依赖于软件，而软件的数量和复杂度在持续增长。到现在为止，我们（大型系统的构建者）通过补丁和令人难以相信的数目的运行时测试来满足挑战。对于起动器，除去并发执行，我们的计算机并没有变得更快，以弥补软件膨胀。处理方法中我最感乐观的是对软件提供原则的方法，依赖更多的数学推理，更多声明的属性和更多对程序属性的静态检验。STL是朝那个方向（受C++限制的约束）前进的一个例子。函数式编程也是朝那个方向（受基本的内存模型与硬件不一样和未充分利用静态属性的限制）前进的例子。在那个方向要有更大的进步要求语言支持想要的属性和推理的规范说明（概念就是向那个方向的迈进的一步）。对于这类推理、运行时解析、运行时测试和多层运行时映射代码到硬件至少是浪费和严重的障碍。机器接近、类型安全和组合的抽象模型是最终的力量，对推理提供最少的障碍。

10、回顾

显然，我亏欠了C++标准委员会的委员们许多—他们以前工作，并且现在仍工作在—这儿描述的语言和库特征。类似的，我要感谢编译器和工具构建者，他们让C++成为一个用于解决现实世界中的问题的能存活下来的工具。

感谢Dave Abrahams，Matt Austern，Paul A. Bistow，Steve Clamage，Greg Colvin，Gabriel Dos Reis，S. G.Ganesh，Lois Goldthwaite，Kevlin Henney，Howard Hinnant，Jaakko Järvi，Andy Koenig，Bronek Kozicki，Paul McJones，Scott Meyers，Nathan Myers，W. M. (Mike) Miller，SeanParent，Tom Plum，PremAnand M. Rao，Jonathan Schilling，Alexander Stepanov，Nicholas Stroustrup，James Widman和J. C. van Winkel对本文草稿的建设性的评论。

同样感谢HOPL-III的审稿人：Julia Lawall，Mike Mahoney，Barbara Ryder，Herb Sutter和Ben Zorn—他们的努力使得许多介绍性的材料被添加进来，从而有希望使得本文更自我包含和更容易被那些不是C++内行的人所阅读。

我非常感激Brain Kernighan和Doug McIlroy，从1990年他们就花时间在广泛的语言相关的主题上倾听我的意见并教育我。特别感谢Al Aho of Columbia University借给了我一间办公室，从而使我可以在那里写出本文的第一份草稿。

引用：

[1] David Abrahams: *Exception-Safety in Generic Components*.M. Jazayeri, R. Loos, D. Musser (eds.): Generic Programming,Proc. of a Dagstuhl Seminar. Lecture Notes on Computer Science. Volume 1766. 2000. ISBN: 3-540-41090-2.

[2] David Abrahams and Aleksey Gurtovoy: *C++ TemplateMeta-programming* Addison-Wesley. 2005. ISBN 0-321-22725-5.

[3] Andrei Alexandrescu: *Modern C++ Design*. Addison-Wesley. 2002. ISBN 0-201-70431.

[4] Ping An, Alin Jula, Silvius Rus, Steven Saunders, Tim Smith,Gabriel Tanase, Nathan Thomas, Nancy Amato, Lawrence Rauchwerger: *STAPL: An Adaptive, Generic Parallel C++ Library*. In Wkshp. on Lang. and Comp. for Par. Comp.(LCPC), pp. 193-208, Cumberland Falls, Kentucky, Aug 2001.

[5] AT&T C++ translator release notes. *Tools and ReusableComponents*. 1989.

[6] M. Austern: *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*. AddisonWesley. 1998. ISBN: 0-201-30956-4.

[7] John Backus: *Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs*. Communications of the ACM 21, 8 (Aug. 1978).

