zjufirefly

目录视图 摘要视图 订阅 管理博客 写新文章

摘录的一些Bjarne Stroustrup关于C++的谈话内容

分类： C++ 2014-05-02 12:30 64人阅读 评论(0) 收藏 编辑 删除

专访 Bjarne Stroustrup

来源：荣耀 马皓明 译 作者：Bjarne Stroustrup 等级：一般
发布于2005-10-22 22:54 被读1091次 【字体：大 中 小】

Bj p 其它言论 http://www.royaloo.com/bjarne/bjarne.htm

承蒙孟岩先生允许，本译文引用了他的摘译稿，谨致谢意。

Elden Nelson：如果 ，头设计C++语言，您会做些什么样的改变？

Bjarne Stroustrup：当然，你永远都不可能重新设计一种语言，那没有意义，而且任何一种语言都是它那个时代的产物。如果让我今天再设计一种语言，我仍然会综合考虑逻辑的优美、效率、通用性、实现的复杂程度以及人们的喜好。要知道人们的习惯对于其喜好有着巨大的影响。现在，我会寻找一种简单得多的语法 — 它可能与人们对“熟悉”和“简单”的混淆认识相悖，我会把对类型系统的侵犯限制在极少的语言构造里，并且用明显“丑陋”的语法来标识它们。（就象我对对新风格的“转型”的处理，比方说，reinterpret_cast<数据类型>(p)就是一个用来描述一种“丑陋”操作的“丑陋”记号）。这样可以很容易地禁止不安全的操作。我还会把核心语言的体积搞得尽可能小一些，包括类和模板的关键的抽象特性，而把很多其它的语言功能放在库里来解决。当然我也会保证核心语言足够强大，使得那些库本身也足以用此核心语言来产生。我可不希望标准库的编写者依赖于普通用户无法使用的额外的语言特性。我还努力使核心语言的定义更加精确。最重要的是，我会在该语言被广泛使用之前尽可能维持一个很长的酝酿期，这样我就可以根据来自真实用户的坚实反馈对它进行改进。这可能是最困难的，因为一旦有什么东西明显出色并且前途光明，它就会被广为使用，此后进行任何不兼容的修正都将变得极其困难。我相信这些思想与我当初设计C++时的理念是非常类似的，它们同样也指引着一、二十年来C++的不断演化。当然，我认为现在还没有什么东西能让我觉得象是“完美的语言”。

Elden Nelson：当您设计C++语言时，您是否借鉴了其他崭露头角的对象语言（例如Modula-2）的思想？

Bjarne Stroustrup：在C++的设计过程中，我吸取了C、BCPL、SIMULA、ALGOL 68、Ada、ML以及其他一些语言的思想。当时我知道Modula-2，还知道至少一打别的语言，但我回想不起来Modula-2对我产生了什么直接的影响。为了回答“为什么C++会是这样（以及为什么不这样）”之类的问题，我撰写了《The Design and Evolution of C++》一书。也就是说，这本书记录了导致C++现状的设计决策、原则以及折中权衡，我推荐对此类问题感兴趣的读者阅读这本书。

Elden Nelson：您预期对C++做哪些增强，会不会删掉一些东西？

Bjarne Stroustrup：很不幸，虽然有一些东西真的可以扔掉，但恐怕很难删掉任何东西。第一个应该抛弃的东西就是C风格的转型机制和类型截断转换（narrowing conversions）。就算不禁止，编译器的作者们至少也应该对这种行为给与强烈的警告。我希望能用标准库中的vector之类的东西彻底取代数组，但这显然是行不通的。不过如果程序员们能主动在所有应用编程中使用vector来代替数组，就会立刻受益匪浅。关键是你不必再使用C++中最复杂难缠的技巧了，现在有优秀得多的替代方案。我没打算去掉任何大的特性。特别是那些把C++与C区别开来的主要特性恐怕没法风平浪静地被抛掉。通常问这些问题的人是希望我挑出诸如多继承、异常、模板等

vim-ctags-taglist-netrw (249)
性能测试命令字段解释 (213)
正则表达式 (211)
SystemTap (195)

评论排行

为什么需要auto_ptr_ref (1)
valgrind内存检查 (0)
cgroups (0)
linux下如何配置openvpn (0)
git操作命令 (0)
thrift (0)
Apache开源软件 (0)
vim-ctags-taglist-netrw (0)
vim常用配置 (0)
shell按行读取字符串，并 (0)

推荐文章

* 挣扎与彷徨--我的2014
* 校招回忆录---小米篇
* Android UI-自定义日历控件
* 30岁程序员回顾人生、展望未来
* 2014年终总结，我决定要实现的三个目标
* Android 启动问题——黑屏 死机解决方法

最新评论

为什么需要auto_ptr_ref
zjufirefly: explicit auto_ptr_ref(_Tp1* __p): _M_ptr(__p) {}...

机制来接受批判。所以在此我想大声讲清楚，我认为多继承机制对于一门具有继承机制的静态类型语言来说是必需的，异常机制是在大系统中对付错误的恰当的方法，模板机制对于类型安全、优雅和高效的程序设计来说不可或缺。我们可以在语言细节方面对这些机制吹毛求疵，但在大的方面，这些基本概念都必须坚持。现在我们仍在学习标准C++，也正在标准所提供的特性基础上发展出更新且更有意思的编程技术。特别是人们刚刚开始使用STL和异常机制，还有很多高效强大的技术鲜为人知，所以大可不必急匆匆地跑去增加什么新机制。我认为当前的重点是提供很多新的、比以前更加精致的且更有用的库，这方面潜力巨大。例如，如果有一个能被广泛使用的、更精致的支持并发程序设计的库，那将是一大福音 — C风格的线程库实在不理想。我们也就可以与各种其他的系统比如SQL以及不同的组件模型更好地契合起来。在优雅高效的库的开发方面，数值计算领域的人们看起来已经走到了前面（例如Blitz++、POOMA和MTL，请访问http://www.research.att.com/~bs/C++.html）。有了足够的经验之后，我们就可以更好地决定什么能够（也应该）被标准化。

Elden Nelson：我们正不可避免地走向一个以Web为中心、分布式计算为主流的时代。那么您觉得C++还能维持其地位吗？程序员们可不可能把若干种专用语言（比如Perl、Javascript）综合运用以彻底取代某一种通用语言？为了配合新的计算模式，C++或标准库应该做出怎样的调整？

Bjarne Stroustrup：从来没有哪一种语言能适合所有的工作，恐怕以后也不会有。实际系统通常是用多种语言和工具构造起来的。C++只是想成为若干语言和工具中的一个，当某些专用语言在其领域里特别突出时，它们可以与C++互为补充。也就是说，我觉得如果大多数现在的专用语言能借助特定领域的C++库共同工作的话，它们会表现得更出色，而脚本语言通常导致难以维护的代码，这或许跟语言选择关系不大，可能更是因为急着想将产品尽快推向市场。如此一来，哪还有什么精力考虑程序结构、伸缩性和可维护性方面的问题？我不敢肯定未来的代码是否真的会以Web为中心，就算是直接处理Web的系统也主要是由处理本地资源（比如IP连接）的程序模块构成的。地理上的分布性以及服务器软件对于并发机制的高度依赖对于系统的构建者来说的确是个挑战。针对上述问题的一些库已经出现，也许我们将会看到它们最终得以标准化。当然，一些原语操作和保证规则应该被加到核心语言中以提供对这些库的更佳支持。总的来说，对于Web和网络，我们非常需要一个真正的系统/网络级的安全模型。指望下载的“以JavaScript之类的语言编写”的脚本来实现这个模型无异于痴人说梦。请注意，我也并没有宣称C++提供了这个问题的解决方案。C++被设计为对所有系统资源提供高效的访问，而不是为了防止被欺骗。

Elden Nelson：您认为C++未来的走向如何？在接下来的10年里它会衰落吗？或者基本保持现状，或者演化为某种不一样的东西？

Bjarne Stroustrup：C++有着极美好的未来。用它你能写出伟大的代码。不管被多少敌意的宣传所攻击，C++仍将是开发高性能、高复杂度系统的最佳语言。据我所知，还没有哪种语言能象C++这样，将通用性、效率和优雅有机结合。我没看到C++有衰落的迹象。在我能预见的未来里，它的用途还会不断增长。当然，在未来的十年里我们会看到一些变化，但不会象这篇访谈中的这套问题所暗示的那么多。跟每一种语言一样，C++也会不断演化。“语言专家们”要求改进的喧嚣声震耳欲聋，但是系统开发者们的基本请求是保持稳定。C++会改进，但这些改进将是“经验”的结果而非对“狂热”的反应。为了更高效地使用一些新的编程技术，比如通用编程技术，可能会增加一些小特性。会有大量的库涌现，我预期会出现一些新颖的便利设施以支持更好的库。我希望新的扩展主要集中在支持抽象方面的一般特性，而不是为支持某些特殊任务的特定机制。打个比方，属性（properties）是一个有用的应用层的概念，但我不认为在一种通用编程语言中有它的容身之地。用标准C++的一组类可以很容易地支持这一概念。如果我们感觉那族类对于“属性”这一概念的支持不合口味，我们也不应该立刻跑去在语言里增加属性机制，而是仔细考虑如何改进类和模板以帮助库设计人员尽可能接近“属性”这个概念。也许通过改进函数对象机制能够给这个问题一个满意的答复。为了使C++在接下来的十几年中保持生命力，很基本的一点就是不要让标准C++赶什么学术或商业时髦。人们要求增加的特性中很大一部份通过使用现有的标准C++开发新库的方式都可以实现。还有，事实上人们渴望得到的很多奇妙的特性已经包含于标准C++之中，并且被所有最新版本的编译器所支持。对许多程序员来说，提高代码质量的最佳途径不是追求什么语言扩展，而是静下心来品味最新的C++技术书籍。

Elden Nelson：对于当前脚本语言的兴旺态势您怎么看？特别是Python，与C++相比，它似乎提供了一种更简单的OO技术学习途径。

Bjarne Stroustrup：有些语言很不错。比如Python，我很喜欢。但是我认为你从不同的语言中学到的OO技术是不完全相同的。当然，每一个职业程序员都要通晓几门语言，并且应该意识到，在不同的语言之中，编程和设计技术有着显著不同。在我看来，用脚本语言建造的系统与用C++那样的通用语言建造的系统大不相同。从两类语言中学到的技术区别明显。不存在“可以满足绝大多数高效系统构建所需”的公共OO技术子集。

Elden Nelson：有没有计划对标准C++语言进行扩充或改进，从而为分布式计算提供更好的支持？

Bjarne Stroustrup：没有，我也不认为有这个必要。用更好的库就差不多能解决问题了。充其量为了支持这类的库，我们可能会增加一些低级的“原操作（primitives）”或“保证（guarantees）”。

Elden Nelson：将来C++有没有可能定义一个可移植的二进制接口？

Bjarne Stroustrup：如果你说的“可移植”是指跨硬件和跨操作系统，我想答案是no。我们当然可以设计一个解

释器或者虚拟机什么的，但这样一来，由于无法以最优方式访问系统资源，C++的能力就会受到削弱。我希望在不远的将来能够看见平台ABIs（platform ABIs）。例如，有人正在努力为Intel新的IA64架构定义C++ ABI，我想这些努力会得到用户社群的强烈支持。能够把一台PC上不同编译器产生的代码连接（link）在一起是一件美妙的事。

Elden Nelson：您目前是否正在为其他新语言开展工作？

Bjarne Stroustrup：没有。我还在学习如何使用标准C++，并且进行一些分布式计算的试验。我认为编程本身远比编程语言细节有趣。我认为只有当你有一些东西无法用已有语言合理表达时，才需要考虑设计一门新语言。对我来说，绝大部分工作都可以通过C++很好地完成。

Elden Nelson：以“后知之明”的角度来看，您是否认为“使一个成员函数默认为非虚函数”是一个明智的决定？假设有机会改变，您会改变这个决定吗？

Bjarne Stroustrup：也是也不是。使C++保持生命力的要素之一即是“零开销规则”：你无需为你不用的功能付出代价。使一个成员函数默认为非虚函数会违反这条规则，还会为“提供高效的具体类型（concrete types）”增加难度。对于那些认为“类”即为存在于复杂层次结构中的大家伙的人们来说，默认为virtual是显然的。一般来讲，虚函数不适合那些关键的小而具体类型，例如复数、points、vectors、lists，以及函数对象。对于这样的类型来说，如下方面至关重要：表达的紧凑性、基本操作的内联化、访问的直接性、堆栈的配置，还有对“重载函数不会对语义造成不希望的修改”的保证。此外，如果默认为virtual的话，你将需要一个non-virtual或final关键字，而且当过度使用它时会导致扩展性问题。在语言设计里，的确没有免费午餐。

Elden Nelson：在您看来，经由IEEE执行的标准化过程对C++语言的完备性、灵活性以及能力等方面产生了怎样的影响？

Bjarne Stroustrup：对于C++来说，ISO标准化过去以和现在都是重要的。其重中之重在于，标准委员会为技术人员提供了一个探讨技术问题的“中立的舞台”。还有什么别的地方能够让来自于相互竞争的机构（比如Microsoft、IBM、Borland/Inprise以及Sun等）的用户和编译器编写者们出于对用户利益着想，坐到一起合作共事呢？ISO的工作是民主的，是以大多数人的意见为基础的。为了达成一致的意见是需要时间的，但这样的努力非常值得。否则将导致语言的定义只为某一家公司（或少数几家公司）的利益服务。由ISO工作而形成的标准C++比以前任何版本的C++都更加接近我的理想。标准C++的“异常”与我自己定义的几乎相同，模板更具灵活性，名字空间和运行时类型信息也被添加进来。从对编程风格（你也可称之为“范型”）提供支持的角度来看，其余内容皆属于细节问题，自然而然，标准委员会工作的主要部分便是精确定义这些细节。鉴于接近标准的编译器可以广泛获得，人们试验那些新功能的时机已经来临。在几年之前很多东西还没有成为现实，如今可以在实际应用中使用它们了。那些对于大多数人来说仅能在语言定义上看到的技术，现在已被开发出来。例如STL（标准库中容器和算法的框架）就是一个有意思的新技术的优秀资源。当然了，在接下来的关键项目中，你不应该冲到最前面使用所有语言特性和所有新技术，但是，是开始学习新语言特性和新标准库并试验它们哪些适合你哪些不适合你的时候了。假如需要文档资料，你可以通过ANSI以18美元购得C++标准（参见www.research.att.com/~bs/C++.html），也可以免费获得接近标准的草案。但是，这份标准并不是教本。我向有一定经验的程序员推荐我的《The C++ Programming Language》第三版，这本书以更易理解的方式讲述了完整的语言和标准库，它还阐明了C++支持的很多基本设计和编程技术。然而，即便这本书也不适合初学者阅读，因此，请先浏览我的个人主页（www.research.att.com/~bs/）以了解我的写作风格和详细程度能否满足你的需要。

