C++11FAQ 中文版

声明：

我（http://blog.csdn.net/jhkdiy）只是将 C++11FAQ 网页版整理转换为 PDF 版本，以便阅读，所有版权归源翻译者（陈良乔等人）和作者（Stroustrup 先生）所有。

源网址：http://www.chenlq.net/cpp11-faq-chs

C++11 FAQ 中文版

2012-02-15 14:20 by 陈良乔, 25,508 次阅读, 35条评论

更新至英文版 October 3, 2012

译者前言

经过 C++标准委员会的不懈努力，最新的 ISO C++标准 C++11，也即是原来的 C++0x，已经正式发布了。让我们欢迎 C++11！

今天获得 Stroustrup 先生的许可，开始翻译由他撰写和维护的 C++11 FAQ。我觉得这是一件伟大而光荣的事情，但是我又觉得压力很大，因为我的英语水平很差劲，同时自己的 C++水平也很有限，很害怕在翻译过程中出现什么错误，贻笑大方不要紧，而误人子弟就罪过大了。所以，我这里的翻译只能算是抛砖引玉，如果你的英文很好，你可以直接阅读他的原文。或者，你也可以参照两者进行阅读，我想一定会有更多的收获。

当然，我也非常欢迎大家指出翻译中的错误，或者是加入进来和我一起翻译这份文档，共同为 C++11在中国的推广做一点事情。你可以通过 chenlq at live.com 联系到我。

对自己的翻译做一点说明：

在翻译的过程中，尽量遵照原文含义，可能有时候也会自己根据自己的理解加一点批注，希望可以帮助大家理解。

另外，虽然 C++11刚刚公布，但是现在已经有很多编译器支持 C++11中一些相对比较独立的特性，比如 gcc 以及它在 Windows 下的 MinGW，Visual C++ 2012也部分支持，大家可以使用这三款编译器尝试这个文档中的部分例子。

在下面的目录中，已经翻译的问题链接到相应的中文文档，未翻译的问题则链接到英文原文。

感谢所有参与翻译的志愿者(排名不分先后)：interma，Chilli，张潇，dabaidu，Yibo Zhu，lianggang jiang，nivo，陈良乔

在这里有一份 Stroustrup 先生关于 C++11的访谈，可以帮助你从更高地角度把握整个 C++11新标准，你应该阅读一下。

最后，祝大家阅读愉快:)

C++11FAQ

Stroustrup 先生关于中文版的授权许可邮件

On 2/18/2011 8:03 PM, ChenLiangqiao wrote:

Dear Mr Stroustrup,

Just as you said, there will be a long time before we can read your new book, the 4th edition of the C++ Programming Language.And it will take another long time to translate it into Chinese. Instead, may i translate some parts of your C++0x FAQ into Chinese and post it on my blog? Then, many Chinese C++ Developers will learn and use the C++0x much more early.

Certainly. I think that is a great idea and would be a great service to Chinese programmers. I have just a few requests:

Keep what you translate in one document so that I (and others) can link to it, rather than scattering its pieces all over your blog (you can of course refer to it from your blog as your translations progresses).

Insert a link back to my original

Translate whole sections rather than parts of sections

Stroustrup 先生关于 C++11 FAQ 的一些说明

这份文档由 Bjarne Stroustrup 进行编写并维护。任何建设性的意见，校正，引用和建议，都是欢迎的。目前，我正在努力让这份文档更加完善并进行一些参考的清理工作。

C++11是下一个国际标准组织 ISO 的 C++标准。目前，已经有草案可供大家参考提出意见。提供意见。以前的（和目前的）标准通常被称为为 C++98和 C++03。C++98和 C++03之间的差异很小并且太过技术化，不应当引起用户的关注。

最终的标准委员会草案已经于2010年3月由国家标准机构表决通过。在让所有反馈意见都得到处理并让 ISO 的官员们都满意之前，还有很多工作要做。在现阶段，任何功能（即使是很小的）都不要指望被添加进入标准或者从标准中移出。C++0x 这个名字只是我和其他人使用之后留下的一个遗留物，我们原本希望它是 C++08或 C++09。然而，为了减少混淆，我会继续谈到即将到来的 C++标准，它有着与我们在这里为 C++0x 定义的相同的功能特性。我们可以把 x 看成是一个十六进制数，就像‘B’，这样 C++0x 就成了 C++11。（译注：C++0x 是这个新标准的代称，等标准通过之后，这个标准很可能被称为 C++11。再译注：已经被正式确定为 C++11了。）