[8] J. Barreiro, R. Fraley, and D. Musser: *Hash Tables for the Standard Template Library*. Rensselaer Polytechnic Institute and Hewlett Packard Laboratories. February, 1995. `ftp://ftp.cs.rpi.edu/pub/stl/hashdoc.ps.Z`

[9] Graham Birtwistle et al.: *SIMULA BEGIN*. Studentlitteratur.Lund. Sweden. 1979. ISBN 91-44-06212-5.

[10] Jasmin Blanchette and Mark Summerfield: *C++ GUI Programming with Qt3*. Prentice Hall. 2004. ISBN 0-13-124072-2.

[11] Hans-J. Boehm: *Space Efficient Conservative Garbage Collection*. Proc. ACM SIGPLAN '93 Conference on Programming Language Design and Implementation. ACM SIGPLAN Notices. June 1993. `http://www.hpl.hp.com/personal/Hans_Boehm/gc/`.

[12] Hans-J. Boehm and Michael Spertus: *Transparent Garbage Collection for C++*. ISO SC22 WG21 TR NN1943==06-0013.

[13] Hans-J. Boehm: *An Atomic Operations Library for C++*. ISO SC22 WG21 TR N2047==06-0117.

[14] Hans-J. Boehm: *A Less Formal Explanation of the Proposed C++ Concurrency Memory Model*. ISO SC22 WG21 TR N2138==06-0208.

[15] Grady Booch: *Software Components with Ada*. Benjamin Cummings. 1988. ISBN 0-8053-0610-2.

[16] The Boost collection of libraries. `http://www.boost.org`.

[17] Walter Bright: *D Programming Language*. `http://www.digitalmars.com/d/`.

[18] Peter A. Buhr and Glen Ditchfield: *Adding Concurrency to a Programming Language*. Proc. USENIX C++ Conference. Portland, OR. August 1992.

[19] David Butenhof: *Programming With Posix Threads*. ISBN 0-201-63392-2. 1997.

[20] T. Cargill: *Exception handling: A False Sense of Security*. The C++ Report, Volume 6, Number 9, November-December 1994.

[21] D. Caromel et al.: *C++//*. In *Parallel programming in C++*. (Wilson and Lu, editors). MIT Press. 1996. ISBN 0-262-73118-5.

[22] Fernando Cacciola: *A proposal to add a general purpose ranged-checked numeric_cast<>* (Revision 1). ISO SC22 WG21 TR N1879==05-0139.

[23] CGAL: Computational Geometry Algorithm Library. *http://www.cgal.org/*.

[24] Siva Challa and Artur Laksberg: *Essential Guide to Managed Extensions for C++*. Apress. 2002. ISBN: 1893115283.

[25] K. M. Chandy and C. Kesselman: *Compositional C++: Compositional Parallel Programming*. Technical Report.California Institute of Technology. [Caltech CSTR: 1992.cstr-92-13].

[26] Shigeru Chiba: *A Metaobject Protocol for C++*. Proc.OOPSLA'95. `http://www.csg.is.titech.ac.jp/~chiba/openc++.html`.

[27] Steve Clamage and David Vandevoorde. Personal communications.2005.

[28] J. Coplien: *Multi-paradigm Design for C++*. Addison Wesley. 1998. ISBN 0-201-82467-1.

[29] Lawrence Crowl: *Thread-Local Storage*. ISO SC22 WG21 TR N1966==06-0036.

[30] Lawrence Crowl: *Defaulted and Deleted Functions*. ISO SC22 WG21 TR N2210==07-0070.

[31] Krzysztof Czarnecki and Ulrich W. Eisenecker: *Generative Programming — Methods, Tools, and Applications*. Addison-Wesley, June 2000. ISBN 0-201-30977-7.

[32] Beman Dawes: *Filesystem Library Proposal for TR2 (Revision 2)* . ISO SC22 WG21 TR N1934==06-0004.

[33] Jeffrey Dean and Sanjay Ghemawat: *MapReduce: Simplified Data Processing on Large Clusters*. OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December, 2004.

[34] D. Detlefs: *Garbage collection and run-time typing as a C++ library*. Proc. USENIX C++ conference. 1992.