Elden Nelson：在不少流行领域，C++正渐渐失去光芒，因为它要求人们花很大的精力去对付一些很基本的工作，比如管理内存（因为没有垃圾回收机制），管理模块之间的依赖性（因为无法创建包（packages）），管理组件的版本。C++缺乏一些现代语言已经视为标准的特性，谣传中最酷的Java语言试图解决这些问题，那么解决这些问题是否会导致C++的发展背离其根本宗旨呢？C++应该怎样发展以保证我们在这种语言上的投资能有合理的回报，而不是被迫从头学用另一种语言？

Bjarne Stroustrup：我倒还没有注意到C++比以前用的少了，相反，我看到的指标表明C++的使用还在稳步上升。只不过这种基数很大的稳定增长以及在标准性、移植性和库方面的不断提高并没有造成什么具有欺骗性的新闻效应而已。我认为你所说的“失去光芒”只不过是市场营销和新闻意义上的现象。如果你需要垃圾回收机制的话，你可以在C++应用程序中插入一个垃圾回收器。有不少免费的和商业的垃圾回收器已经在重要的实践中被证明非常出色（可以参见www.research.att.com/~bs/C++.html）。如果你不想使用垃圾回收机制也没关系，你可以使用标准容器类，它们大大减少了对于显式分配和回收内存的需要。这样，通过使用现代的库所支持的现代的编程风格，你就能避免绝大多数内存管理问题。同样的技术还可以用来避免一般资源的管理问题。并不是只有内存才会泄漏，线程句柄、文件、互斥锁、网络连接等都是重要的资源，为了建立可靠的系统，它们必须被正确地管理。如果你以为有了自动垃圾回收机制就可以解决所有资源管理问题，那你最好赶快从美梦中醒来。C++提供了一些机制来管理常见资源。关键的手段 — 资源获取即初始化（resource acquisition is initialization）依赖于函数对象来管理生存期问题。语言中关于对象的部分构造规则和异常机制对这项技术提供了一般性的支持。关于异常处理技术的讨论，请参阅The C++ Programming Language (Special Edition) 的新附录“Standard-Library

Exception Safety”，它也可以从我的Web站点访问到(www.research.att.com/~bs/3rd_safe0.html)。某些语言的狂热支持者总是用讽刺的手法来描述C++，然而C++实际上要好得多。特别是我认为很多其他特性已经被吹嘘过度了，而在C++中，通常这些特性能够很容易地被模拟出来。相反，新语言总有一种“在损害一般性的情况下”添加新特性的倾向，这也是一门新语言从诞生到被接受为一种用于常见计算的有用的工具，其体积和复杂度通常会增加两到三倍的原因之一。目前，使用C++的个人和组织可以进行的最佳投资就是去更好地理解标准C++和现代的C++设计和编程技术。大多数人使用C++的方式实际上停留在80年代中期的水平，甚至比那更落后。精确区分语言和系统/平台之间的责任是一个困难的问题，我的观点是，它们之间应该有一个明显的分界线，并且，依赖关系应该尽可能地置身于语言之外，系统相关和系统依赖的库才是解决系统依赖性的地方，而语言并不是。我不认为组件版本管理之类的问题应该由编程语言来解决，这是一个系统范畴的问题，在语言里应该通过提供用于系统访问的适当的库来解决。C++有这样的机制。解决这样的问题并不会违背我对C++所奉行的理念。但在另一方面，给C++增加很多特殊的特性会使C++偏离轨道，而且在保持可移植性和平台独立性方面也是一个倒退。

Elden Nelson：如果为库中某个基类定义一个派生类，要想覆写（override）基类的一个虚函数，你就必须得到这个基类的源代码，以便获知是否需要调用基类中该函数的实现代码，您是否认为C++类库在这方面有那么一些失败？

Bjarne Stroustrup：唉，造成某些C++类库的这种缺陷的原因是，其设计者认为必须将此问题定义到他们的库中，并且一些用户认为他们必须以这种方式来使用库。这真是差劲的设计，是对C++差劲的使用！如果你不想依赖基类的数据或者代码，就不要把它们放到基类中，这正是抽象类的使命。考虑如下代码：

```
class Reader
{
public:
    virtual bool empty() = 0;
    virtual Element get() = 0;
};
```

它为所有派生类提供了“Reader”功能的一个接口。Reader的用户完全不依赖于那些派生类的实现细节。尤其是当某个派生类的代码改动时客户代码并不需要重新编译。还有，用户可以同时使用Reader类的多种不同实现（也就是说，可以同时使用多个不同的派生自Reader的类）。从1989年发布的2.0版起，抽象类就已被直接支持，并且这项技术/风格总是随时可用的。这些历史以及语言设计考虑在《D&E》中皆有描述，自然而然，The C++ Programming Language解释了在什么情况下以及该怎么使用抽象类。随便说一句，通过继承一个抽象接口类，再继承某个类层次结构中一个具体类（以获得它提供的有用功能的实现代码），是多重继承的一个最简单、最明显的用法：

```
class My_class : public Interface, protected Implementation
{
// override virtual functions from Interface,
// implementing the overriding functions
// using facilities offered by Implementation
// Where needed, also override virtual functions
// from Implementation
}
```

我认为抽象类是一项远远没被充分利用的C++特性。程序员总是设计层次深深的继承体系，还在基类中添加大量数据和代码。有时这么做是有道理的，但在大的系统接口设计中，你更需要的是程序不同部分间的独立性，由抽象类提供的纯粹的接口往往便是更好的设计选择。较老的C++库的另一个问题在于，其设计者还不能使用模板。有些情况下继承的使用毫无必要或缘于无知，因为将类型参数化更加合适。

Elden Nelson：为什么在C++中没有关键词“super”？

Bjarne Stroustrup：因为在C++中，它既非“必要”也不“充分”。“super”之所以不必要，是因为Base::f这种符号允许程序员表达f是“Base的一个member”或是“Base的一个base”。“super”之所以不“充分”，是因为你需要表达f来源于Base1而非Base2。

Elden Nelson：一些厂商已经或正在对他们的C++编译器进行修改，以支持平台相关的语言扩展。您对此有什么看法，您认为这会有什么作用？

Bjarne Stroustrup：我认为平台相关的语言扩展应该最小化，当必须进行扩展时，其设计应该局部化于库中。当然，比起我自己，平台供应商们倾向于认为要进行更多必不可少的扩展。他们还倾向于使扩展弥漫于应用代码之中，从而使用户很难更换供应商。身为一个注重可移植性的用户，我对这种套牢用户的伎俩表示遗憾。出于用户的利益着想，理想状态必须是可移植的，且平台相关的代码应被隔离在应用代码的一些特别段落。可移植性，以及不因厂商的奇思异想而改变的语义，是一门标准语言比专有语言优越之处。我认为C++供应商们应该意识到这可以成为一个竞争优势，并将专有扩展及扩展所引起的冲击最小化。假如你想用Java和Visual Basic那样的专

有语言，你知道到哪儿去找它们。

Elden Nelson：您对目前花色繁多的C++编译器有什么看法？请不要因为这是《Visual C++ Developers Journal》的采访而影响您的看法。

Bjarne Stroustrup：它们正变得越来越好。这包括所有C++编译器。我通常使用六种不同的C++编译器。几年前我是做不到这一点的，当时一些被广泛使用的C++编译器还不能完全满足我的需要。我乐意让"这是《Visual C++ Developers Journal》的采访"的事实影响我要说的话。因为这正是鼓励微软更加遵循标准的绝佳场合！VC++已得到改进，但是凭微软所拥有的资源，还可以进一步提高符合标准的程度，并为核心语言特性和标准库提供更高质量的支持。例如，对于目前大多数C++编译器来说，关于模板的出错提示信息仍有较大改进余地。在遵循标准方面，现在的情况比以前要好多了，但在VC++中，我们仍然不能使用模板友元和模板部分特化功能。我非常乐意看到有人实现了对模板的分别编译 — 从Cfront时代起，它就是一项我想用而无法用的重要功能。如果VC++能为新手学用标准功能打开方面之门，那将是一件好事。下边是那个没什么实际价值的"第一个"程序：

```
#include<iostream>
int main()
{
    std::cout << "Hello, new world/n";
}
```

在我看来，略微提高投在标准库方面的资源，而不是提高投在专有扩展和专有功能方面的资源，将是微软帮助最大数量的程序员最廉价的方法。通常产生的代码的性能挺好。基于用户社群不同的关注方面，各种编译器倾向于各有区别。我认为最重大的收益来源于对标准库的调整。例如，从一个istream中将一个字符序列读入一个string，就是一个值得优化的操作，不为别的，它可以避免程序员去摆弄诸如字符读取、显式缓冲、空间分配和指针之类的玩艺。举个例子，下面的代码既优雅又不失效率：

```
vector<string> vs;
string terminator = "endend";
string s;
while (my_input>>s && s!=terminator) vs.push_back(s);
```

请参阅我写的《把标准C++当作一门新语言来学习》一文中关于风格和效率的议题（我的"publications"页面有链接）。

Elden Nelson：标准C++并没有任何方式定义支持并发、持久化和基于组件的编程，这导致了互不兼容的、平台相关的框架（例如CORBA、DCOM和SOM等）的繁殖，所有这些都有违直觉、杂乱无章，这不正表明标准C++应该为并发（尤其是线程）和组件对象模型提供直接支持吗？

Bjarne Stroustrup："并发"和"组件对象模型"无疑是摆在当今所有语言的设计者面前的巨大挑战。不幸的是，这些挑战往往出于政治原因而非技术因素，有太多的金钱使之走样。用户的理想语言应该是一门直接支持广泛的并发需求并对组件这一通用概念提供良好支持的语言。在理想状态下，程序员不费吹灰之力即可建立起语言功能与给定的组件架构之间的映射，这是"保持应用程序与它们所运行的硬件和操作系统之间适当的独立性"的理想的一个现代表现方式。这可能与"组件可通过任意编程语言编写并且组件都应设计为可被任意语言无痛复用"的理想相悖。组件模型的一些支持者主张，程序员应该考虑编程语言的可互换性，为给定的对象模型专门编写代码，此模型的基本元素在这些代码中高度可见。我并不赞同这一观点。然而，我猜测这两种理想可以达成一个极好的折衷。在一个"幼稚"的程序员看来，一个被调用的组件就是一个美其名曰"抽象类"的东西：你从某处获得一个接口，然后通过一个句柄（handle）来使用它，正像你通过句柄、指针或是引用使用其他抽象类一样。这样便可以向用户整洁优雅地展示组件模型，无需显式使用语言扩展，也无需"幼稚用户"明确知道到底在用哪种组件模型。例如，一个句柄类可以带有一个string参数的服务名字：

```
Printer_handle ph("d208d");
```

此处的d208d恰巧是我办公搂下大厅的打印机的名字。这样一来，使用与"通过标准I/O为用户屏蔽操作系统和硬件"完全相同的技术，便可为"幼稚用户"和大型程序的绝大多数模块屏蔽掉系统依赖性。"幼稚用户"应该永远无需显式处理组件模型的"魔法" — 比如唯一组件标识符（unique component identifiers）。更加复杂的使用需要更多的知识，更常需要编写平台依赖的代码，还可能不得不明确使用那些为支持某种组件模型而提供的语言扩展。这里关键之处在于，对组件模型的"曝光"可以而且应该是缓和的，还应依赖于此应用对模型所提供的功能进行直接操纵的需要程度。在大多数情况下，我希望扮演"幼稚用户"角色。组件模型提供了很多东西，但目前它们与C++的"捆绑"迫使程序员为了使用它们而不得不做太多的工作，迫使程序员要对所使用的模型了如指掌，还常常从程序员手中剥夺了丰富的编程语言功能。由此便有这样一种趋势 — 鼓励程序员以该模型支持的所有语言的一个公共子集来编写代码。语言中立性所提出的需求并不意味着所有接口对于所有语言都必须可用。也就是说，并不要求必须以该模型支持的所有语言的最小公共特性集来表达接口。当我编写C++程序时，我希望使用标准库中的vector、string、map以及list这样的C++便利设施，以便在组件之间进行通讯。我并不想降低抽象层次，转而使用int*、char*、void*以及转型操作（casts）。既然组件意味着可以应用于所有语言，因此，我既可以提供一

个工作于较低层次的附加接口（使之为另一个抽象类，这正是适合用多重继承处理的一种情况），也可以其他语言方式编写例程以提供对我的C++抽象功能的存取。别误会，我并不是说你今天就可以在你的机器上如此这般做事，我的意思是说，如果使用组件模型就有可能获得此等程度的优雅性和灵活性，并且，在尽量最小化系统相关的语言扩展“侵入”的前提下提供组件模型是可能的。我相信这应该是用户的理想。我将鼓励组件模型的供应商们出于其用户的利益而努力接近这个理想。目前C++的绑定物“侵入”程度过于严重，向用户暴露的组件模型细节过多。自然而然，并发（concurrency）也将闪亮登场。我再次声明，以相对“非侵入”的方式在库上大做一番文章即可。在直接语言支持方面仅停留在原语级别，且大多数情况下对大部分用户来说不可见。另一个需要解决的问题是安全性。我并非只意指类型安全，而是指对系统一致性的保证以及对资源访问的控制。我怀疑没有操作系统和组件模型的支持是做不到这一点的。这是一个热门研究领域。

Elden Nelson：那些业已存在的事实上的标准，比如pthreads，将有多大可能成为标准库的一个组成部分（虽然已拥有一个面向对象接口）？毕竟，C标准库的主要部分都是在考虑Unix的前提下设计的，而且其设计者并不试图保持平台中立。

Bjarne Stroustrup：这是某种“并发”支持进入标准的一个良好契机。平台中立性将会被严肃地解决，因为标准委员会的一个美妙之处在于它是由使用许多不同平台的代表组成，而且委员会是在“争取一致意见”的要求下运作的。

Elden Nelson：在另一次采访中，您将C的声明语法定义为“一个失败的尝试”。但是，这种语法结构已形成27年了（可能还不止），为什么您认为它是有问题的（撇开其麻烦的语法不谈）？

Bjarne Stroustrup：我正是因其麻烦的语法才认为它有问题的。能够表达诸如此类的想法是有益而且必要的：“p是一个指向一个数组的指针，此数组有10个元素，每一个元素皆为一个函数指针，被指向的函数接受两个整型参数并返回一个bool型值”，然而，

```
bool (*(*p)[10])(int,int);
```

并非表达此意的一种明显的方式。实际上，我不得不使用一个typedef才能妥当表达：

```
typedef bool (*Comparison)(int,int);
Comparison (*p)[10];
```

即便在此处，包围指针的圆括号也似乎是多余的，但事实并非如此，因为任何线性符号（linear notation）的优先级要比非线性符号来得高。例如，请试着从左到右阅读下边这行代码：

```
p: *[10]*(int,int)bool
```