如果你曾经就 C++0x 提出过一些建议，请找你们国家的标准化组织，或者是任何的标准化组织，向他们提交你关 C++0x 的建议和意见。目前，这是唯一的提交意见和建议的途径，这样可以保证标准委员会不用处理来自不同途径的相似的意见和建议。请记住，标准委员会全部由志愿者组成，他们的时间是有限的。

所有关于 C++11的官方文档都可以在 ISO C++标准委员会的官方网站上找到。标准委员会的官方名字是 SC22 WG21。

请注意：这份 FAQ 将在很长一段时间内都是处于建设状态。任何的意见，建议，问题，参考，更正都是欢迎的。

关于 C++11 的一般性的问题：

您是如何看待 C++11 的?

对于我来说，这是一个最最容易被问到的问题。它可能是被问到的次数最多的问题。让人吃惊的是，C++11就像一种新的编程语言：跟以前旧的 C++不同，C++11的各个部分被更好地组合在一起，并且我找到了一种更加自然的高层次的编程方式，而且同样有很好的效率。如果你仅仅是将 C++当作更好的 C，或者是一种面向对象语言，那么你将错过其中非常精彩和关键的东西。C++11中的抽象机制将比以前更加灵活，并且更加经济实惠。就像古老的“咒语”一样：如果你的头脑中有一个想法或者对象，想要在程序中直接对其进行表现，那么，你需要对现实世界中的对象进行建模，并在代码中对其进行抽象。现在这一过程更加容易了：你的想法将直接对应成为枚举、对象、类（例如，对默认值进行控制）、类的继承（例如，继承的构造函数）、模板、别名、异常、循环、线程等。这将远远好于以前那种简单的“以一双鞋适应所有脚”的抽象机制。

我的理想是，使用编程语言的各个功能来帮助程序员从另外一个角度思考系统的设计和实现。我认为C++11可以做到这一点。并且，不仅仅是为了让 C++程序员可以做到，还包括更多的习惯于其它编程语言的，在更广泛的领域内进行系统编程的程序员都可以做到这一点。

换句话说，我依然是一个乐观主义者。

什么时候 C++11 会成为一部正式的标准呢？

就是此刻！

一份正式标准的初稿产生于2008年9月。近期（2010年3月），一份最终的标准委员会草案标准即将接受国

家标准机构的投票表决。

我们知道，新标准的看起来更象一个 modulo minor（它已经随着独立的功能特性而发生了改变 ）(?)。新标准很可能命名为 C++11，但即使是简单的官方审批程序也可能使之成为 C++12。就个人而言，当我需要区分之前版本的时候，我更喜欢简单地用 C++和年度标识来标记，例如 ARM C++，C++98和 C++03。现在，我按照惯例，将继续使用 C++0x 作为下一个 C++标准的名字。将’ X'看做十六进制数吧，这样更好理解一些。

(翻译：Chilli)

编译器何时将会实现 C++11 标准呢？

目前，业界普遍使用的已经发布的编译器（例如，GCC C++, Clang C++,IBM C++, 和 Microsoft C++）已经实现了许多 C++11的特性。例如，在发布编译器时，同时发布全部或者绝大多数的新标准库文件似乎非常普遍，并且十分受用户的欢迎。我希望越来越多的新特性会出现在每次的版本发布中。可能性最大的，相对独立的特性，像 auto, lambda, 和 strongly typed enums，我们将最早看到。我不禁猜想，何时所有编译器都将完全支持 C++11——毫无疑问，这将会需要数年的时间 ——但我注意到，每一个 C++11的特性都已经被一些人实现过，所以编译器的开发者们是有可用的实现经验可以参考的。（译注：关于各个主流编译器对 C++11的支持情况，可以参考这里

http://wiki.apache.org/stdcxx/C%2B%2B0xCompilerSupport ）

你也可以通过下面的链接获得各个主流编译器对 C++11的支持情况：

各大主流编译器支持情况比较

GCC

IBM

Microsoft

EDG

Clang

（翻译：Chilli）

我们何时可以用到新的标准库文件？

目前，随着 GCC、Clang 和 Microsoft 的实现，新标准库文件的初始版本已经开始发布，并且在 Boost 库中已经有很多标准库的组件可用。（注：Boost 库是一个可移植、开放源码的 C++库，作为标准库的后备，是 C++标准化进程的发动机之一。）