[35] M. Ellis and B. Stroustrup: *The Annotated C++ Reference Manual* ("The ARM") Addison Wesley. 1989.

[36] Dinkumware: *Dinkum Abridged Library* `http://www.dinkumware.com/libdal.html`.

[37] Gabriel Dos Reis and Bjarne Stroustrup: *Formalizing C++*. TAMU CS TR. 2005.

[38] Gabriel Dos Reis and Bjarne Stroustrup: *Specifying C++ Concepts*. Proc. ACM POPL 2006.

[39] Gabriel Dos Reis and Bjarne Stroustrup: *Generalized Constant Expressions* (Revision 3). ISO SC22 WG21 TR N1980==06-0050.

[40] The Embedded C++ Technical Committee: *The Language Specification & Libraries Version*. WP-AM-003. Oct 1999 (`http://www.caravan.net/ec2plus/`).

[41] Ecma International: *ECMA-372 Standard: C++/CLI Language Specification.* `http://www.ecma-international.org/publications/standards/Ecma-372.htm`. December 2005.

[42] Boris Fomitchev: *The STLport Story*. `http://www.stlport.org/doc/story.html`.

[43] Eric Gamma, et al.: *Design Patterns.* Addison-Wesley. 1994.ISBN 0-201-63361-2.

[44] R. Garcia, et al.: *A comparative study of language suppor for generic programming.* ACM OOPSLA 2003.

[45] Jeff Garland: *Proposal to Add Date-Time to the C++ Standard Library*. ISO SC22 WG21 TR N1900=05-0160.

[46] N.H. Gehani and W.D. Roome: *Concurrent C++: Concurrent Programming with Class(es).* Software—Practice and Experience, 18(12):1157—1177, December 1988.

[47] Geodesic: Great circle. Now offered by Veritas.

[48] M. Gibbs and B. Stroustrup: *Fast dynamic casting*. Software—Practice&Experience. 2005.

[49] William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir: *MPI: The Complete Reference — 2nd Edition: Volume 2 — The MPI-2 Extensions* . The MIT Press. 1998.

[50] Keith E. Gorlen: *An Object-Oriented Class Library for C++ Programs*. Proc. USENIX C++ Conference. Santa Fe, NM. November 1987.

[51] J. Gosling and H. McGilton: *The Java(tm) Language Environment: A White Paper*. `http://java.sun.com/docs/white/langenv/`.

[52] Douglas Gregor, Jaakko Järvi, Jeremy G. Siek, Gabriel Dos Reis, Bjarne Stroustrup, and Andrew Lumsdaine: *Concepts: Linguistic Support for Generic Programming*. Proc. of ACM OOPSLA'06. October 2006.

[53] D. Gregor and B. Stroustrup: *Concepts*. ISO SC22 WG21 TR N2042==06-0012.

[54] D. Gregor, J. Järvi, G. Powell: *Variadic Templates* (Revision 3). ISO SC22 WG21 TR N2080==06-0150.

[55] Douglas Gregor and Jaakko Järvi: *Variadic Templates for C++*. Proc. 2007 ACM Symposium on Applied Computing. March 2007.

[56] R. H. Halstead: *MultiLisp: A Language for Concurrent Symbolic Computation*. TOPLAS. October 1985.

[57] H. Hinnant, D. Abrahams, and P. Dimov: *A Proposal to Add an Rvalue Reference to the C++ Language*. ISO SC22WG21 TR N1690==04-0130.

[58] Howard E. Hinnant: *Multithreading API for C++0X – A Layered Approach*. ISO SC22 WG21 TR N2094==06-0164.

[59] J. Hughes and J. Sparud: *Haskell++: An object-oriented extension of Haskell*. Proc. Haskell Workshop, 1995.

[60] IA64 C++ ABI. `http://www.codesourcery.com/cxx-abi`.

[61] IDC report on programming language use. 2004. http://www.idc.com.