（“一个指向数组的指针，此数组有10个元素，每一个元素皆为一个函数指针，被指向的函数带有两个整型参数并返回一个bool型值”）如果你既需要声明语法的线性化，又想要C风格的声明语法和表达式语法等价，所有操作符必须是“后缀型”（或者为“前缀型”，但你不能同时拥有这两种语法）。但是，“业已为人熟知”的力量很强大。让我们以英语为例作为对比。或多或少，我们早就欣然接受了关于“to be”的荒谬的语法规则（am、are、is、been、was、were……），而所有简化它们的企图都被视为“不尊重”或者说得好听点 — 幽默。这真是一个奇怪的世界，过去将来都是如此。也就是说，编程语言还是远比自然语言来得简单，因此，我希望某天我们可以使用一种更好的声明语法，到那时，我们便会纳闷：“那些老家伙们怎么能忍受同这些荒谬的玩艺朝夕相处呢？”

Elden Nelson：今日世界的处理器速度要比C++刚诞生时的处理器快上数千倍，因此对于很多开发商来说，目前软件开发时间显得比软件运行时间更有价值。C++应该如何改变（若有改变的必要的话）以应对这项根本性的转变？

Bjarne Stroustrup：程序员的时间一直被认为很宝贵，而我们对计算机的时间要求却超出千倍。鉴于系统性能的不断提高，我认为C++不应该改动，不过，许多程序员却可以从系统性能提高中获益，因为他们可以对低阶效率问题放松一点警惕，而将重点集中到高阶结构和正确性上。这样做意味着更加注重抽象，且要以“一门高阶语言”的眼光来看待C++。由此可受益良多：编程更加容易，移植更加方便，维护更加轻松。甚至还可以获得性能上的好处。现代代码中多数严重的效率问题源于对系统设施和算法的欠妥运用。每当看到有人使用数组、宏、转型以及指针把C++程序写得一团糟时，我都感到不寒而栗。这种程序看起来就像用C++语法写成的汇编代码。仅仅用标准库的vector和string来代替数组，便是一个巨大进步。还有，并非所有程序都运行在一台有显示屏、用户等待反馈的这种传统型计算机上。实际上，大多数程序运行于嵌入式系统中以控制我们的电话、汽车、照相机以及气泵。那些情形往往对资源有严格的限制，C++编写紧凑、快速代码的能力尤显重要。另外，许多服务器应用和科学应用是在硬件资源受限的环境下工作的。比方说，假如你使用C++来实现一个气候模型，你必须要有一个如同优化的Fortran一般的库来执行计算。

Elden Nelson：将在2003年出台的标准的下一个修订版中，会有哪些特性加入到C++之中？您认为哪些特性属于当务之急？

Bjarne Stroustrup：委员会已将此话题的讨论延期，直到提交一份关于此内容的缺陷报告，还要发布一份有关性能议题的技术报告，之后才能继续进行。与此同时，鼓励人们积累运用目前标准的实践经验，以及进行有关改进的试验。我认为这是一种鼓励“稳定性”的良好方式，由此也可以避免削弱委员会可以依赖的时间和技巧方面的有限资源。别忘了委员会是完全由拥有“日常工作”的志愿者组成的。有鉴于此，我可不愿让人看到我想以一些特

定的私愿而凌驾于于委员会之上。你可以在这篇采访的其他部分获悉我对此问题的概括观点。

Elden Nelson：刚获批准的C9X标准在ISO C中加入了好几项C++尚未支持的构造。例如long long数据类型和restrict指针，还有其他几项。它们将来会被加入C++吗（果真如此，C++还将需要“restrict引用”，对吧）？您认为编译器厂商们应该等到标准的下一个修订版出台，还是应该以非标准扩展的形式将它们实现？

Bjarne Stroustrup：C99新加的功能会在C++环境中予以考虑。无庸置疑，其中一些特性会被采纳 — 正像C99中也包含了一些C++特性一样。然而，这并不像某些人想象的那么容易。我认为，把C99的全部功能添作C++的扩展同时还要保持与当前C++标准兼容，是不可能做到的。一个更加严肃而基础的问题是“语言设计哲学”。对C功能的添加是为了提供一些专门功能（非常类似于Fortran对数值计算的支持）。我认为这种方式过时且与C++以及多数其他现代语言所采取的途径相悖，后者更注重语言的抽象机制，这些设施允许用户和库构建者在不增加核心语言复杂性的前提下提供更广范围的专门功能。C99添加了数据类型（long long和复数）、特别方式的初始化、转换规则以及一个有着特别语法形式的内建数组。这与C++采取的方式形成了鲜明对比。C++通过构造函数指定初始化和转换操作，通过标准库的类提供新数据类型。C采取的方式增加了基础语言的复杂度，加重了内建类型的不规则性，并和C++以标准库方式提供的东西重叠。这就意味着假如你把C99的功能加入到C++，用户就不得不在“C99风格的动态数组”和“C++的vector”以及“C99风格的复数”和“C++的复数”之间做出抉择。我认为这是C与C++发展过程中的一个协调上的失败，并对C++社群乃至所有为C/C++社群着想的人们造成了严重负担。然而，从网上的消息来判断（这一贯是一种不可靠的行为），有些观点认为难以解决的兼容性问题象征着C语言“独立”的胜利，我却认为C与C++双方委员会亟待更深入的协调，用户社群和C/C++编译器供应商们可以表达出自己的观点以改善现状。restrict是一个相对简单的问题。restrict可以作为非标准扩展加入到C++而不会造成什么伤害。毕竟，只要利用宏将restrict替代为空就可以将使用它的程序转变成合法的C++程序。你的看法是对的，C++变量还需要提供受限的引用（restricted references）。

restrict为什么不是C++的一部分有下述三个原因：

- 委员会认为C++已经加入了足够的内容，因此应该将新特性最小化。诸如restrict之类的单独特性在被加入标准C++之前应该先经最需要它的用户社群充分试验。
- 总体来说，restrict特性并不能总被编译器验证具有安全性，因此委员会认为需要做更多的工作，以确认是否能找到一项表现更佳的替代方案。
- C++标准库提供了valarray以支持高性能Fortran-like向量计算。

重新考虑restrict的原因有：

- 随着更多的处理器要求如“long pipelines”和“parallel execution”之类的架构特性，restrict显得越来越重要。委员会最初讨论restrict时，在C++社群使用所有处理器中，只有大约5%具备使得restrict有价值的高性能特性。时至今日，我怀疑这个比例已接近95%了。
- 据我所知，我们还没有找到一个可行的方法，使得在C++环境中实现自身静态类型检查。
- 在C++标准库中，valarray的地位还没到“举足轻重”的地步。果真如此，valarray与restrict共存便不会有问题。因此，更主要的问题在于C同C++ ANSI和ISO委员会之间的协调。无论如何，所有C和C++编译器供应厂商都面临一个两难境地：遵从标准的一个分支乎？使全部功能在所有环境下都可用乎？

我认为厂商可以通过提供大量的编译选项以从这个窘境解脱，这样便把牌打到了用户那儿。最终，用户们会因感到不方便而要求标准化。我希望为C++程序提供反映C++标准的默认编译选项，而为C程序提供反映C标准的默认编译选项。总之，我鼓励编译器厂商将他们的C++编译器的默认选项设置为“遵从ISO C++标准”。不幸的是，在这方面，Windows世界还有一段路要走。

Elden Nelson：标准C++推出有些日子了，Java也在大踏步地往前赶并取得了显著的进步，您怎么比较Java与C++？您认为Java想要变成像C++一样“好”的语言还需要做些什么？您希望C++从Java身上学到什么经验？有没有什么Java（或其他语言）所具备的特性您认为是可以为C++所吸纳的？

Bjarne Stroustrup：我不比较语言。做好这项工作是十分困难的，而且很少具有专业水准。我认为C++的进步将会主要以它的用户在使用中遇到的问题以及其自身逻辑为基础。当然，其他语言中的某些思想也会被考虑，但不能被简单的移花接木过来。你必须审视某个功能所支持的技术和概念，并且找到在C++中支持它们的最佳方案。有时最好的选择是综合使用几种语言。毕竟没有任何一种语言是放之四海而皆优的。C++现在是将来也继续会是在广泛应用领域中最好的语言之一。但是，我们不能被拉下水，不能把所有可能的特性都加到C++里面来向大众献媚。我认为Java和C++现在和将来都会是十分不同的语言，它们语法相似，但背后的对象模型明显不同。对于我来说，一个很重要的区别是C++有一个ISO标准而Java是一个专有语言。

Elden Nelson：在Java刚刚出现的那几年，有很多欺骗性的言论说它将会是“终极语言”，会取代C++。您觉得在过去两三年里Java对C++开发社群产生了什么影响？

Bjarne Stroustrup：到现在关于Java的不实之辞也还随处可见。暂且不提Java在过去5年间的创纪录的发展，狂热的大众似乎认为Java将最终取代的不仅仅是C++，而且还有所有其他编程语言。但在另一方面，C++的使用

仍在继续增长。我不认为Java对于C++的影响已经使得人们转而把本打算用来创造更好的C++工具库的资源调过去开发Java工具库。Java对于学习编程的人来说没有太多的新东西，所以对于C++的定义也没什么影响。在那个领域，Java还得努力追赶。比方说，我认为Sun迟早会往Java里加入类似模板的机制。人们应该认识到C++和Java的目标是何等的不同。以C++的设计理念来衡量Java或以Java的设计理念来衡量C++，得出的结论都不会太好。在访谈末尾，或许我该表明态度：C++仍然是我喜爱的语言，还没有哪种语言能够像它那样在如此广泛的应用领域和平台上编写出既高效又优雅的代码。

C++ 热点问题一席谈

来源：荣耀/刘未鹏 译 作者：Bjarne Stroustrup 等级：精品
发布于2006-10-22 12:46 被读923次 【字体：大 中 小】

C++热点问题一席谈
— Bjarne Stroustrup 2005新春专访
荣耀 访 荣耀/刘未鹏 译

荣耀：Herb Sutter和Stan Lippman目前正在微软主持C++/CLI的设计工作，意图将动态的、基于组件的.NET编程模型和ISO C++集成在一起。您对此有何评价？您认为C++需要.NET吗？您认为C++/CLI会取得成功吗？

Bjarne：不，C++根本不需要.NET，C++只需要最小限度的运行时支持，用于new/delete、异常处理以及RTTI等，而且仅当你使用这些特性时才需要。C++程序通常可以使用每一分可用的资源，在硬件上直接跑。C++的这些能力使其非常适合于系统级编程以及嵌入式系统任务。当然，也有些C++应用需要.NET，比如那些为了和微软.NET框架和服务紧密集成而专门设计的应用。然而，C++语言和标准库的宗旨是远离这些平台相关性的纠缠。另一方面，许多.NET设施都依赖于C++，因为除了C++之外，再也找不到更通用、更高效的语言来很好地完成这个任务，从这个意义上说，.NET需要C++。

从“很多人将会使用它”这个意义上来说，C++/CLI是会成功的。使用.NET CLI，开发者选择甚少，而C++则是最佳选择之一，而且很明显在Windows上也是，因为微软给予C++最好的支持。话虽如此，我仍然倾向于在设计系统时保持良好的移植性，而将对平台相关或专有特性的使用限制在特定的代码块中，并使用以ISO标准C++所表达的接口去访问它们。

荣耀：尽管我现在相信这是一个毫无意义的问题，不过我想我最好还是澄清一下。当我说“C++需要.NET吗？”，我的意思是想问“我们需要.NET来使C++更普及吗？”。这就好比问“世界和平需要美国吗？”，或者，“我们需要美国来维护世界和平吗？”。当然了，我们都不喜欢讨论政治性话题，也许这个比方很不合适。

Bjarne：政治关乎可行性。从这个意义上来说，我们必须考虑政治，而你的问题当然也是合理的。鉴于微软在软件领域的地位以及它对.NET强大而完全的支持（将.NET的系统接口以CLI来表达），.NET的地位会变得很重要。要想在微软的世界里玩得转的话，C++必须很好地绑定到.NET。事实上这种绑定（C++/CLI）已经建立了，微软还为之申请了ECMA标准。.NET跟我理想中的尚有些差距，而C++/CLI如果让我来设计的话可能也不会是这个样子，然而不可否认的是，C++/CLI在.NET上的确是能力非常强的语言，也是迄今为止为.NET设计的语言中能力最强的。如果微软未考虑将C++作为.NET上的关键语言之一，或者.NET平台上的应用创建者没有坚持对C++提供第一流支持的话，情况会糟很多。

所以，为了能够在微软的世界里流行，C++需要一个良好的CLI绑定。我仍然鼓励人们将C++/CLI仅仅视作一个绑定物。这就是说，把对C++/CLI特性的使用隔离到一些特定的区域里，并且通过ISO标准C++设施去访问它们。C++/CLI的一些设施使其成为一门具有吸引力的语言，然而和标准C++却相去甚远，所以如果你在代码中到处使用这些C++/CLI设施的话，那么你将失去平台无关性，并有损失性能优势的潜在危险。

荣耀：鉴于Herb Sutter的双重身份：ISO C++标准委员会主席和微软软件架构师，C++/CLI对C++0x标准会产生什么样的影响？这种可能的影响是您希望看到的吗？

Bjarne：C++/CLI会在一些领域对C++产生影响，因为将会有许多人使用它。显而易见，在使用中人们将会建议加入一些新的设施，以便和ISO C++平滑地互操作。在确保ISO C++平台中立的前提下，这些建议会依据其优点而被评估。我并不认为Herb的双重身份会带来任何负面影响。请注意，ISO委员会的会议召集人的角色属于管理方面的。我认为微软将Herb的才能贡献到标准化进程中是一个积极的信号。一如往常，微软在实践着其“拥抱并扩展”策略，但至少他们拥抱的是ISO C++而非某种C++方言。

荣耀：C++0x标准大概可于哪一年颁布？目前标准化工作进展如何？我们在这个新标准中预期可以看到哪些新特性？

Bjarne：我希望三、四年内能够颁布，不过目前我们还没有一个明确的进度表。

我们打算在语言的扩展上持保守态度，并在与C++98的兼容性方面仔细斟酌。改进的关键可能会落在对泛型编程更好的支持以及对新手更易学习上。我们期望一个关于模板实参的类型系统“concepts”能成为泛型编程的基

石。

在标准库的扩展方面我们打算胆子更大一些。新标准库技术报告也许会使你对它的发展方向有一些认识。我们的关键目标是使标准库成为一个更广泛的系统编程平台！

荣耀：一个冒昧的问题。为什么C++标准委员会主席是Herb Sutter而不是您？我记得您是进化工作组主席，我们都很有兴趣知道您在目前标准化过程中具体从事什么工作。