（翻译：Chilli）

C++11 将提供何种新的语言特性呢？（请参考以下关于语言特性的问题）

你当然不会仅仅因为别人的一个想法，就给语言添加一个特性。事实上，关于 C++，基本上每一个最现代化的语言特性都有人向我建议过，试着想象一下，C99、C#、Java、Haskell、Lisp、Python 还有 Ada 的扩展集会是个什么样子？（译注：如果想着把这些语言的特点都集合到 C++上，那 C++就是一个四不像了）我们想问题要想的更加深入些，记住，即使标准委员会表决认为某个旧特性是不好的，完全剔除掉也是不可行的：事实表明，用户会迫使每一个开发者在兼容选项下（或默认）继续提供过时甚至已被禁止的特性达几十年。

为了试着从洪水般的建议中选择合理的建议，我们设计了一套具体的设计目标。我们不应该完全依据设计目标(?)，而且它也不能完全的指导标准委员会的每个细节（而且依我所见也不可能完全）。

其结果就是，C++成为一种被大大改良过的抽象机制的语言。这个抽象的范围比起手工操作的专业代码，

大大增加了，而且 C + +可以优雅，灵活，零成本的表达出来。当我们提到“抽象”的时候，人们往往只是想到“分类”或“对象”。C++0x 中远不止这些：用户自定义的类型可以清晰安全的表达出来，而且类型的范围已经随着初始化列表，统一初始化，模板别名，右值引用，默认的和删除函数(?)特性以及可变参数模板等特性而不断增长扩大。而有些特性则简化了它们的实现，比如 auto, inherited constructors 和 decltype。这些增强功能足以使 C++0x 像一种新的语言。

已被接受的语言功能的列表，请参阅功能列表 。

（翻译：Chilli）

C++11 会提供哪些新的标准库文件呢？（请参考以下关于标准库的问题）

我本来也是希望看到更多的标准库文件的。然而，(我改观是因为我)注意到标准库文件中定义的篇幅就占了超过75%的规范性文字（而且这些还不包括作为参考文献的 C 标准库文件）。虽然我们中的许多人也很希望看到更多的标准库文件，但是没人责备库工作组的懈怠。值得一提的是，C++98标准库已通过新语言特性的应用，如初始化列表 ，右值引用 ，可变参数模板 和 constexpr(常量表达式) 而取得了显著改善。相比 C++98，C++11更易使用，并且能够提供更高的性能。

如果想查看所有已经被接受的标准库文件的列表，可以访问这里 the library component list .。

（翻译：Chilli）

C++11 努力要达到的目标有哪些？

C++是一种偏向于系统编程的通用编程语言，所以它应该：

- 支持数据抽象
- 支持面向对象的编程
- 支持泛型编程

C++0x 努力的总体目标是为了加强：

使 C++成为一种适用于系统编程和创建程序库的更好的语言——也就是直接利用 C++进行编程，而不是为某个特定的领域提供专门的开发语言。（例如，专门为数值计算或 Windows 的应用程序开发提供支持）。

使 C++更容易教和学——增加一致性，加强安全性，为初学者提供相关的配套组件（让 C++更容易学习和使用）（初学者总是比专家多的）。（译注：C++0x 现在真的是更好用了）

自然，这是在非常严格的兼容性约束下完成的。虽然我们在 C++0x 中引入了很多新的关键词（例如：static assert，nullptr，还有 constexpr），但是标准委员会也很少会破坏标准库中的已经让人非常满意的代码。(?)

你可以通过下面这些参考获得更多详细的信息：

B. Stroustrup: What is C++0x? . CVu. Vol 21, Issues 4 and 5. 2009.

B. Stroustrup: Evolving a language in and for the real world: C++ 1991-2006 . ACM HOPL-III. June 2007.

B. Stroustrup: A History of C++: 1979-1991 . Proc ACM History of Programming Languages conference (HOPL-2). March 1993.

B. Stroustrup: C and C++: Siblings . The C/C++ Users Journal. July 2002.

（翻译：Chilli）

指导标准委员会的具体设计目标是什么？

自然，涉及不同标准化的不同组织或个人都会有某些不同的目的，尤其是在细节和优先级方面。此外，详细的目标总是随时间的改变而变动的。请记住，委员会做不到认同每个人的意见本身也是件好事——志愿者们的资源还是非常有限的。然而，这里已经有一套在实际探讨中使用着的规范，以此来确定那种特性或是库文件可适当的用 C++0x 中：