[62] *Standard for the C Programming Language*. ISO/IEC 9899.("C89").

[63] *Standard for the C++ Programming Language*. ISO/IEC 14882. ("C++98").

[64] *Standard for the C Programming Language*. ISO/IEC 9899:1999. (“C99”).

[65] International Organization for Standards: *The C Programming Language*. ISO/IEC 9899:2002. Wiley 2003. ISBN 0-470-84573-2.

[66] International Organization for Standards: *The C++ Programming Language* ISO/IEC 14882:2003. Wiley 2003. ISBN 0-470-84674-7.

[67] *Technical Report on C++ Performance*. ISO/IEC PDTR 18015.

[68] *Technical Report on C++ Standard Library Extensions*.ISO/IEC PDTR 19768.

[69] ISO SC22/WG21 website: `http://www.open-std.org/jtc1/sc22/wg21/`.

[70] Sorin Istrail and 35 others: *Whole-genome shotgun assembly and comparison of human genome assemblies.* Proc. National Academy of Sciences. February, 2004.`http://www.pantherdb.org/`.

[71] Jaakko Järvi: *Proposal for adding tuple type into the standard library*. ISO SC22 WG21 TR N1382==02-0040.

[72] Jaakko Järvi, Gary Powell, and Andrew Lumsdaine: *The Lambda Library: Unnamed Functions in C++*. Software-Practice and Experience, 33:259-291, 2003.

[73] J. Järvi, B. Stroustrup and G. Dos Reis: *Deducing the type of a variable from its initializer expression*. ISO SC22 WG21 TR N1894, Oct. 2005.

[74] Jaakko Järvi, Douglas Gregor, Jeremiah Willcock, Andrew Lumsdaine, and Jeremy Siek: *Algorithm specialization in generic programming: challenges of constrained generics in C++*. Proc. PLDI 2006.

[75] L. V. Kale and S. Krishnan: *CHARM++* in *Parallel programming in C++*. (Wilson and Lu, editors). MIT Press. 1996. ISBN 0-262-73118-5.

[76] Brian Kernighan and Dennis Richie: *The C Programming Language* (“K&R”). Prentice Hall. 1978. ISBN 0-13-110163-3.

[77] Brian Kernighan and Dennis Richie: *The C Programming Language (2nd edition)* (“K&R2” or just “K&R”). Prentice Hall. 1988. ISBN 0-13-110362-8.

[78] Brian Kernighan: *Why Pascal isn’t my favorite programming language.* AT&T Bell Labs technical report No. 100. July 1981.

[79] Andrew Koenig and Bjarne Stroustrup: *Exception Handling for C++(revised)*. Proc. USENIX C++ Conference. San Francisco, CA. April 1990. Also, Journal of Object-Oriented Programming. July 1990.

[80] Christopher Kohlhoff: *Networking Library Proposal for TR2*. ISO SC22 WG21 TR N2054==06-0124.

[81] Mark A. Linton and Paul R. Calder: *The Design and Implementation of InterViews*. Proc. USENIX C++ Conference. Santa Fe, NM. November 1987.

[82] A. Mishra et al.: R++: *Using Rules in Object-Oriented Designs*. Proc. OOPSLA-96. `http://www.research.att.com/sw/tools/r++/`.

[83] W. G. O’Farrell et al.: *ABC++* in *Parallel Programming in C++*. (Wilson and Lu, editors). MIT Press. 1996. ISBN 0-262-73118-5.

[84] Thorsten Ottosen: *Proposal for new for-loop*. ISO SC22 WG21 TR N1796==05-0056.

[85] Sean Parent: personal communications. 2006.

[86] J. V. W. Reynolds et al.: *POOMA* in *Parallel Programming in C++*. (Wilson and Lu, editors). MIT Press. 1996. ISBN 0-262-73118-5.

[87] Mike Mintz and Robert Ekendahl: *Hardware Verification with C++ — A Practitioner’s Handbook*. Springe Verlag.2006. ISBN 0-387-25543-5.