Bjarne：我并不想担当会议召集人的职务。Herb和他的前任们在那个职位上比我所能做到的要出色得多。会议召集人主要负责管理性和组织性的事务。我的（非正式的）角色在于努力维持语言一致性的方向。我是语言进化工作组的主席，这个工作组负责处理所有语言扩展方面的提议。这个职位意味着绝大部分定义语言改变的文本都是我写出来的，而这些文本最终形成了标准文档。

作为进化工作组的主席，目前我正致力于三件事情：“concept”，改进的初始化设施，以及对C++新手的更好支持。你可以从我的主页上以下链接看到人们对标准下一个修订版（即“C++0x”）建议的期望特性列表：http://www.research.att.com/~bs/C++.html。简单的数一下你就会发现这些特性根本不能全部塞到标准中去，所以，我们需要一些优先级上的考虑。此外，我们还需要把注意力集中到一些相互关的问题上，而不是每次单独处理这些个体提议。如果把每个特性都单独考虑的话，你无论如何也不能得到一个内在一致且易教学的语言。正如语言本身一样，特性必须是为解决问题而设计的，而不仅仅是一个一个地“看上去很美”。

让我扼要介绍一下我眼下正以高优先级进行的三个语言扩展：concepts，初始化，以及“消除一些令人不愉快的瑕疵”：

在《C++语言的设计与演化》（D&E）中对模板的讨论部分，我花了整整三页来讨论模板实参的约束。很明显，我觉得它需要一个更好的解决方案。在使用模板（例如标准库算法）时哪怕出一丁点儿差错都可能招致极其“壮观”而无用的错误信息。问题在于，模板代码对其模板实参的“期望”是隐式的。让我们考虑一下find_if()：

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

这里，我们对类型In和Predicate作了若干假设。从代码中我们可以看出，In必须支持!=、*和++，并且这些操作符还必须具有恰当的语义。另外，我们必须能够拷贝In类型的对象作为实参和返回值。类似地还可以看出，我们可以“以*作用在In对象上所返回的值”作为实参来调用Pred，并且把“!”运用到该调用返回的结果上，从而得到一个可以在语义上看成是布尔值的东西。然而，这些约束在代码中都是隐式表达的。标准库为前向迭代器（此处为In）以及谓词（此处为Pred）小心翼翼地记录了各自要求的条件，但编译器可不会阅读文档！试试以下错误的代码，看看你的编译器会有什么反应：

```
find_if(1,5,3.14); // 错误！
```

我以前的想法提供了一个不完备、但相当高效的解决方案，即使用一个构造函数来检查对于模板实参的假定条件（见D&E 15.4.2），这个解决方案现在得到了广泛运用，被称为“concepts（概念）”或“constraints classes（约束类）”。你可以从我的主页上的技术FAQ中找到一些例子：http://www.research.att.com/~bs/bs_faq2.html#constraints。

然而，我们真正想要告诉编译器的是“我们期望模板实参是什么样子的”或者“我们期望模板实参满足哪些要求”，例如：

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

假设我们可以表达Forward_iterator和Predicate是什么，编译器就能够在不用查看find_if()定义的情况下检查对它的调用是否正确。这里我们所要做的就是为模板实参构建一个类型系统。在现代C++中，这种“类型的类型”被称为“concepts”。有多种方式可以用于表达concepts，眼下我们暂且把它们看成一些受到直接的语言支持并具有优雅语法的“约束类”。一个concept表明了一个类型必须提供哪些能力，但并不强制规定它们如何提供这些能力。理想的concept（例如<Forward_iterator In>）应该非常类似于数学抽象（对于任意的In，它必须可被递增（++）、解引用（*）以及拷贝），就像原来的形式“<class T>”从数学上来说是“针对所有的类型T”那样。

这样一来，在仅仅给出find_if()的声明的情况下，我们可以写：

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

这将会失败，因为int并不支持解引用（*）。换句话说，这个调用将不能通过编译，因为int并不是一个Forward_iterator。很重要的一点是，这将会使编译器更容易在该调用首次被看到的那一点上报告用户所犯的错误。

遗憾的是，仅仅知道iterator实参是Forward_iterator以及predicate实参是Predicate还不足以保证对find_if()的调用能够成功编译。这两个实参类型之间是有着交互作用的。说得详细一点就是，predicate所接受的实参是一个

被解引用的iterator （pred(*first)）。我们的目标在于对模板进行“和调用相分离”的完全检查以及无需查看模板定义就能对每次调用进行的完全检查，所以，concept必须具有足够强的表达力，以便处理这种模板实参之间的交互关系。方式之一是对concept本身也进行参数化，就像模板本身的参数化那样。例如：

```
template<Value_type T,
Forward_iterator<T> In,  // 对一个T序列进行迭代
Predicate<bool,T> Pred>  // 接受一个T并返回一个bool
In find_if(In first, In last, Pred pred);
```

在这儿，我们要求Forward_iterator必须指向一个T类型的元素，而该元素的类型必须和Predicate的实参类型一样（译注：实际上只要类型兼容即可）。

这方面的工作正在进行中。你可以从C++委员会的文件、学术文献以及有关C++0x的讨论中找到这方面更多的信息，在这里我没有太多的时间或地方进行更详细地阐述。“concept”的目标是提供模板使用和模板定义的完美的分离式检查，同时不引入任何运行期负担（译注：在C#的所谓的泛型中，concept只不过是间接函数调用的语法糖而已，运行期额外负担仍然存在。）以及不必要的concept耦合（译注：在C#所谓的泛型中，concept要求模板实参继承自一个公共基类，因此耦合仍然存在。）。

换句话说，我们想要把静态类型检查的好处引入到C++中的高度抽象的层面上去，同时不损及目前的模板技术所提供的灵活性和效率。

C++的基本思想之一是“对用户定义类型提供和内建类型一样良好的支持”（见D&E4.4）。但考虑下面这个例子：

```
double vd[ ] = { 1.2, 2.3, 3.4, 4.5, 5.6 };
vector<double> v(vd, vd+5);
```

我们可以直接使用“初始化列表”来直接初始化数组，而对于vector，最好的情况是我们可以先创建一个内建数组然后再用它来初始化vector。如果只有很少的几个初始化值，我可能会倾向于使用push_back()以避免将初始值的数目显式“写死”在代码中（上面例子中的初值是5个）：

```
vector<double> v;
v.push_back(1.2);
v.push_back(2.3);
v.push_back(3.4);
v.push_back(4.5);
v.push_back(5.6);
```

我想任何人都不会认为这两种解决方案有任何“优雅”可言。要想得到可维护性更好的代码并且让vector比内建（具有固有的危险性）数组更“讨人喜欢”的话，我们需要这样的能力：

```
vector<double> v = { 1.2, 2.3, 3.4, 4.5, 5.6 };
```

或者：

```
vector<double> v ({ 1.2, 2.3, 3.4, 4.5, 5.6 });
```

由于实参传递是依据初始化来定义的，因此这对接受vector为参数的函数同样奏效：

```
void f(const vector<double>& r);
// …
f({ 1.2, 2.3, 3.4, 4.5, 5.6 });
```

（译注：这里即是说，实参传递和初始化的语义是一样的，例如：

```
void f(T a);
f(x);
```

这里“把x作为实参传递给a”的过程等同于

```
T a = x;
```

这是个初始化表达式。）

我相信这种初始化器（initializers）的一般形式将会成为C++0x的一部分，这不过是将成为对构造函数进行全面检修的一部分，因为人们已经发现了有关构造函数的不少弱点，这些弱点看起来可以通过对构造函数进行一些修整来解决，例如“转发构造函数（forwarding constructor）”、“有保障的编译期构造函数（guaranteed compile-time constructors）”以及“继承的构造函数（inherited constructors）”等。

第三件事是“剔除语言里的一些令人不愉快的瑕疵”，也就是说，修整一些细小的不合常规或不方便的东西，它们对有经验的C++老手不会产生什么影响，然而却可能严重打击C++新手。一个非常简单的例子就是：

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

在C++98中，这里有一个语法错误，因为“>>”被看成一个单独的词汇标记，而不是两个“>”。正确的写法如下：

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

对于C++98这样一个“不近人情”的规则，虽然有足够技术上的理由，但这不应该强加给任何背景的新手（包括其

它语言的专家）。如果编译器不接受前一种最为明显的v的声明形式的话，那么C++用户和教师都会在这上面浪费大量的时间。我希望这个“>>问题”以及其它一些瑕疵都会在C++0x中消失。事实上，在和Francis Glassborow以及其他一些人的工作中，我一直努力去系统地消除出现频率最高的此类语言瑕疵。

荣耀：和大多数人一样，我认为C++缺乏一个大一统的库是阻碍C++更为广泛地使用的关键原因，您认为现在C++社群有足够的资源来开发一个像Java或.NET那般规模的库了吗？如果没有，我们该怎么做？我发现使用形形色色的第三方库非常不方便（一个插曲。我在使用微软Visual C++时，有时希望使用STL组件，例如vector，但由于我大幅使用了MFC，而MFC中也有类似的容器，所以，虽然vector更好用，但为了避免因链接两个不同的库而导致文件体积增大，我最终往往放弃使用标准库。不过，倘若标准库提供了MFC所提供的所有功能，我将肯定全部改用标准库）。

Bjarne：毫无疑问，MFC是迄今为止被广泛运用的最糟糕的基础库。它违反了一个好的C++设计应该遵循的大多数原则。它严重地扭曲了许多程序员对于“什么是C++”的看法！

当然，我也认为缺乏全面且标准的基础库是C++社群的一个主要问题。对于个体程序员来说，这也许是他们面对的最困难的问题。我在《C++语言的设计与演化》中谈到了这一点，至今我仍然坚持这一点。

C++社群没有一个有钱的公司来支持“平台中立的”标准库的开发 — 从来没有，跟其它专有语言（以及它们的“标准”库）相比，这一直是阻碍C++发展的一道藩篱。和专有的基础库的扩展速度相比，我们对标准库的扩展是很慢的（不过和其他ISO标准库相比，我们的扩展速度应该算是比较快的）。我们还期望能够和新兴的非标准库（译注：如Boost）逐渐达成更为平滑的整合。别忘了，当年MFC以及其它“后80年代”风格的库被设计出来投放市场之际，尚无任何标准库可作它们的构建基础。因此，我希望今后的专有库和开源库都能充分尊重标准库，以便使它们之间的互操作变得容易 — 至少容易一点点。

C++社群并不是为大规模设计和实现而组织的。C++社群中也没有什么传统或惯例。和其它社群相比，我们缺少一个“统帅”，他可以有效地激励新库的创建，否决或“保佑”我们的努力和成果。然而，我对“统帅模式”是否可行心里没底 — 除非你所做的事情只是基本的模仿而已。Boost（www.boost.org）是一个优秀的成果，它的某些方面已经超出了模仿的范畴，然而它仍然缺乏一个明确的目标以及用于维持自身发展的权威机构或权威人物。

在以下三个相关的领域中，大规模的合作努力对于C++社群而言是必需且有意义的：基本的并发编程库（包括线程、锁和lock-free算法等）；平台无关的操作系统服务库（目录和文件操纵以及套接字等）；以及GUI编程库。前两个看起来是可行的，但要想建立一个标准的GUI库，也许从技术上、经济上尤其是政治上都显得太困难了。

荣耀：是的，我相信对于许多C++程序员而言，一个标准的GUI库是极其重要的。许多人以为GUI库不是一个太复杂的问题，但我却认为这可能是最棘手的问题。为了解决它，我们可能需要难以置信的大量的资源。同时我认为政治问题可能是其他问题的根源。举个例子，我们知道Windows的GUI风格和Java的GUI风格是不一样的，甚至不同版本的Windows的GUI风格也不一样。而且我们知道GUI风格很不稳定，它发展演化得非常快，而且很容易被微软这样的大公司所引导和操纵。

Bjarne：在这个问题上，资源（包括财力和人力）是一个关键问题，与“保持设计蓝图的一致性”同样重要。另外，即使我们已经有了这三样东西：蓝图、人力（数打甚至更多）、以及财力（至少要担负得起一些优秀的人在这个项目上的全职工作），这仍然会是个需要N年才能完成的项目。而SUN、微软、苹果以及其他主要竞争对手将会做些什么呢？毕竟包含了一个工业强度的GUI库的ISO标准C++对于它们的专有系统可不啻一记重击啊！我的猜测是它们大多会通过强大的市场手段来保护既得利益，标准委员会可不是为了在这种市场环境中呼风唤雨而组织起来的，我们至少还需要一个工业社团的支持以及和开源社群主力军的联盟。

荣耀：在您看来，C++要想继续向前发展，除了开发一个更为广泛、更具威力的库以外，我们还应该做些什么？

Bjarne：我认为库方面的工作是关键。标准委员会没有足够的资源去创建一个全新的库，而且无论如何“让委员会设计”都不是个好主意。如果人们创建了新的库，在紧密相关的领域把它们的努力融合起来以达到冲突最小，并且文献记录良好（并非只记录细节，还有设计原则），那么他们也许可以把这些东西带到委员会来，并有希望让它们成为标准。如果没有成为标准的话，至少他们的努力有最大的机会成为“准”标准。有此打算的人应该把C++真正当作“C++”（使用继承、模板以及异常等）来看待，而不是从其它特性不是那么丰富的语言中抄袭设计而来。如果一个库的设计被发现没有用C++最佳地表达出来，那么它被接受为标准的几率微乎其微。

另一个可作贡献的领域是对技术的开发和推广。C++往往被以“很不理想”的方式使用着，MFC就是一个典型的例子，它甚至还达不到80年代中期对一个良好的OO设计的看法！我所说的“不理想”，是指没有达到它所能达到的可维护性的设计（通常这是由于对设计决策的糟糕的分解、低劣的封装以及对概念的拙劣表达而造成的），而并非指在外部压力下要尽快把项目赶出来的个体程序员或团队的“不理想”。通常，一个较好的设计和现有的实践是背道而驰的，后者往往需要立即付出代价（往往在很短时间内）。显然，任何显著的改进都需要在“一切照常”的基础上，并且至少追加学习所需的代价以及时间上的延迟。

这就把我们带到了最重要并且也许是最明显的改进实践的途径面前：教育。我们对编程和设计的教育必须比当前做得更好。新的语言特性、新技术以及新库碰到了不能理解并使用它们的人仍然是无用之刃。遗憾的是，我们恰恰缺乏优秀的C++入门书籍。Francis Glassborow的新书《You Can Do It》（译注：中文版《C++编程你也行》即将由人民邮电出版社出版）和Koenig & Moo的《Accelerated C++》是打破旧式而令人厌烦的教育方式的例