- 保持稳定和兼容性——不要打破旧代码，而如果你非打破不可的话，不要静静的做（注：应该是让做点工作告知大家吧）。
- 重库文件而非语言拓展——在一个理想状态下，委员会中不是人人都成功的，他们中有太多人更喜欢“真实的语言性能”
- 重一般性而非专业性——聚焦于改善抽象机制（类，模板等）。
- 要专家新手都支持——新手可以通过更好的库文件及更多的一般性规则得到帮助，而专家需要一般且有效的特性。
- 提升类安全——主要的措施是通过允许程序员以避免类型不安全的性能。
- 提高性能和直接与硬件工作的能力——使 C++甚至更好的用于嵌入式系统编程和高性能计算。
- 与实际世界相符——考虑工具链，实施成本，转换问题，ABI 问题，教学和学习等注意到整合性能（新的和旧的）使之结合工作是个关键——基本上大部分的工作都是。整体大于各部分之和。另一种看待详细目的的方式是观察使用领域和使用风格：
- 机械模型和一致性——为使用现代硬件提供更强的保障和更好的设施（如多核及柔软的连贯内存模型？）。例子如 thread ABI, thread-local storage, 和 atomics ABI。
- 泛型编程——GP 也是 C ++ 98取得的巨大成就，我们需要基于经验改进对其的支持。例子像 auto 和

template aliases。

• 系统编程 – 改善与硬件相近的编程(如低级别的嵌入式系统编程),提高效率。例子有 constexpr, std::array, 和 generalized PODs.

• 库建设 – 消除抽象机制的局限性 ,效率低和不规范。例子有 inline namespace, inherited constructors, 和 rvalue references.

(翻译 : Chilli) 未整理

在哪里可以找到标准委员会的报告？

前往 the papers section of the committee’ s website 。那里有相当多的细节说明。寻找 “issues lists” 或 “state of” 列表（如 , State of Evolution (July 2008))。主要的分组有：

• Core （ CWG ）——处理语言技术事件并公式化

• Evolution （ EWG ）——处理与语言功能建议及横跨语言 / 库边界的事件问题

• 库（ LWG ）——处理库设备提案问题

以下是提出的 C++Ox 标准建议草案。

（翻译：Chilli）未整理

从哪里可以获得有关 C++0x 的学术性和技术性的参考资料？

你可以从以下这些地方获得你想要的资料：

Bjarne Stroustrup: Software Development for Infrastructure. Computer, vol. 45, no. 1, pp. 47-58, Jan. 2012, doi:10.1109/MC.2011.353. A video interview about that paper and video of a talk on a very similar topic (That's a 90 minute talk incl. Q&A).

Saeed Amrollahi:

Modern Programming in New Millenium: A Technical Survey on Outstanding features of C++0x. Computer Report (Gozaresh-e Computer), No.199, November 2011 (Mehr and Aban 1390), pages 60-82. (in Persian)

Hans-J. Boehm and Sarita V. Adve: Foundations of the C++ concurrency memory model . ACM PLDI' 08.

Hans-J. Boehm: Threads Basic . Yet unpublished technical report. // Introductory.

Douglas Gregor, Jaakko Jarvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis, and Andrew Lumsdaine: Concepts: Linguistic Support for Generic Programming in C++ . OOPSLA' 06, October 2006. // The concept design and implementation as it stood in 2006; it has improved since, though not sufficiently to save it .

Douglas Gregor and Jaakko Jarvi: Variadic templates for C++0x . Journal of Object Technology, 7(2):31-51, February 2008.

Jaakko Jarvi and John Freeman: Lambda functions for C++0x . ACM SAC ' 08.

Jaakko Jarvi, Mat Marcus, and Jacob N. Smith: Programming with C++ Concepts . Science of Computer Programming, 2008. To appear.

M. Paterno and W. E. Brown : Improving Standard C++ for the Physics Community . CHEP' 04. // Much have been improved since then!

Michael Spertus and Hans J. Boehm: The Status of Garbage Collection in C++0X . ACM ISMM' 09.