[88] Nathan C. Myers: *Traits: a new and useful template technique*.The C++ Report, June 1995.

[89] C. Nelson and H.-J. Boehm: *Sequencing and the concurrency memory model*. ISO SC22 WG21 TR N2052==06-0122.

[90] Mac OS X 10.1 November 2001 Developer Tools CD Release Notes: *Objective-C++*. `http://developer.apple.com/releasenotes/Cocoa/Objective-C++.html`

[91] Leonie V. Rose and Bjarne Stroustrup: *Complex Arithmetic in C++*. Internal AT&T Bell Labs Technical Memorandum.January 1984. Reprinted in AT&T C++ Translator Release Notes. November 1985.

[92] P. Rovner, R. Levin, and J. Wick: *On extending Modula-2 for building large integrated systems*. DEC research report #3.1985.

[93] J. Schilling: *Optimizing Away C++ Exception Handling* ACM SIGPLAN Notices. August 1998.

[94] Marc Snir, Steve Otto, Steven Huss-Lederman, DavidWalker, and Jack Dongarra: *MPI: The Complete Reference* The MIT Press. 1995. `http://www-unix.mcs.anl.gov/mpi/`.

[95] D. C. Schmidt and S. D. Huston: *Network programming using C++*. Addison-Wesley Vol 1, 2001. Vol. 2, 2003. ISBN 0-201-60464-7 and ISBN 0-201-79525-6.

[96] Jeremy G. Siek and Andrew Lumsdaine: *The Matrix Template Library: A Generic Programming Approach to High Performance Numerical Linear Algebra* ISCOPE '98. `http://www.osl.iu.edu/research/mtl`.

[97] J. Siek et al.: *Concepts for C++*. ISO SC22 WG21 WG21-N1758.

[98] Jeremy G. Siek and Walid Taha: *A Semantic Analysis of C++ Templates*. Proc. ECOOP 2006.

[99] Yannis Smargdakis: *Functional programming with the FC++ library*. ICFP'00.

[100] Olaf Spinczyk, Daniel Lohmann, and Matthias Urban: *AspectC++: an AOP Extension for C++*. Software Developer's Journal. June 2005. `http://www.aspectc.org/`.

[101] A. A. Stepanov and D. R.Musser: *The Ada Generic Library: Linear List Processing Packages*. Compass Series, Springer- Verlag, 1989.

[102] A. A. Stepanov: *Abstraction Penalty Benchmark, version 1.2 (KAI)*. SGI 1992. Reprinted as Appendix D.3 of [67].

[103] A. Stepanov and M. Lee: *The Standard Template Library*. HP Labs TR HPL-94-34. August 1994.

[104] Alex Stepanov: Foreword to Siek, et al.: *The Boost Graph Library*. Addison-Wesley 2002. ISBN 0-21-72914-8.

[105] Alex Stepanov: personal communications. 2004.

[106] Alex Stepanov: personal communications. 2006.

[107] Christopher Strachey: *Fundamental Concepts in Programming Languages*. Lecture notes for the International Summer School in Computer Programming, Copenhagen, August 1967.

[108] Bjarne Stroustrup: *Classes: An Abstract Data Type Facility for the C Language*. Bell Laboratories Computer Science Technical Report CSTR-84. April 1980. Revised, August

1981. Revised yet again and published as [109].

[109] Bjarne Stroustrup: *Classes: An Abstract Data Type Facility for the C Language*. ACM SIGPLAN Notices. January 1982.Revised version of [108].

[110] B. Stroustrup: *Data abstraction in C*. Bell Labs Technical Journal. Vol 63. No 8 (Part 2), pp 1701-1732. October 1984.

[111] B. Stroustrup: *A C++ Tutorial*. Proc. 1984 National Communications Forum. September, 1984.

[112] B. Stroustrup: *The C++ Programming Language* ("TC++PL"). Addison-Wesley Longman. Reading, Mass., USA. 1986. ISBN 0-201-12078-X.