子。那种教育方式把C++当作一个“稍微好一点的C”或者一门在实现“真正的面向对象”方面失败的语言。前者倾向于用一大堆语言技术细节来迷糊和恼怒读者，偏离了学习好的编程及设计技术的正确道路。后者通常除了存在这些问题之外，还没能够教会在性能攸关的领域里编程的必要技术，而且没有让人感受到静态类型系统的价值。尤其遗憾的是，Glassborow和Koenig & Moo的书的风格都不是编程入门教材的传统风格，这也阻碍了它们的广泛普及。

我自己的两本书《C++程序设计语言》和《C++语言的设计与演化》的目标读者则是已经知道如何去编程然而不知道如何使用C++的人。这两本书确实满足了这部分读者，但我们要做的还不止这些。

任何时候只要有可能，程序员都可以通过将程序的主要部分用ISO C++来写，并将系统依赖性封装起来，从而来支持标准。也就是说，把系统依赖性限定到特定的区域，并通过以标准C++表达的接口去访问它们。通常这并不容易做到，因为厂商总是怂恿程序员在代码中使用专有的、排他性的特性，但这就影响了可移植性，从而使程序移植到其他系统上非常困难。然而，站在应用构建者的立场来说，从长远来看，挣脱厂商的束缚、维持可移植性是一件好事，这种隔离系统依赖性的努力终将从经济和技术两方面都得到回报。

荣耀：根据您掌握的资料，模板和泛型编程在业界被广泛采用了吗？还是主要局限于库作者？您认为对于普通程序员（而非库作者）来说，面向对象和泛型编程哪一个更重要？为什么？您认为今后模板和泛型编程应该比今天得到更普遍的应用吗？

Bjarne：“加法和乘法哪个更重要？”大多数情况下这是个愚蠢的问题。同理类推，“面向对象编程和泛型编程哪个更重要？”的问题也毫无意义。关键在于，它们都是基础性的东西，并且是互补的。对于许多问题而言，最佳解决方案往往要求将它们结合运用。

从理论的角度来说，你是在“ad-hoc 多态”和“参数化多态”之间做出选择。而从实现角度来说，则是在“运行期多态”和“编译期多态”之间进行选择。你必须对两者都有所了解，并且恰当地使用它们。和“ad-hoc多态”（在C++中以类继承来表达）相比，“参数化多态”（在C++中以模板来表达）更具规则性，更利于逻辑或抽象思考。这就是为什么它被称为“ad-hoc”、为什么模板代码通常具有更好的性能、以及为什么当个体表达式和语句的性能很重要时我们应该考虑泛型编程的原因之所在。相反，类继承则能够在分离式编译和维护的代码块之间提供更为明晰的接口。

（译注：“ad-hoc”是“专门”的意思。事实上，多态一般分为“ad-hoc多态”和“universal多态”，前者一般指重载，后者一般指“参数化多态”（模板）或“包含多态”（类继承）。你可以从以下链接找到详细的解释：http://www.javaworld.com/javaworld/jw-04-2001/jw-0413-polymorph.html。）

我不能说哪个将会变得更重要，或者哪个应该更重要。那就好像是在加法和乘法之间偏袒某一方一样。但是我怀疑，和没有充分使用类继承和面向对象编程的人相比，有更多的人没有充分使用模板和泛型编程。因此，我更多的时候鼓励人们考虑并使用泛型编程而不是面向对象编程。我确信我们将会看到越来越多的泛型编程应用。泛型编程目前仍然没有被足够的理解和充分的使用。我们接受了近十年的面向对象的耳濡目染，所以我的感觉是面向对象常常被滥用了。值得注意的是，无论如何，泛型编程和面向对象编程这两种技术/风格都比使用一团乱麻似的选择语句、单薄的数据结构以及指针来解决问题要强得多。我们的目标应该是在代码中更为直接和优雅地表达思想，模板和类继承只是为了达到这个目的的工具。

我的感觉是许多“普通的程序员”确实使用了模板，而不仅仅是一些“库设计”方面的精英。当然，我们自己写的模板代码可能要比基础库里的代码简单一些，这很自然，因为对于其它编程技术来说，也存在同样的现象。

荣耀：我想知道您对模板元编程的看法，您甚至选编了一本模板元编程的书。

Bjarne：我想我并不愿意把泛型编程和模板元编程区别开来，它们的区别仅仅是在层次和侧重点上。我通常倾向于把这一块统称为泛型编程。事实上我认为模板元编程是一个非常重要的领域，而目前的C++对它的支持却不佳，以至于在生成的代码中它并不能发挥应有的潜力。我认为人们在这一领域所作的许多努力是实验性的，其中许多理应获得成功，因为比起替代方案来，它们能使代码更清晰地表达基础概念，并且具有更好的性能以及可维护性。然而C++98对这些技术的支持却不是很好，所以它们目前还不能成为主流。鉴于此，我在语言的改革方面所作的许多工作都跟泛型编程直接或间接有关。concept（用于分离式检查模板的使用和定义，以及用于更好地重载模板等），更好的初始化，以及更少的不规则性，都是对此有所帮助的努力，同样，用于支持标准库的设施（例如type traits）也会带来帮助。

荣耀：我个人认为标准C++流库是面向对象和泛型编程结合运用的典范，您赞成这一点吗？对于准备尝试混合使用面向对象和泛型编程技术的程序员，您有什么建议或忠告？

Bjarne：不，我认为流输入输出流是一个不错的早期尝试。然而，它仅仅是一个非常初步的尝试而已，随着时间的推移而显得过于精致而复杂（正如发生在大多数成功的系统中的那样）。我们现在可以做得更好。在我设计第一个流库时我意识到泛型编程的必要性，但是当时我并没有料到泛型编程最终会变成C++中如此重要的一个组成部分。

荣耀：我必须得承认我也许太闭目塞听了，我真的没有看到过比标准C++输入输出流更好的面向对象编程和泛型编程的结合应用范例。您能给我一些线索吗？更重要的是，您能告诉我们有哪些是结合运用面向对象编程和泛型编程的最佳场合？谢谢！

Bjarne: 如果你面对的问题既需要某些运行期决议（需要面向对象编程），又具有一些能够从编译期决议中获益的方面（泛型编程的用武之地）的话，那么你就需要将面向对象编程和泛型编程结合起来。例如，面向对象编程的经典例子 — 将一个保存了shape的容器中的所有元素都显示出来就属于这类问题。几十年前我第一次在Simula中看到过这个例子，后来直到遇到了泛型编程，我才看到它的改进实现。考虑以下代码：

```
void draw_all(vector<Shape*>& vs)
{
   for (int i=0; i < vs.size(); ++i) vs[i]->draw();
}
```

我猜想这并不能被看作纯粹的面向对象编程，因为我直接利用了“vs是一个装有Shape*元素的vector”这个事实。毕竟，类型的参数化通常是被认为属于泛型编程的范畴。我们也可以消除这种对静态类型信息的使用（所谓“不纯粹的面向对象编程”）：

```
void draw_all(Object* container)
{
   Vector* v = dynamic_cast<Vector*>(container);
   for (int i=0; i    {
      Shape* ps = dynamic_cast<Shape*>(v[i]);
      ps->draw();
   }
}
```

但凡鼓励以上这种风格的语言，其语法通常都比较漂亮，然而这个例子却说明了当你把静态类型信息的使用减至最小的时候发生了什么。如今，在C++或其它语言中，仍然有人在使用这种风格。我只是希望他们在错误处理方面有系统化的准备。

在前一个例子中，vector<Shap*>依赖于对泛型编程的一个最简单的运用：vector的元素类型被参数化了，而且我们的示例代码正获益于此。在这个方向上我们还可以走得更远，即推而广之到所有标准库容器身上：

```
template<class Container> void draw_all(Container& cs)
{
   for (typename C::iterator p=cs.begin(); p!=cs.end(); ++p)
      (*p)->draw();
}
```

例如，这段代码就既可以作用于vector上，又可以作用于list上。编译期决议确保我们不用为这种泛化处理付出任何运行期额外代价。我们还可以通过在draw_all()的使用接口中运用迭代器，从而进行进一步的泛化处理：

```
template<class Iter> void draw_all(Iter fist, Iter last)
{
   for (; first!=last; ++first)
      (*first)->draw();
}
```

这就使内建数组类型都得到了支持：

```
Shape* a[max];
// 向a中填充Shape*类型的元素
draw_all(a,a+max);
```

我们还可以结合运用标准库算法for_each()和函数适配器mem_fun()来消除显式的循环：

```
template<class Iter> void draw_all(Iter fist, Iter last)
{
   for_each(first, last, mem_fun(&Shape::draw);
}
```

在这些例子中，我们结合了面向对象（对虚函数draw()的调用以及对类继承体系的假设）和泛型编程（参数化的容器和算法）技术。我看不出如果这两种编程风格（即所谓的“范型”）各自独立运用如何达到同样好的效果。这也是一个简单的“多范型编程”的例子。

我认为在设计和编程技术方面，我们还需要做更多的工作，以便确定出“关于何时采用哪种范型以及如何结合运用它们”的更为具体的规则。为此，我们还需要一个比“多范型编程”更好的名字。

注意，这也是一个关于编译错误信息变得可怕的例子，因为我们并没有显式地表达出我们的假设。例如，我们假设容器里的元素类型为Shape*，然而在代码中，这个假设却相当隐晦。这种情况下我们可以使用约束类（此处为Point_to）：

```
template<class Iter> void draw_all(Iter fist, Iter last)
```

```
{
   Points_to<Iter,Shape*>();
   for_each(first, last, mem_fun(&Shape::draw);
}
```

然而我们又确实很想说明“first和last必须为前向迭代器”：

```
template<Forward_iterator<Shape*> Iter>
void draw_all(Iter fist, Iter last)
{
   for_each(first, last, mem_fun(&Shape::draw);
}
```

这是“concepts”可以大展拳脚的地方之一。

荣耀：请原谅我重复一个老俗套问题。由于Java和C#今天都已经大获成功，您对Java和C#曾经的看法今天有无改变？我个人认为，与其说Java和C#的成功是语言自身的成功，还不如说是SUN的Java战略和微软的.NET战略的成功。

Bjarne：在对它们的本质技术优点以及对它们的市场能量的估计上面，我都是正确的。低估市场的影响是不明智的，尤其当它背后有价值上百万美元的“免费”库所支持的时候。Java和C#是不坏的语言，而且SUN及其盟友以及微软及其盟友为其（过分夸张）的宣称提供重大的库和工具的支持。不过，这么说并不意味着我比喜欢C++更喜欢它们，对于要求严苛的应用而言更是如此。

荣耀：您自觉或自发地使用过GOF描述的设计模式了吗？您对设计模式怎么看？您对Loki库中采用模板技术描述的静态设计模式怎么看？

Bjarne：我并不喜欢根据特定的具名模式去思考，但我知道并且通常会使用这些在《设计模式》中描述的技术。顺便一提，《设计模式》是一本经典书籍，人们在搜寻最近最好的信息时不应该忘了这本书。其中的许多模式对于好的设计来说非常重要 — 给它们起什么名字倒无所谓。甚至你在TC++PL中也能够找到一些有关它们的运用。要想了解在某个特定领域中对模式有意识且系统的运用，可以参考“深入C++系列”中Schmit和Hunston的两本关于ACE的书。

除了明显的强大能力之外，我认为模式有两大弱点：

- 它倾向于鼓励“精致的专用术语”，这会阻碍新手的学习。
- 如果没有具体的“工具”支持，要想把一个思想广泛地传播到应用中是极其困难的。

例如，一个优秀的库本身携带了很多优秀的思想，并允许程序员（和设计者）直接利用这些思想在库中的实现品来工作。而模式只是对某个思想（或一系列互相关联的思想）的尽量一般性的描述，并刻意避免将这些思想作为库实现出来而招致的特殊性。这就导致了这些思想在传播上的问题，以及从代码中如何识别出模式的问题 — 特别是在代码被维护修改过之后。同样，要想从抽象层面上来理解一个模式也是非常困难的。在某个模式的抽象描述之后的实例代码进入我的视野之前，我倾向于对自己的理解持保留态度。我见到过一些人，他们认为自己是在使用某个模式，而实际上做的却是该模式被设计用来避免的事情。这些都说明思想的传授可能会异常困难。

可以让模式更具有可利用性的方式之一是为某些特定的环境提供模式的库的实现。Andrei Alexandrescu的书和他的Loki库可以被看成一次寻求结合高灵活性和高效率（和手写代码一样高效）编程风格的尝试。而模板元编程在大多数情况下都符合这个描述，STL亦然。为了从这种非常一般性的参数化中获益，设计（或编码）抉择必须从运行期转移到编译期，从而程序才更容易在时间或速度上得到优化。遗憾的是，许多编译器都不能很好地把握模板技术所提供的明显的优化机会，这通常是由于编译器过早地扔掉了类型信息，并试图去优化每一片代码，就好像它们是用弱类型风格的C所编写的一样。

荣耀：您用过UML吗？您对UML怎么看？您认为它对C++程序设计很有用吗？

Bjarne：我尝试过UML，但并不是为了一些严肃的事情，所以我的看法参考价值不大。对于我最常考虑的设计问题来说，UML不是特别有意义。我发现在设计和记录设计时草图是不可或缺的，但我并不认为把过多的细节加到草图中是个好主意。相比之下，代码更易于表达精确的关系。当然了，这么说并不意味着UML没有用，我所尊敬的一些人认为UML在文档化大型系统时非常重要。

荣耀：您目前还在写书吗？或者有C++新书写作计划吗？

Bjarne：我目前正基于我正在讲授的一门新手课程撰写一本面向初学者的编程书。这对于那些具有很少甚至没有编程背景但很想通过努力成为程序员的人应该有所帮助。此前我从未试图为非程序员写书，因为我对完全没有编程经验的人们所知甚少。现在我正教授初学者，这就让我有机会去尝试我的想法，并基于我的教学经验对之不断修改。对于任何瞄准于初学者的东西来说，这样做都是必不可少的。

我的目标是先教给新手最小的一套原则、技术以及语言设施，让他们可以先开始第一个实在的项目。基本上，我打算让那些想要成为职业C++程序员的人由此起步。为了达到这个目的，我一开始的讲授要涵盖很多背景知识，包括数据结构、算法、图以及类设计等，这在传统的教学中是不会很早涉及的。这比我了解到的当前大多数的教

学方式更加雄心勃勃。当然，我使用C++作为编程语言，并且我会在讲授中涵盖STL的基础知识。

荣耀：您能透露一下您的新书何时出版吗？

Bjrane：我还没有写完呢，而且我还没有跟出版商（Addison-Wesley）协商好进度计划。至少我还需要半年时间来和我的学生们一起精化和锤炼这本书。理论上，我们（Lawrence Petersen是我的合作作者）在明年秋天大概可以完成，如果我们在重审的过程中发现了重大问题的话，那就得等到圣诞节之后才能付梓了。我知道出版商会督促我们早点完成，而我们则会争取更多的时间来检验这本书并采纳反馈。对于这样的一本书，我们是不可以草率对待的。