Verity Stob: An unthinking programmer's guide to the new C++ — Raising the standard . The Register. May 2009. (Humor (I hope)).

[N1781=05-0041] Bjarne Stroustrup: Rules of thumb for the design of C++0x.

Bjarne Stroustrup: Evolving a language in and for the real world: C++ 1991-2006 . ACM HOPL-III. June 2007. (incl. slides and videos). // Covers the design aims of C++0x, the standards process, and the progress up until 2007.

B. Stroustrup: What is C++0x? . CVu. Vol 21, Issues 4 and 5. 2009.

Anthony Williams: Simpler Multithreading in C++0x . devx.com.

上述列表是不完整的，随着有些人出版一些新的作品，列表中的文章会过时。如果你发现有文章应该出现在这个列表中，但事实上它并没有出现，请及时联系我进行更新。并不是所有的文献都能随着标准及时更新。我会尽力保证内容的一致性。

（翻译：nivo）

还有哪些地方我可以读到关于 C++11 的资料？

随着标准的日趋完善，有关 C++0x 的资料会愈来愈多， C++ 的各种实现（例如， GCC，Visual C++） 开始提供新的 语 言 特性 和 库 。下面是 资 源列表：

- 委 员 会的网址 .
- C++0x 草案 .
- the C++0x 维 基百科 . 积 极 维护 ，但明 显 不是委 员 会的成 员
- 帮助 页 GCC's experimental implementation of C++0x features .

（翻译：nivo）

有关于 C++11 的视频吗？

希望大家知道这真的是一个 FAQ，而不是一些列出的我个人喜欢的问题集；我不喜欢技术话题的视频，我发现视频容易分心，并且经常包括一些微小的口头上的技术错误。

B. Stroustrup, H. Sutter, H-J. Boehm, A. Alexandrescu, S.T.Lavavej, Chandler Carruth, and Andrew Sutter:

several talks and panels from the Going Native 2012 conference.

Herb Sutter:

Writing modern C++ code: how C++ has evolved over the years. September 2011.

Herb Sutter:

C++ and Beyond 2011: Herb Sutter – Why C++?. August 2011.

Try Google videos.

Lawrence Crowl:

Lawrence Crowl on C++ Threads.

in Sophia Antipolis, June 2008.

Bjarne Stroustrup:

The design of C++0x

at U of Waterloo in 2007.

Bjarne Stroustrup:

Initialization at Google in 2007.

Bjarne Stroustrup:

C++0x — An overview.

in Sophia Antipolis, June 2008.

Lawrence Crowl:

Threads.

Roger Orr:

C++0x. January 2008.

Hans-Jurgen Boehm:

Getting C++ Threads Right. December 2007.

（翻译：nivo）

C++11 难学吗?

虽然我们不能在删除大量代码的前提下从 C++中移除任何有影响的特性，C++0x 仍旧比 C++98大，所以如果你想熟知每一个规则，学习 C++0x 将会是很困难的。有两个工具可以帮助我们简化学习过程（从学习者的角度而言）

一般化: 替换，也就是用 C++0x 所提供的新特性替换 C++以前所使用的各种特性。 (例如， uniform initialization, inheriting constructors, 和 threads). (?)（译注：这一段不太理解，但是从给出的例子来看，大约是某些原来使用 C++98实现起来非常复杂的功能，现在可以在 C++0x 中轻松简便地实现，所以用 C++0x 替换 C++98，比如线程就是一个非常明显的例子。）

简单化：提供比原来的方法更加简单的第二种选择。 (例如，array, auto, range for statement, and regex，

这些特性都使得 C++的开发更加简单。）

显然，“自下而上”的教/学方式将使得这些优势毫无发挥的地方，并且目前几乎没有别的不同方式。这应该随时间而变化。

（翻译：nivo）

标准委员会是如何运行的？

ISO 标准委员会，SC22 WG21，是在 ISO 规则下运行的。奇怪的是，这些规则并非标准化的，而是随着时间的变化而变化。（译注：标准委员会的规则并不标准）

大多数国家都有活跃的 C++团体并形成了自己的国家标准。这些团体举行会议，通过网络协调一致，并向 ISO 会议推选代表。加拿大，法国，德国，瑞士，英国和美国是出席这些会议较多的国家。丹麦，荷兰，日本，挪威，西班牙和别的一些国家则是出席人数比较少的国家。