[113] Bjarne Stroustrup: *What is Object-Oriented Programming?* Proc. 14th ASU Conference. August 1986. Revised version in Proc. ECOOP'87, May 1987, Springer Verlag Lecture Notes in Computer Science Vol 276. Revised version in *IEEE Software Magazine*. May 1988.

[114] Bjarne Stroustrup: *An overview of C++*. ACM Sigplan Notices, Special Issue. October, 1986

[115] B. Stroustrup and J. E. Shopiro: *A Set of Classes for Coroutine Style Programming*. Proc. USENIX C++ Workshop. November, 1987.

[116] Bjarne Stroustrup: quote from 1988 talk.

[117] Bjarne Stroustrup: *Parameterized Types for C++*. Proc. USENIX C++ Conference, Denver, CO. October 1988. Also, USENIX Computer Systems, Vol 2 No 1. Winter 1989.

[118] B. Stroustrup: *The C++ Programming Language*, 2nd Edition ("TC++PL2" or just "TC++PL"). Addison-Wesley Longman. Reading, Mass., USA. 1991. ISBN 0-201-53992-6.

[119] Bjarne Stroustrup and Dmitri Lenkov: *Run-Time Type Identification for C++*. The C++ Report. March 1992. Revised version. Proc. USENIX C++ Conference. Portland, OR. August 1992.

[120] B. Stroustrup: *A History of C++: 1979-1991*. Proc ACM HOPL-II. March 1993. Also in Begin and Gibson (editors): *History of Programming Languages*. Addison-Wesley. 1996. ISBN 0-201-89502-1.

[121] B. Stroustrup: *The Design and Evolution of C++*. ("D&E"). Addison-Wesley Longman. Reading Mass. USA. 1994. ISBN 0-201-54330-3.

[122] B. Stroustrup: *Why C++ is not just an object-oriented programming language*. Proc. ACM OOPSLA 1995.

[123] B. Stroustrup: *Proposal to Acknowledge that Garbage Collection for C++ is Possible*. WG21/N0932 X3J16/96-0114.

[124] B. Stroustrup: *The C++ Programming Language*, 3rd Edition ("TC++PL3" or just "TC++PL"). Addison-Wesley Longman. Reading, Mass., USA. 1997. ISBN 0-201-88954-4.

[125] B. Stroustrup: *Learning Standard C++ as a New Language*. The C/C++ Users Journal. May 1999.

[126] B. Stroustrup: *The C++ Programming Language*, Special Edition ("TC++PL"). Addison-Wesley, February 2000. ISBN 0-201-70073-5.

[127] B. Stroustrup: *C and C++: Siblings*, *C and C++: A Case for Compatibility*, *C and C++: Case Studies in Compatibility*. The C/C++ Users Journal. July, August, and September 2002.

[128] B. Stroustrup: *Why can't I define constraints for my template parameters?*. `http://www.research.att.com/~bs/bs_faq2.html#constraints`.

[129] B. Stroustrup and G. Dos Reis: *Concepts — Design choices for template argument checking.* ISO SC22 WG21 TR N1522. 2003.

[130] B. Stroustrup: *Concept checking — A more abstract complement to type checking.* ISO SC22 WG21 TR N1510.2003.

[131] B. Stroustrup: *Abstraction and the C++ machine model*. Proc. ICESS'04. December 2004.

[132] B. Stroustrup, G. Dos Reis: *A concept design* (Rev. 1). ISO SC22 WG21 TR N1782=05-0042.

[133] B. Stroustrup: *A rationale for semantically enhanced library languages.* ACM LCSD05. October 2005.

[134] B. Stroustrup and G. Dos Reis: *Supporting SELL for High-Performance Computing*. LCPC05. October 2005.

[135] Bjarne Stroustrup and Gabriel Dos Reis: *Initializer lists.* ISO SC22 WG21 TR N1919=05-0179.