荣耀：您对您负责编辑的“深入C++系列”（Addison-Wesley）有何评论？您是否认为其中一些书已经过时了？一些书仅对有限的读者群有作用？您对新近出版的几本书有何评价？还有哪些新成员即将加入这套丛书？我特别想听一听您对《Modern C++ Design》和《C++ Template Metaprogramming》这两本书的看法。

Bjarne：过时？一些早期的书的确有点过时，但是从总体上来说，这些书都很好地经受住了时间的考验。新的书籍会以稳定的速度加进来。我的猜想是每年会增加三本新书。当然，我希望有更多，但是要想找到质量和实用性足够好的“轻薄洗练”的书并非易事。事实上我很期望能够看到更多精专的书，比如着眼于数值计算的特定方面的书，以及嵌入式系统编程方面的书。人们应该注意的是，这些书中的许多都是以专家或至少有经验的程序员为目标读者的。并且，作为一个专家，应该知道何时使用何种技术。我认为泛型编程和模板元编程是将来的一部分，但那只能算是基础，而STL这样的东西才是当前的主流。你所读到的一些书，像Alexandrescu和Abrahams的那两本书，会为你带来新的概念和可供试验的思想，但是你并不会在正准备部署的系统中立即应用这每一项技术。

荣耀：您对Boost似乎情有独钟。“深入C++系列”中已经包含了《Boost Graph Library》和《C++ Template Metaprogramming》两本书，我猜将来还会有更多有关Boost的书会加入到这个系列中来，您对Boost怎么看？选编这方面的图书出于什么考虑？

Bjarne：事实上，我并非根据它们是否是关于Boost的来选择书籍，“深入C++系列”中的大部分书籍都不是关于Boost的。我根据它们是否提供了有关编程技术、原则和概念的有用信息做出选择。恰巧Boost的作者在尝试“根据标准库的精神”来扩展基础库时，使用了有趣的技术去解决有趣的问题而已。

荣耀：在过去的一年里，只出了有限的几本C++书籍。除了“深入C++系列”中的三本新书外，Addison-Wesley还出版了一本《Imperfect C++》，您对这本书怎么看？（译注：因本书由我们翻译，故有此一问。）另外， 您认为C++0x标准会催生更多的C++新书出版吗？

Bjarne：人们好像没有以前那样喜欢读书了。今年的C++新书只有寥寥几本，不过已出版的C++书籍已经有很多。“老”书并不一定过时。我认为我自己的书就是很好的例子。《C++程序设计语言》卖得比大多数新近出版的书要好，而《C++语言的设计与演化》则刚刚被翻译成日语。类似地，K&R（译注：《C程序设计语言》）在25年后的今天仍然是最畅销的书！请不要忘了经典。我觉得有一个现象蛮有趣的，我附近的技术书店的Java部分的书几年来第一次比C++部分小了。

我看过《Imperfect C++》的草稿。如果它的页数可以降到原来的一半的话，我非常愿意把它加入到“深入C++系列”中去。该书作者是一个C++热爱者，他希望向开发者展示如何对付C++中诸多不完美之处，以便写出更好的代码，而这些代码在声称为“理想”的语言中更难实现。

我期望C++0x会催生一批新一代的C++书籍。不过，我希望这些新书能够集中于语言对编程风格和设计技术更强固的支持上，而不是简单地列举语言特性。我们已经看到了一大堆令人厌烦的列举语言规则的书。如果孤立开来看，任何语言特性都是无趣的，真正有趣的主题是编程。

荣耀：由于您正在给C++新手讲解编程课，您愿意给世界上其他C++教师谈些教学经验吗？

Bjarne：多年来，我一直对普遍的编程教育尤其是C++教学质量感到不愉快。当然，有很多好老师，而且也确实有很多学生变成了很好的程序员，然而，严重误导性的教学和被严重搞迷惑的程序员似乎没有个尽头。大概一年半前，有人建议我为编程新手设计一个全新的编程课程。我有过犹豫，但最终还是答应了，我和我们最具经验的讲师Lawrence Petersen合作，他会弥补我在教授新手方面的经验的不足。我们设计了课程并已讲授了两学期，并且基于我们的讲授方式、客观效果以及反馈（反馈是不可或缺的），不断加以改进。

当然，我并非仅凭个人的主观看法和所知的实验性的途径就一意孤行。事实上，我阅读了大量的口碑好的C++编程入门教材，看看它们到底好在何处。大约有两个星期，我烦躁地走来走去，抱怨着“要是那些玩意就是C++的话连我自己都不会喜欢它！”。我的感觉是许多教材彻底讹传了C++，可怜的学生们！所以，我们首先基于“一个学生成为职业程序员所需要的”知识列出了一个课程大纲，写了讲稿，列出了练习并附以大量的告诫。我们快速的重审材料并根据哪些可行、哪些不可行多次对其进行了调整。是年秋天，我们再次重审了材料，并且将课程的文字印刷成书。这下好多了，但是我们根据学生的反馈再一次调整了许多细节。我们的确越做越好，学生们的反响（包括口头反映和考试成绩）说明了这一点。今春，我们将会再一次讲授这门课，到了夏末时大概就可以进行更大范围的普及推广了。

我们把这种讲授方式称为“深度优先”，因为在课程一开始，我们会给学生介绍许多材料但并不深入细节。我们从第二周开始简单地使用STL，到了第五周学生就知道错误处理策略和类设计。在那之后，我们拓宽他们的知识

面，并且让他们把知识应用到一些领域去，譬如文本处理、文件操纵以及绘图等。我认为这是一个充满雄心的教学方式，不过在好的（不一定是极好的）大学里好的学生身上效果不错。

荣耀：虽然这个问题应该去问Alex Stepanov本人，不过由于最近您们二位结伴前来中国杭州参加一个嵌入式软件系统会议，我顺便想打听一下，Alex目前是否在为C++新标准库忙些什么？

Bjarne：实际上你真的应该去问Alex本人。他的见解总是很有意思，而且往往出人意料。是的，他也参加了那个大会。在杭州，我们分别做了主题报告并给大学生们做了演讲。让很多人感到惊讶的是，他的报告是着眼于软件业的金融基础的，而绝大多数人所期望的是他能做一个关于STL的高性能应用的演讲。我讲的是有关将抽象映射到机器层结构上的东西，这是C++在嵌入式系统编程中高效应用的基础。我很多主要的想法都可以在C++标准委员会关于性能的技术报告中找到（见我的C++主页上的链接http://www.research.att.com/~bs/C++.html）。

荣耀：尽管在一些领域C++受到其它语言的挤压，但我相信未来10年内C++仍然是最重要的系统开发语言，您是否赞同这个观点？

Bjarne：我并不认为C++被“挤压”了多少。IDC的评估数据表明，今天的C++程序员数量大约是十年前的三倍，而且C++仍然比任何其它语言更多地被使用。我认为，更确切的描述应当是这样的：C++在过去的十年里，只吸收了软件开发巨大膨胀的一部分。C++也正在向一些新领域扩张，譬如硬实时的程序设计。我怀疑“最重要的系统开发语言”这一说法很难被量化评估，但勿庸置疑，C++仍将占据非常重要的地位。关于C++应用的多样性，可以看看我列出的应用程序清单：http://www.research.att.com/~bs/applications.html。

荣耀：还有没有我没有问及而您又希望补充的内容？

Bjarne：你没有问我容易回答的问题。谢谢。

谢谢您，Bjarne Stroustrup！

C++0x 热点问题访谈

来源：荣耀 作者：Bjarne Stroustrup 等级：精品

发布于2006-10-22 12:54 被读1015次 【字体：大 中 小】

C++0x热点问题访谈

2004年底前后，经过较长一段时间的沉默，一批世界级的C++著作相继面世。2005年4月，在挪威Lillehammer举行的C++标准委员会会议上，Bjarne Stroustrup促成委员会达成一致意见：让C++0x中的x等于9。2005年11月，Bjarne Stroustrup、Herb Sutter、Stanley B. Lippman、Andrei Alexandrescu等前辈、新锐将在Las Vegas庆祝C++ 廿周年。2005年底，C++中国社群将在上海举办首届“现代C++设计和编程”技术大会……C++好戏连台，令人振奋。笔者近日就C++0x以及其他一些热点问题请教了Bjarne先生。大师观点，不敢专美，整理成文，以飨同好。

C++0x

荣耀：Library TR1（library Technical Reports，库技术报告）和TR2的动机是什么？TR1和TR2、TR1/TR2和C++0x、Performance TR（Performance Technical Reports，性能技术报告）和C++0x、Boost和C++0x之间的关系如何？

Bjarne：Library TR是对标准库改进工作的具体结果的体现。当一套设施、特性就绪后，TR1即被表决通过，然后人们继续向TR2前进。TR1和TR2之间的区别仅在于“时间不同，所做的事情不同”而已。大部分TR1和TR2中的内容有望成为C++0x的一部分。

Performance TR是一个关于C++适合于性能严苛和资源受限的编程的报告，大多和嵌入式系统编程有关。其作用主要在于教育程序员并努力驱散萦绕C++的能与不能的流言蜚语。特别要指出的是，该TR证明C++是一门极好的适合于嵌入式系统编程的语言。

启动Boost项目的人们（著名的如Beeman Dawes）过去是、现在仍然是C++标准委员会的成员。Boost的目标是提供对标准库的扩展，并使得大多数有用且成功的Boost库成为新一代C++标准。其中一部分（而非全部）将会成为C++0x的一部分。注意，当一个库被添加进标准时，往往需要对其进行某种程度的修改。许多TR1扩展一开始都是作为Boost的一个组成部分而开始它们的生命旅程的。

荣耀：我们经常听到C++在不损及效率的前提下达成抽象和优雅，在C++0x的演化（设计）中同样如此，您能从技术的层面来谈谈C++(0x)是如何做到这一点的吗？

Bjarne：这个问题说来话长，需要专门写一篇论文。不过基本的思想一如既往：遵从零开销原则（参见D&E），拥有内建的直接到硬件的操作映射，以及提供强大灵活的静态类型系统。

> 荣耀注：D&E是Bjarne写的一本关于C++语言设计原理、设计决策和设计哲学的专著。该书全名是《The Design and Evolution of C++》。零开销原则的含义是：无需为未使用的东西付出代价。详情参见D&E。

荣耀：您最近提到在C++0x标准化过程中，委员会奉行的若干原则中有这样一条：只做可以改变人们思考方式

的改变，请问此原则的出发点是什么？

荣耀注：D&E中并没有明确地提及这条设计原则，但Bjarne在最近的一篇文章里特别提到了这一点，故有此一问。

Bjarne：任何改变都会带来在实现、学习等方面的代价。对一个程序员编写具体哪一行代码的方式的改进不会带来太大的好处，能够改进程序员解决问题和组织程序的方式才可以。面向对象和泛型编程已经改变了很多人的思考方式，这也是很多C++语言设施支持这些风格的用意。因此，对于语言和库设计者来说，最好将时间花在有助于改变人们思考方式的设施和技术之上。

荣耀：如果让您分别列出可能会进入C++0x标准的Top 5语言新特性和库扩展，您愿意列出哪些？

Bjarne：语言特性包括：

- Concepts
- 一般化的初始化（generalized initialization）
- 对泛型编程的更好的支持（auto、decltype等）
- 内存模型
- ……

荣耀注：Concepts在泛型编程中具有极其重要的地位，但C++98并未对concepts提供直接的支持，它们只是以文档的形式连同一套松散的编程约定而存在。目前STL中的concept的作用仅限于在模板实例化时对模板参数进行检查，然而，如果某个函数模板参数不符合某个concept，编译器并不能对该函数模板的定义进行及早的强类型纠错。尽管Bjarne先生一向对语言扩展持保守态度，但在为concept提供语言层面的直接支持方面却一向积极，实际上，他本人正是为C++加入concept的最早的提案人之一。几乎可以肯定该语言特性会被加入到C++0x之中。

荣耀注：关于“一般化的初始化”问题可参见后面的问答。内存模型也就是后面所说的机器模型。auto和decltype则分别是计划加入C++0x的新关键字和操作符，它们均可用于改善泛型编程。

库设施则有：

- unordered_maps（即哈希表）
- 正则表达式匹配
- 线程
- ……

我不愿意勉强向这种“top N”列表中添加更多的东西，关于还有哪些主要的东西应该被添加进来，目前尚存在激烈的讨论。

荣耀：您说过，“在C++0x的concept的设计中存在的最大的困难是维持模板的灵活性。我们不要求模板参数适合于类层次结构或要求所有操作都能够通过虚函数进行访问（就象Java和C#的“generics”所做的那样）”。可以详谈一下吗？

Bjarne：在“generics”中，参数必须是派生于在generic定义中所指定的接口（C++中的等价物则是“抽象类”）。这就意味着所有generic参数类型必须适合于某个类层次结构。举个例子，如果你编写了一个generic，我定义一个class，人们无法使用我的class作为你的generic的参数，除非我知道你指定的接口，并使该class派生于它。这种做法太呆板。当然了，对此问题有迂回解决办法，但需要编写复杂的代码。

另一个问题是，因为内建的类型（比如int）不是类，它们没有generics所使用的接口所要求的函数，因此你不得不制作一个包装器类来容纳内建的类型。

还有，一个generic的典型操作是以虚函数调用的方式实现的。与仅使用一个简单的内建操作（例如+或<）相比，这种做法可能代价高昂。Generics的实现方式表明它们不过是抽象类的语法糖而已。

荣耀注：请注意，这并不是对Java/C# Generics的“全面”的描述，对C# 2.0 generics而言尤其不适合。

荣耀：您说过，C++0x极有可能通过“一个机器模型外加支持线程的标准库设施”来支持并发访问，请解释一下您所指的机器模型是什么？

Bjarne：当程序只有一个执行线程并且表达式都是按照程序员编写的顺序正确地执行时，处理内存是件很简单的事情：我们只管读写变量，无需想得太多。我们希望在今天更复杂的机器（具有高速缓存、宽数据总线以及多处理器）上仍然可以如此。一个内存模型是一套规则：它告诉编译器作者必须确保哪些东西在所有机器上都得到保证。

考虑

```
char x;
char y;
```

如果某个多线程程序的一个线程在递增x，另一个线程在递增y，我们当然希望两个字符变量均被正确地递增。然而，在很多现代机器上，我们无法在寄存器和高速缓存之间或高速缓存和内存之间正好传递8个比特。因此，如果这两个字符都分配在同一个字（word）上，我们就无法做到在读写x时不去读写y。