大多数通过网络召开的会议都是介于这两者之间，会议记录由标准委员会编号并存放在 WG21。

标准委员会每年召开两到三次会议 ,每次约一周的时间。这些会议的大部分工作就是工作划分 ,比如“核心 “ ,” 库 “ , ” 演化 “ , “并发” 。根据需要，也会为了解决一些迫切的问题而召开会议，比如“概念”和“内存模型” 。会议主要用来投票表决。首先，工作组举行民意投票来判断某个论点是否可以作为一个整体提案递交给标准委员会。然后，如果这个提案被接受，标准委员会将进行每人一票的投票表决。我们花费大量注意力在那些我们没有进入但是已经有很多人表现出来和国家不同意的形势 - - 这将会导致长期的争论。最后，官方草案的投票由国家标准机构通过邮件完成。

标准委员会和 C 标准组织以及 POSIX 有正式的联系，并和其他一些组织也有或多或少的联系。

(翻译：nivo)

谁在标准委员会里？

标准委员会包括大约200个人，其中有大约60位会出席每年两到三次一周时间的会议。除此之外，在一些国家还有一些国家标准组织和会议。多数成员通过出席会议，邮件讨论或提交论文供标准委员会斟酌等方式贡献自己的力量。多数成员有朋友或同事提供帮助。第一天，标准委员会召集从各个国家而来的代表，并且每一次会议由6到12个国家的代表参加。最终投票由20个国家标准组织完成。这样，ISO C++标准是一个集合了众人集体智慧而成的最终成果，而并不是人们通常认为的——它只是一些为了创造完美语言的不相干的人创造出来的空中楼阁。这个标准需要获得他们的同意，只有这样才可保证所有人可以接受标准。

很自然地，多数志愿者（并非全部）有他们自己的日常工作：我们有的人开发编译器，有的人写生成工具，有的人写程序库，还有人写应用程序（此类人很少），还有少数的研究者，还有顾问，还有编写测试工具的等等。

这有一个简短的关于组织者的列表：Adobe,Apple,Boost,EDG,Google,HP,IBM,Intel,Microsoft,Red Hat,Sun.

这还有一个标准委员会的简短的成员列表，你有可能会在网上或是著作里遇到他们：

Dave Abrahams, Matt Austern, Pete Becker, Hans Boehm, Steve Clamage, Lawrence Crowl, Beman Dawes, Doug Gregor, Howard Hinnant, Jaakko Jarvi, Francis Glassborow, Jens Maurer, Jason Merrill, Sean Parent, P.J. Plauger, Tom Plum, Gabriel Dos Reis, Bjarne Stroustrup, Herb Sutter, David Vandervoorde Michael Wong. Apologies 还有更多成员就不一一列出了。请关注一些论文的作者列表：一个标准是由很多人共同完成的，而不是一个匿名的标准委员会。

你可以从 WG21 papers 获得有关这些作者的更深入的专长介绍以获得更深的印象，但请牢记那些为标准的完成做出了巨大贡献但并没有写太多东西的人们。

（翻译：nivo）

实现者应以什么顺序提供 C++11 特性？

标准中并没有关于引入 C++0x 特性的顺序；它只是简单的列出了为了达到完整地 C++11特性所需要做的事情。然而，如果实现者分阶段引入新的 C++0x 特性，我们也认为这是合理的。毕竟，我不会使用不支持的特性。所以，一个基于“易于提供”和“对多数人有用”的理念，是早期实现的关键原则。

没有功能特性的新库取决于新的语言特性，比如可变参数模板和常量表达式 constexpr。

简单且易于实现的特性将在细小但重要的地方帮助用户：

auto

enum class 枚举类

long long

nullptr 空指针

right brackets 右括号

static_assert 静态断言

帮助实现 C++11 标准库的语言特点：

常量表达式

初始化列表

一般的和统一的初始化 (包括 预防宽转窄)

右值引用

可变参数模板

标准库用到的所有特点

相关的并发特性：

memory model 内存模型

线程的本地化存储 thread_local

atomic types 原子类型

local types as template arguments 作为模板参数的局部类型

lambdas

标准库的完整支持

PODs

如果你看得仔细，你会发现我对很多语言特点（在引入这些特性的时间上）并没有看法。很自然地，我也希望这些特性能够被尽快地实现。但是，我并没有一个关于何时这些特性应该被实现的判断。显然，每一个C++实现者都有自己的原则，所以我们不能期望他们步调一致，但是我希望他们可以稍微关注一下别人在做什么，这样可以让用户更早地开始他们的移植工作。