[136] B. Stroustrup: *C++ pages*. `http://www.research.att.com/~bs/C++.html`.

[137] B. Stroustrup: *C++ applications.* `http://www.research.att.com/~bs/applications.html`.

[138] H. Sutter, D. Miller, and B. Stroustrup: *Strongly Typed Enums* (revison 2). ISO SC22 WG21 TR N2213==07-0073.

[139] H. Sutter and B. Stroustrup: *A name for the null pointer: nullptr* (revision 2). ISO SC22 WG21 TR N1601==04-0041.

[140] Herb Sutter: *A Design Rationale for C++/CLI*, Version 1.1 — February 24, 2006 (later updated with minor editorial fixes). `http://www.gotw.ca/publications/C++CLIRationale.pdf`.

[141] Numbers supplied by Trolltech. January 2006.

[142] UK C++ panel: *Objection to Fast-track Ballot ECMA-372 in JTC1 N8037*.

http://public.research.att.com/~bs/uk-objections.pdf. January 2006.

[143] E. Unruh: *Prime number computation*. ISO SC22 WG21 TR N462==94-0075.

[144] D. Vandevoorde and N. M. Josuttis: *C++ Templates — The Complete Guide*. Addison-Wesley. 2003. ISBN 0-201-73884-2.

[145] D. Vandevoorde: *Right Angle Brackets* (Revision 1). ISO SC22 WG21 TR N1757==05-0017 .

[146] D. Vandevoorde: *Modules in C++* (Version 3). ISO SC22 WG21 TR N1964==06-0034.

[147] Todd Veldhuizen: *expression templates*. C++ Report Magazine, Vol 7 No. 4, May 1995.

[148] Todd Veldhuizen: *Template metaprogramming*. The C++ Report, Vol. 7 No. 5. June 1995.

[149] Todd Veldhuizen: *Arrays in Blitz++*. Proceedings of the 2nd International Scientific Computing in Object Oriented Parallel Environments (ISCOPE'98). 1998.

[150] Todd Veldhuizen: *C++ Templates are Turing Complete* 2003.

[151] Todd L. Veldhuizen: *Guaranteed Optimization for Domain-Specific Programming*. Dagstuhl Seminar of Domain-Specific Program Generation. 2003.

[152] Andre Weinand et al.: *ET++ — An Object-Oriented Application Framework in C++*. Proc. OOPSLA'88. September 1988.

[153] Gregory V.Wilson and Paul Lu: *Parallel programming using C++*. Addison-Wesley. 1996.

[154] P.M. Woodward and S.G. Bond: *Algol 68-R Users Guide*.Her Majesty's Stationery Office, London. 1974. ISBN 0-11-771600-6.

[155] *The Visualization Toolkit, An Object-Oriented Approach To 3D Graphics, 3rd edition*, ISBN 1-930934-12-2. http://public.kitware.com/VTK.

[156] *FLTK: Fast Light Toolkit*. http://www.fltk.org/.

[157] Lockheed Martin Corporation: *Joint Strike Fighter air vehicle coding standards for the system development and demonstration program*. Document Number 2RDU00001 Rev C. December 2005. http://www.research.att.com/~bs/JSF-AV-rules.pdf.

[158] Microsoft Foundation Classes.

[159] *Smartwin++: An Open Source C++ GUI library*. http://smartwin.sourceforge.net/.

[160] *WTL: Windows Template Library —- A C++ library for developing Windows applications and UI components*. http://wtl.sourceforge.net.

[161] *gtkmm: C++ Interfaces for GTK+ and GNOME*. http://www.gtkmm.org/

[162] *wxWidgets: A cross platform GUI library*. http://wxwidgets.org/.

[163] Windows threads: *Processes and Threads*. http://msdn.microsoft.com/library/en-us/dllproc/base/processes_and_threads.asp.

[164] Wikipedia: *Java (Sun) — Early history*. http://en.wikipedia.org/wiki/Java_(Sun).