可能还存在更多的高速缓存（每一个都持有一份x和一份y的拷贝）。基本上，一个优秀的内存模型可以使程序员

无需关心此类机器问题——这些问题留给编译器作者去操心，为程序员提供的总是显而易见的语言语义。

荣耀：您说过，C++0x的高级目标之一是使C++更易于教和学。通过“增强的一致性、更有力的保证”来做到这一点，您能否给出这方面的一些例子？

Bjarne：比如说，目前的初始化规则太复杂且不规则。考虑一下在哪儿可以使用“={initializers}”、哪儿可以使用“(initializers)”、哪儿可以使用“=initializer”吧。我希望通过允许当前不被允许的用法，使得程序员只需学习和记住较少的规则，从而达到简化这个领域的目的。这方面的工作集中于实现“{initializers}”语法。

荣耀注：举个例子。在现有的C++中，我们这样来“初始化”一个vector：

```
vector<int> v;
v.push_back(1);
v.push_back(2);
v.push_back(3);
```

采用“{initializers}”语法，我们就可以消除对push_back()的重复调用。如下：

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

一个“更有力的保证”的例子，如前所述，精确地定义一个内存模型从而使得并发程序更具有可移植性。

我还希望看到针对所有标准容器的越界检查的标准方式。这种有益的保证的例子之一是：“如果你对任何标准容器的任何访问是越界的，那么就会抛出一个异常”。

……

总而言之，关键的思想在于使得广泛的常用语句变得更易于学习和使用。

荣耀：顺带一问，您是否认为目前的泛型代码对于普通用户来说太难以理解了？

Bjarne：是的。很多泛型代码和模板元编程代码太难以理解了，而且，对于哪怕很微小的用户错误给出的错误消息也非常难以理解。

要想使得模板定义成为主流，我们必须使得模板更容易编写和使用。当前的很多用法过于聪明。优秀的代码应该是简单的（相对于现状而言），并且易于检查，易于优化（即高效）。

荣耀：C++0x会支持元数据和反射吗？此外，你对Boost.Serialization库又怎么看？

Bjarne：C++0x不太可能支持元数据。我对Boost.Serialization库不予置评。

荣耀：为何C++0x无意支持元数据和反射？看上去它是不少C++程序员“梦寐以求”的特性。

Bjarne：很多程序员梦寐以求很多语言特性，如果C++将它们统统采纳进来，C++就会因为过度膨胀而失去用处。元数据这样的语言特性需要付出较多的工作，已经超出了标准委员会的能力。而且我并不认为元数据很好地适合于C++对“静态类型以及在concepts和types之间的直接对应”的强调。元数据可以被用于隐藏“走访”数据结构（元数据）的例程中的程序的真实语义，我对此类用法的正确性和性能尚持怀疑态度。

荣耀：在Unicode、XML以及网络方面，有没有C++0x标准库的好的候选者？

Bjarne：关于Unicode的工作有在做，但主要是C风格的。除了线程和sockets外，尚无严肃的关于XML、网络方面的工作。虽然应该进行这方面的工作，但标准委员会好像没有足够的资源来做这些事。这儿的资源指的是“有时间、有兴趣的技术能手”。

荣耀：一些人认为当前C++对TMP（Template Metaprogramming，模板元编程）的支持不够好，就像C对OOP的支持那样，仅仅勉强可行，您是否赞同这一点？为了更好地支持TMP，您认为还需要向C++0x中添加什么样的特性？

Bjarne：就目前来看，C++对TMP的支持似乎比任何语言都要好。毋庸置疑，我们可以改进语言使其更好地支持TMP，问题是我们到底该不该这么做？什么样的做法才算是一种改善？什么样的程序最好执行于编译期？什么样的程序又应该采用通用语言特性进行实现？没错，我们是可以设计一种专用的语言，与C++相比它对TMP有着更好的支持，就像我们可以为任何特殊的任务设计一种专用的语言那样。问题还是那句话：这样做值得吗？在C++0x（标准库）的设计中，中心议题仍然是经典的泛型编程。

荣耀：一个关于TMP的小问题。C++98标准建议编译器实现至少要支持17层递归模板实例化，但由于这只是一个“建议”而非“规定”，一个即使只支持1层递归实例化的编译器也是遵从标准的，不是吗？换句话说，TMP的重要基础之一——递归模板实例化并未得到标准更为“名正言顺”的支持——这项技术是和特定编译器有关的。C++0x有意增强这一点吗？

Bjarne：我不知道，不过17这个数字也太小了。

荣耀注：实际上，我们常用的编译器对递归模板实例化的层数支持远大于17层，一般可达数百乃至数千层。

荣耀：考虑到要添加现有C++98编译器所不支持的新语言特性，标准委员会在进行C++0x标准化工作的过程中，都采用哪些编译器来测试（验证）新语言特性？

Bjarne：有很多编译器可以用来测试新语言特性。所有程序员都可以使用GCC进行尝试，学院人士可以使用EDG。所有编译器厂商都在他们的编译器中试验新语言特性。

荣耀注：EDG（Edison Design Group）是一家专注于开发C/C++、Java以及Fortran 77编译器front end（用于将源代码翻译成某种中间语言，然后再由back end将中间语言翻译成机器码）的公司。Intel C++和Comeau

C/C++等许多编译器都使用了EDG front end。

荣耀：一个有点儿八卦的问题。C++标准委员会中有中国人吗？有中国人向C++标准委员会递交过提案吗？

Bjarne：我想不起来最近的提案和中国人有关。委员会中有一个IBM的新代表，姓王。我猜他是中国人，但我还不认识他。

考虑到中国有那么多人在从事计算机工作，我一直都很奇怪为什么看不到你们对C++0x标准化工作的参与。

其他

荣耀：和运行期C++相比，编译期C++、尤其是模板元编程的优点何在？TMP最适合用于哪些场合？TMP已经成功地应用于哪些类型的应用中了？

Bjarne：对于这一个问题我还没有确定的答案。我认为TMP主要适合于相对简单的编译期计算，即用于选择执行于运行期的代码，而不是用于获得数值结果的复杂的计算，例如计算阶乘或质数。在这样的场合下，TMP充当“高级的”#ifdef。

荣耀注：“计算质数”意思是求出小于给定正整数的所有质数。

荣耀：是的。所以像Boost Type Trait这样的库已经被写进了Library TR。另外，我认为Blitz++这样的库对于科学计算来说还是非常有意义的。

Bjarne：对。

荣耀：一个基础的问题。为什么在TMP中不能使用浮点型数据？这儿存在什么技术上的困难吗？我没看到C++0x有支持编译期浮点型数据计算的迹象，您知道，这对于编译期数值计算来说还是很有意义的。

Bjarne：浮点计算不是百分之百精确，我们不能依赖于它的计算结果进行比较……

荣耀：看上去在编译期C++和运行期C++之间存在着一道鸿沟，我们如何修补它？抑或如何将二者结合使用？

Bjarne：问题仍然是：我们应该修补它吗？我们能从对TMP的显著改善中获得什么样的好处？

荣耀：您说过C++是一门“偏向于系统编程的通用语言”。“系统编程”是一个宽泛的概念，不同的人有着不同的理解，您的意思是？

Bjarne：当我说系统编程时，我是指传统上与操作系统以及基础工具有关的编程任务。包括操作系统核心、设备驱动程序、系统工具、网络应用、编辑器、字处理工具、编译器、某些图形和GUI应用，以及数据库系统等。这类工作在当前的C++用户中占有主导地位（参见我的个人主页上的“Applications”单元）。

荣耀：尽管我们都知道在C++中面向对象和泛型编程同等重要，但是，您是否和大多数人一样，认为C++对泛型编程的强力支持是它与其他语言的显著区别？

Bjarne：对泛型编程的强力支持只是显著的区别和显著的优势之一，即便与那些提供了“generics”的语言相比也是如此。一如既往，C++的主要力量不在于在某一方面表现完美，而是擅长于很多事情，因此，对于许多应用领域来说，C++都是一个优秀的工具。

荣耀：看上去C++程序的编译速度要比Java或C#的来得慢，原因何在？当然了，我们可以想象语言自身的复杂性以及模板的实例化（即使程序员没有显式地使用模板，但他所使用的库比如STL，也使用了模板）等因素使然。

Bjarne：Java和C#这样较新的语言，其语言结构较为简单。更重要的是，它们不像C++这样在编译期做很多事情。形形色色的编译期评估是造成慢的一个主要原因，不过编译时间被拉长带来的好处往往是程序运行期性能的提高。还有一个原因值得一提，连接时间往往也被认为是构成C++编译时间一个不小的组成部分。

荣耀：多年以来——今天仍然如此，有很多人在讨论什么是“高级C++”，您能谈谈什么是高级C++吗？C++/高级C++的明天是什么样子？

Bjarne：在我看来，人们在为了变得更聪明、更高级方面用力过度了。设计和编程的真正目标在于产生能够完成工作的最简单的解决方案，并且对该解决方案的表达要尽可能的清晰。理想的境界不是让那些看到你代码的人惊呼“哇塞，好聪明耶！”，而是“哈哈，这么简单？”

我认为这样的简单代码将会包含大量的相对简单的泛型编程，同时带有一些类继承层次结构，后者为需要运行期解决方案的领域提供服务。用今天的话来说，此即为“多范型编程（multi-paradigm programming）”，但我们无疑需要为之寻找一个更好的术语。

参考资源

1. The C++ Standards WG, http://www.open-std.org/jtc1/sc22/wg21/
2. Bjarne Stroustrup, The Design and Evolution of C++, Addison-Wesley
3. Bjarne Stroustrup, The Design of C++0x, C/C++ Users Journal (May 2005)
4. Bjarne Stroustrup homepage, http://www.research.att.com/~bs

C++0x 概览

来源：荣耀 李建忠 译 作者：Bjarne Stroustrup 等级：精品
发布于2006-10-22 13:03 被读1166次 【字体：大 中 小】

C++0x概览

— 2005上海“Modern C++ Design & Programming”技术大会致辞

Bjarne Stroustrup

C++0x的工作已经进入了一个决定性的阶段。ISO C++委员会对C++0x的目标是使其成为“C++09”。这意味着我们要在2008年完成这个标准以便被ISO成员国批准。最后提交的标准设施将选自目前正被讨论的提案。为了按时完成此项工作，委员会已经停止审查新的提案并将精力集中于已经被讨论的那些提案上。

本文简要描述了C++0x标准化工作的指导原则，展示了一些可能的语言扩展例子，并列出了一些被提议的新标准库设施。

说明：本文首发于2005年11月18-20日在上海举行的“Modern C++ Design & Programming”技术大会。

指导原则

C++是一门偏向于系统编程的通用编程语言。它

- 是一个更好的C
- 支持数据抽象
- 支持面向对象编程
- 支持泛型编程

当我说“系统编程”时，我是指传统上与操作系统以及基础工具有关的那一类编程任务。包括操作系统核心、设备驱动程序、系统工具、网络应用、字处理工具、编译器、某些图形和GUI应用、数据库系统、游戏引擎、CAD/CAM、电信系统，等等。这类工作在当前的C++用户中占有主导地位。例子参见我的个人主页“Applications”单元（http://www.research.att.com/~bs/applications.html）。

C++0x的目标是使以上的说法仍然成立。它并不是要消除这些编程风格（styles）（或“paradigms”，范型）之一（比方说，使C++不那么兼容于C），或者添加一种全新的“范型”。最有效的编程风格是联合使用这些技术，这也就是我们常说的“多范型编程（multi-paradigm programming）”。因此，可以说我们希望改进C++使其成为一门更好的多范型编程语言。

C++0x的高级目标是

- 使C++成为一门更好的系统编程语言和构建库的语言。

- 而不是为特定子社群提供专用设施（例如数值计算或Windows风格的应用程序开发）。

- 使C++更易于教和学。

- 通过增强的一致性、更强的保证以及针对新手的设施支持。

换句话说，在C++98已经很强的领域（以及一些更多的、C++98支持的较为自然的、一般化的领域），C++0x应该比C++98做得更好。对于一些专有的应用程序领域来说，例如数值计算、Windows风格的应用程序开发、嵌入式系统编程，C++0x应该依赖于程序库。C++在基本语言特性（如基于栈的对象和指针）方面所具有的效率，以及在抽象机制（如类和模板）方面所具有的通用性和灵活性，使得库在非常广泛的应用领域都能保持 其吸引力，并因此降低C++对各种新的语言特性的需求。

我们不能为了降低C++在教与学方面的难度而移除某些语言特性。保持C++稳定性与兼容性是主要的考虑。因此，不管以什么方式移除其中任何重要的特性都是行不通的（而移除其中不重要的特性对于解决问题又没有实质性的帮助）。那么留给我们的选择恐怕只有“将规则一般化”和“添加更易于使用的特性”。两者都是我们的目标，但是后者更容易一些。例如，更好的程序库（容器与算法）可以帮助用户避免一些底层设施（例如数组与指针）带来的问题。那些能够“简化程序库的定义和应用”的语言设施（例如“concepts”与“通用初始化器列表”，下面将会谈到它们）也将有助于改善C++0x的易用性。

一些人可能对此持有反对意见，“不要为了新手而将C++降格，适合新手的语言已经有很多了！”，或者“最好的办法还是将新手变成专家！”这些人的观点并非毫无道理，但是现实是新手总比专家要多。而且许多C++用户完全不必、也没有意愿成为C++专家——他们是各自领域的专家（比如物理学家、图形学专家、硬件工程师），只不过他们需要使用C++。在我个人来看，C++已经太过“专家友好”了，我们完全可以在花销很少的情况下为“新手们”提供更好的支持。事实上，这种支持不会损及任何C++代码的性能（零开销原则依旧适用）、灵活性（我们不打算禁止任何东西）及简洁性。相反，我们的目标是简化这些理念的表达。最后，值得指出的是，C++是如此之大，而且应用如此广泛，各种设计技巧可谓汗牛充栋，以至于我们很多时候也都是“新手”。

C++0x的改进应该以这样的方式进行：结果所得语言应该更易于学和用。以下是委员会考虑的一些规则：

- 提供稳定性和兼容性（针对C++98而言，可能的话还有C）
- 优先考虑库设施，其次才是语言扩展
- 只进行可以改变人们思维方式的修改

- 优先考虑一般性而非专用性
- 同时为专家和新手提供支持
- 增强类型安全性（通过为当前不安全的设施提供安全的替代品）
- 改进直接处理硬件的性能和能力
- 适应现实世界

当然，对这些思想和规则的应用与其说是一门科学不如说是一门艺术，人们对于什么才是C++的自然发展以及什么才是一种新的范型有着不同的意见。C++0x将极有可能支持可选的垃圾收集机制，并将以一个机器模型外加支持线程的标准库设施（可能还有别的）来支持并发编程。一些人也许认为这过于激进，但我并不这么认为：人们已经在C++中（在垃圾收集有意义的领域）使用垃圾收集很多年了，而且几乎每一个人都曾使用过线程。在这些情况下，我们需要做的不过是将现行的实践加以标准化而已。