（翻译：nivo）

将会是 C++1x 吗？

几乎可以肯定，并且不仅仅是因为标准委员会要拖延 C++11的期限。我听到最多的希望或计划是社区应该在 C++11推出后立即开始实施。与当今科技发展水平相比，标准的发布周期太长了，所以一些人认为三年时间用来修订比较合适。而我认为5年则是更为现实的。那 C++16呢？

（翻译：nivo）

标准中的”concepts”怎么了？

概念（concept）（译注：这里翻译并不准确，请大家参照原文）是允许精确地描述模板参数的一个特性，不幸的是，社区认为未来的关于概念的工作会严重影响标准的进度，并从工作文件中移出了这个特性，相关解释可以参阅我的笔记： 移出概念的决定和 一个基于概念的 DevX 观点和 C++的启示。

即使概念将来成为后续 C++的一部分，我也不得不从该文档中删除这一章节，但是把它们放在后面：

公理 (语义假设)

概念

概念图 （concepts map）

(翻译：nivo)

有你不喜欢的 C++特性吗？

是的，有些 C++98中的特性我是不喜欢的，比如宏。问题在于，并非是我喜欢什么或者我发现它对我需要做得一些事有帮助。事实上，这个问题是，无论是否有人认为确实需要说服他人支持这个想法，或者一些用法在某些用户社区已经根深蒂固到必须提供支持的地步。

（翻译：nivo）

关于独立的语言特性的问题:

__cplusplus 宏

在 C++中，宏__cplusplus 会被设定为与现在的199711L 不同的的值（比现在的大）。

alignment(对齐方式)

有时候，特别是当我们在写操作原始内存的代码的时候，我们需要指定内存分配时的对齐方式。例如：

```
1   // 字符数组 , 但是却以 double 数据的形式对齐数据
2   alignas(double) unsignedcharc[1024];
3   alignas(16)char[100]; // 以 16 字节对齐
```

对应地， C++11 还提供了一个 alignof 操作符用以返回其参数（必须是某种类型）的对齐方式。例如：

```
1   // 返回 int 的对齐方式，每个 int 数据占有 n 个字节的内存
2   constexprintn = alignof(int);
```

参考：

Standard: 5.3.6 Alignof [expr.alignof]

Standard: 7.6.2 Alignment specifier [dcl.align]

[N3093==10-0083] Lawrence Crowl: C and C++ Alignment Compatibility . Aligning the proposal to C’ s later proposal.

[N1877==05-0137] Attila (Farkas) Feh r: Adding Alignment Support to the C++ Programming Language . The original proposal.

attributes(属性)

“属性”是新标准中新添加的一种语法，其目的是为了让程序员可以在代码中提供更多的额外信息（就像__attribute__，__declspec 和#pragma 一样）。与现有的语法不同的是，C++0x 通过在源代码中灵活地添加相关的属性，这个属性就会立即作用于与之相邻到句法实体。例如：

```
1       voidf [[noreturn]] ()   // f() 将永远不会返回
2       {
3           throw“error”;    // OK，但是可以抛出异常
4       }
```

```
5   // 字符数组，但是将与 double 数据类型对齐
6   unsignedcharc [[ align(double) ]] [sizeof(double)];
```

正如你看到的那样，属性被放置在两个双重中括号“[[...]]”之间，noreturn 和 align 是 C++11标准库中定义的四个属性中的两个，另外两个是：

```
1   structB {
2       virtualvoidf [[ final ]] ();   // 函数 f()不可以重写（override）
3    };
4   structD : B {
5       voidf();    // 错误：尝试在派生类中重写函数 f()
6    };
7    structfoo* f [[carries_dependency]] (inti);   // 为优化器提供帮助信息
8    int* g(int* x,int* y [[carries_dependency]]);
```

我们有理由担心属性的大量使用会引起 C++语言的混乱，很可能将产生很多 C++语言的“方言”。所以，在不影响源代码的业务逻辑的前提下，我们推荐使用属性来帮助编译器可以更好的检查代码中的错误（例如，[[final]]可以防止派生类重写基类的成员函数），或者是帮助优化器更好地优化代码（例如，[[carries_dependency]]）。