我们之所以专注于“只进行可以改变人们思维方式的修改”，是因为这种方式可以使我们的努力获得最大的回报。每一项改变都要付出代价，不管是在实现方面，还是在学习等其他方面。而且，每 一项改变的代价并非总是直接与其带来的回报相称。语言上的主要进步/收益并非体现在如何改进程序员编写的某一行代码上，而是体现在如何改进程序员解决问题和组织程序的方式上。面向对象程序设计和泛型程序设计改变了很多人的思维方式 — 这也是C++语言设施支持这些风格的 用意。因此，作为语言和程序库的设计者，最佳做法就是将时间投入到那些能够帮助人们改变思维方式的设施和技巧上。

请注意最后一条规则“适应现实世界”。一如既往，C++的目标不是创建一门“最美丽”的语言（尽管只要有可能我们都希望“美丽”），而是提供最有用的语言。这就意味着兼容性、性能、易于学习，以及与其他系统和语言的互操作性，才是应该严肃考虑的问题。

语言特性

让我们来看看使用C++0x新特性的代码的可能模样：

```
template<class T> using Vec = vector<T,My_alloc<T>>;
Vec<double> v = { 2.3, 1.2, 6.7, 4.5  };
sort(v);
for(auto p = v.begin(); p!=v.end(); ++p)
   cout << *p << endl;
```

在C++98中，除了最后一行代码外其余每一行都是不合法的，而且在C++98中我们不得不编写更多（易犯错误）的代码来完成工作。我希望无需解释你就可以猜测到这段代码的含义，不过我们还是逐行看一看。

```
template<class T> using Vec = vector<T,My_alloc<T>>;
```

在这里，我们定义Vec<T>作为vector<T,My_alloc<T>>的别名。换句话说，我们定义一个名为Vec的标准vector，其工作方式正如我们常用的vector那样，除了它使用我自己定义的配置器（My_alloc）而非默认的配置器之外。C++中缺乏定义这种别名以及绑定（bind）部分而非全部模板参数的能力。按照传统，这被称为“template typedefs”，因为我们一般采用typedef来定义别名，但出于技术上的原因，我们偏向于使用using。这种语法的优势之一是，它将被定义的名字展示于易被人们发现的显著位置。还要注意另一个细节，我没有像下面这样写：

```
template<class T> using Vec = vector< T,My_alloc<T> >;
```

我们将不再需要在表示结束符的两个“>”之间添加空格。原则上这两个扩展已经被接受了。

接下来我们定义和初始化一个Vec：

```
Vec<double> v = { 2.3, 1.2, 6.7, 4.5  };
```

采用一个初始化列表来初始化用户自定义容器（vector<double,My_allocator<double>>）是一种新方式。在C++98中，我们只能将这种初始化列表语法用于聚合体（包括数组和传统的struct）。至于究竟如何实现这种语言扩展仍然在讨论中。最可能的解决方案是引入一种新型构造函数：“序列构造函数”。允许上面的例子可以运作意味着C++将更好地满足其基础设计准则之一：对用户自定义类型和内建类型的支持一样好。在C++98中，数组比vector具有记号上的优势。在C++0x中，情况将不再如此。

接下来，我们对该vector进行排序：

```
sort(v);
```

为了在STL的框架内做这件事，我们必须针对容器和迭代器对sort进行重载。例如：

```
template<Container C> // 使用<对容器排序
   void sort(C& c);

template<Container C, Predicate Cmp> // 使用Cmp对容器排序
   where Can_call_with<Cmp,typename C::value_type>
   void sort(C& c, Cmp less);
```

```
template<Random_access_iterator Ran> // 使用<对序列排序
    void sort(Ran first, Ran last);

template<Random_access_iterator Ran, Predicate Cmp> // 使用Cmp对序列排序
    where Can_call_with<Cmp,typename Ran::value_type>
    void sort(Ran first, Ran last, Cmp less);
```

这里演示了C++0x目前提案中最有意义的扩展部分（也是有可能被接受的部分）：concepts。基本上，一个concept就是一个type的type，它指定了一个type所要求的属性。在这个例子中，concept Container用于指定前两个版本的sort需要一个满足标准库容器要求的实参，where子句用于指定模板实参之间所要求的关系，即判断式（predicate）可以被应用在容器的元素类型上。有了concepts，我们就可以提供比目前好得多的错误消息，并区分带有相同数目实参的模板，例如：

```
sort(v, Case_insensitive_less());   // 容器与判断式
```

和

```
sort(v.begin(), v.end());          // 两个随机访问迭代器
```

在concepts的设计中存在的最大的困难是维持模板的灵活性，因此我们不要求模板实参适合于类层次结构或要求所有操作都能够通过虚函数进行访问（就象Java和C#的泛型所做的那样）。在这些语言的“泛型”中，实参的类型必须是派生自泛型定义中指定的接口（C++中类似于接口的是抽象类）。这意味着所有的泛型实参都必须适合于某个类层次结构。这将要求开发人员在设计时要进行一些不合理的预设，从而为他们强加了一些不必要的约束。例如，如果你编写了一个泛型类，而我又定义了一个类，只有在我知道你指定的接口、并将我的类从该接口派生的情况下，人们才可以将我的类用作这个泛型类的实参。这种限制太过严格。

当然对于这种问题总有解决办法，但那会使代码变得复杂化。另一个问题是我们不能直接在泛型中使用内建类型。因为内建类型（例如int）并不是类，也就没有泛型中指定接口所要求的函数——这时候你必须为这些内建类型做一个包装器类，然后通过指针间接地访问它们。另外，在泛型上的典型操作会被实现为一个虚函数调用。那样的代价可能相当高（相对于仅仅使用简单的内建操作来说，比如+或者<）。以这种方式来实现的泛型，只不过是抽象类的“语法糖”。

有了concepts后，模板将保持它们的灵活性和性能。在委员会可以接受一个具体的concept设计之前，仍然有很多工作要做。然而，由于承诺显著更好的类型检查、更好的错误信息和更好的表达力，concepts将 会成为一个可能性极大的扩展。它允许我们从目前的标准容器、迭代器以及算法开始就设计出更好的库接口。

最后，考虑最后一行用于输出我们的vector元素的代码：

```
for (auto p = v.begin(); p!=v.end(); ++p)
    cout << *p << endl;
```

这儿与C++98的区别在于我们不需要提及迭代器的类型：auto的含义是“从初始化器（initializer）中推导出所声明的变量的类型”。这种对auto的使用方式可以大大消除当前方式所导致的冗长和易出错的代码，例如：

```
for (vector< double, My_alloc<double> >::const_iterator p = v.begin(); p!=v.end(); ++p)
    cout << *p << endl;
```

这儿提到的新的语言特性的目标都在于简化泛型编程，原因在于泛型编程已经是如此流行，“使得现有语言设施受到了很大的压力”。许多“modern”的泛型编程技术接近于“write only”技术，并有孤立于其用户的危险。为了使得泛型编程成为主流（就象面向对象编程成为主流那样），我们必须使模板代码更易于阅读、编写和使用。许多目前的用法只管编写时的好处。但真正好的代码应该简洁（相对于它要做的事情来说）、易于检查且易于优化（即高效）。这就意味着许多简单的思想可以在C++0x中简单地进行表达，并且结果代码坚定不移地高效。在C++98中前者的情况可不是这样，至少对于 很大范围的依赖于模板的技术的情形并非如此。借助于更好的类型检查机制以及对类型信息更广泛的使用，C++代码将变得更简短、更清晰、更易于维护，且更易获得正确性。

库设施

从理想上说，我们应该尽量不修改C++语言，而集中于扩充标准库。然而，那些具有足够大通用性的能够进入标准的库设计起来并不容易，而且一如既往，标准委员会缺乏足够的资源。我们由相对少的一组志愿者构成，并且都有“日常工作”。这就给我们能对新库进行的冒险增添了不幸的限制。另一方面，委员会很早就启动库的工作了，一个关于库的技术报告（Library TR）也在最近被投票通过，它提供了一些对程序员来说具有直接的用处的设施：

- 哈希表（Hash Tables）
- 正则表达式（Regular Expressions）
- 通用智能指针（General Purpose Smart Pointers）
- 可扩展的随机数字设施（Extensible Random Number Facility）
- 数学专用函数（Mathematical Special Functions）

我尤其赏识能够有标准版本的正则表达式和哈希表（名为unordered_map）。此外，Library TR还为基于STL构建泛型库的人们提供了广泛的设施：

- 多态函数对象包装器（Polymorphic Function Object Wrapper）
- Tuple类型
- Type Traits
- 增强的成员指针适配器（Enhanced Member Pointer Adaptor）
- 引用包装器（Reference Wrapper）
- 用于计算函数对象返回类型的统一方法（Uniform Method for Computing Function Object Return Types）
- 增强的绑定器（Enhanced Binder）

这儿不是详述这些库的细节或者深入讨论委员会希望提供的更多的设施的场合。如果你对此感兴趣，我建议你浏览WG21站点（参见后面的“信息资源”）上的提案、库期望列表（wish list）（在我的主页上），以及BOOST库（www.boost.org）。我个人希望看到更多的对应用程序构建者有着直接好处的库，例如Beman Dawes的用于操纵文件和目录的库（当前是一个BOOST库）以及一个socket库。

目前的提案列表仍然相当的保守，并不是各个地方都如我所期望的那样进取。不过，还有更多来自于委员会海量的建议中的提案正被考虑，将有更多的库或者成为C++0x标准的一部分、或者进入将来委员会的技术报告。不幸的是，资源的缺乏（时间、财力、技能、人力等）仍将继续限制我们在这个方向上的进展。悲哀的是，我无法给大家太希望得到的一个新标准库 — 一个标准GUI库 — 带来希望。GUI库对于C++标准委员会的志愿者来说是一个太大的任务，而且是一个太困难的任务，因为已经有很多（非标准、大型、有用但受支持的）GUI库的存在。请注意，纵然它们是非标准的，主要的C++ GUI库还是有比大多数编程语言更多的用户，并且通常有更好的支持。

除了这些通用的库之外，委员会还在“Performance TR”中提呈了一个到最基层的硬件的库接口。该TR的首要目标是帮助嵌入式系统程序员，同时还驳斥了有关C++代码性能低下以及C++正变得不适合低层任务的流言蜚语。

总结

“将一个数组中所有的形状绘制出来”是面向对象编程中一个经典的例子（不妨回忆早期Simula的日子）。使用泛型编程，我们可以将其泛化，从而支持绘制任意容器（存储着Shape指针）中的每 一个元素。

```
template<Container C>
void draw_all(C& c)
where Usable_as<typename C::value_type,Shape*>
{
   for_each(c, mem_fun(&Shape::draw));
}
```

在C++0x中，我们希望将Container作为一个标准concept，将Usealbe_as作为一个标准判断式。其中for_each算法在C++98中 已经存在了，但是接受容器（而非一对迭代器）作为参数的版本要依赖于C++0x中的concept。其中的where子句用于支持算法表达其对实参的要求。就这里而言，draw_all()函数（明确）要求容器中的元素必须可以被作为（即可以隐式转换为）Shape*使用。这里的where子句通过简单要求一个Shape*容器，为我们提供了某种程度的灵活性和通用性。除了元素为Shape*的任何容器外，我们还可以使用那些元素可被用作Shape*的任何容器，例如list<shared_ptr<Shape*>>（其中shared_ptr有可能成为C++0x标准库中的一个类）、或者元素类型继承自Shape*的容器，例如deque<Circle*>。

假设我们有p1、p2、p3三个点，我们可以编写如下代码来测试draw_all()：

```
vector<Shape*> v = {
   new Circle(p1,20),
   new Triangle(p1,p2,p3),
   new Rectangle(p3,30,20)
};

draw_all(v);

list<shared_ptr<Shape*>> v2 = {
   new Circle(p1,20),
   new Triangle(p1,p2,p3),
   new Rectangle(p3,30,20)
};
```

draw_all(v2);

以上例子非常重要，如果你可以很好地实现它，你就掌握了大多数面向对象编程中关键的东西。通过融合泛型编程（concepts与模板）、常规编程（例如独立标准库函数mem_fun()）以及简单数据抽象（mem_fun()函数返回的函数对象），上面的代码演示了多范型编程的力量。这个简单的示例为我们开启了通往许多优雅和高效的编程技巧的大门。

我希望在看完上面的例子之后，你的反应是“真简单！”，而不是“真聪明！真高级！”

在我看来，许多人都在聪明和高级的道路上太过投入。但设计和编程的真正目的在于使用最简单的方案完成工作，并以尽可能清晰的方式表达出来。C++0x设计的目标便是更好地支持这样的简单方案。

信息资源

我的主页（http://www.research.att.com/~bs）上包含有大量的有用的信息。那儿有我自己的作品（书籍、文章、访谈、FAQ等）的信息，以及我发现很有帮助作用的资源的链接（一个有意思的C++应用程序列表，一个C++编译器列表，以及到有用的库的链接（例如BOOST），等等）。关于C++0x，你可以发现：

- 语言特性和库设施的“期望列表（wish lists）”
- 标准：IOC/IEC 14882—International Standard for Information Systems—Programming Language C++
- Performance TR：ISO/IEC PDTR 18015—Technical Report on C++ Performance
- Library TR：JTC1.22.19768 ISO/IEC TR 19768—C++ Library Extensions
- 到WG21（ISO C++ 标准委员会）站点的链接，在那儿你可以看到所有正被讨论的提案
- 一个包含有我给委员会的提案（包括concept）的页面（请记住并非所有提案都会被接受，并且基本上所有被接受的提案，在被接受之前都会有较大的变化和改进）

致谢

感谢荣耀鼓励我对许多地方进行了澄清。同时感谢Nicholas Stroustrup、Bjorn Karlsson，以及来自我的689班级的有益的反馈。

关于作者

Bjarne Stroustrup是C++语言的设计者和第一位实现者。现任美国德州农工大学（Texas A&M University）计算机科学系首席教授，此前他一直担任AT&T实验室的大规模程序设计研究部门（自其2002年底创建以来）的主管。

http://blog.csdn.net/snailjava/article/details/1646521

上一篇 《C++ Primer》作者Stanley B.Lippman谈C++语言和软件产业的发展

下一篇 使用隐式转换auto_ptr导致程序崩溃原因

查看评论

暂无评论

发表评论

用户名： zjufirefly

评论内容：

提交

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

核心技术类目

全部主题 Hadoop AWS 移动游戏 Java Android iOS Swift 智能硬件 Docker OpenStack
VPN Spark ERP IE10 Eclipse CRM JavaScript 数据库 Ubuntu NFC WAP jQuery
BI HTML5 Spring Apache .NET API HTML SDK IIS Fedora XML LBS Unity
Splashtop UML components Windows Mobile Rails QEMU KDE Cassandra CloudStack
FTC coremail OPhone CouchBase 云计算 iOS6 Rackspace Web App SpringSide
Maemo Compuware 大数据 aptech Perl Tornado Ruby Hibernate ThinkPHP HBase
Pure Solr Angular Cloud Foundry Redis Scala Django Bootstrap