在未来的计划中，属性的一个重要用途是为 OpenMP 提供额外的辅助信息，更好地支持 OpenMP。例如：

```
1       // 使用[[omp::parallel()]]属性告诉编译器，这个 for 循环可以并行执行
2   for[[omp::parallel()]] (inti=0; i<v.size(); ++i) {
3           // ...
4       }
```

就像上面的代码展示的那样，通过指定 for 循环的[[omp::parallel()]]属性，编译器将使用 OpenMP 对这个 for 循环进行并行化处理，从而这个 for 循环将并行执行。

参考:

Standard: 7.6.1 Attribute syntax and semantics, 7.6.2-5 Alignment, noreturn, final, carries_dependency, 8 Declarators, 9 Classes, 10 Derived classes, 12.3.2 Conversion functions

[N2418=07-027] Jens Maurer, Michael Wong: Towards support for attributes in C++ (Revision 3)

[[base_check]], [[hiding]], [[override]]

atomic operations

atomic_operations

Stroustrup 尚未完成此主题，期待中

对此主题感兴趣的朋友，可以参考

C++小品：榨干性能：C++11中的原子操作（atomic operation）

VC11有点甜：原子操作和< atomic>头文件

auto——从初始化中推断数据类型

考虑下面的代码：

auto x = 7;

这里的变量 x 被7初始化，所以 x 的实际数据类型是 int。通常，我们可以这样写：

```
1    auto x = expression;
```

这样，这个表达式计算结果的数据类型就是变量 x 的数据类型。

当我们很难准确地推断一个变量的数据类型，或者是这个变量的数据类型难于书写时，我们通常利用编译器从

一个变量的初始化中探测其数据类型。考虑下面的代码：

```
1    template<classT>voidprintall(constvector<T>& v)
2    {
3          // 根据 v.begin()的返回值类型自动推断 p 的数据类型
4          for(auto p = v.begin(); p!=v.end(); ++p)
5              cout << *p << “n”;
6    }
```

为了表示同样的意义，在 C++98中，我们不得不这样写：

```
1   template<classT>voidprintall(constvector<T>& v)
2   {
3       for(typenamevector<T>::const_iterator p = v.begin();
4          p!=v.end(); ++p)
5           cout << *p << “n”;
6   }
```

当一个变量的数据类型依赖于模板参数时，如果不使用 auto 关键字，将很难确定变量的数据类型。例如：

```
1   template<classT,classs U>voidmultiply (constvector<T>& vt,
2   constvector<U>& vu)
3   {
4   // …
5   auto tmp = vt[i]*vu[i];
6   // …
7   }
```

在这里，tmp 的数据类型应该与模板参数 T 和 U 相乘之后的结果的数据类型相同。对于程序员来说，要通过模板参数确定 tmp 的数据类型是一件很困难的事情。但是，对于编译器来说，一旦确定了 T 和 U 的数据类型，推断 tmp 的数据类型将是轻而易举的一件事情。

auto 特性是 C++11中最早被提出并被实现的特性。早在1984年，我就在我的 Cfont 中实现了 auto 特性，但是由于一些兼容性问题，它没有被纳入以前的标准。当 C++98和 C98同意删除“implicit int”之后，这些兼容性问题已经不复存在了，也就是 C++语言对“每个变量和函数都要有确切的数据类型”的要求消失了。auto 关键字原来的含义（表示一个本地变量）是多余而无用的——标准委员会的成员们在数百万行代码中仅仅只找到几百个用到 auto 关键字的地方，并且大多数出现在测试代码中，有的甚至就是一个 bug。

作为简化代码的一个主要手段，auto 关键字并不会影响标准库的实现。

参考：

the C++ draft section 7.1.6.2, 7.1.6.4, 8.3.5 (for return types)

[N1984=06-0054] Jaakko Jarvi, Bjarne Stroustrup, and Gabriel Dos Reis:Deducing the type of variable from its initializer expression (revision 4).

C99 功能特性

为了与 C 语言标准保持高度的兼容性，在 C 标准委员会的协助之下，一些细小的改变被引入到 C++0x 中。

long long

扩展的整型数据类型（例如，关于可选的更长的整型数的规则）

关于 UCN 的改变[N2170==07-0030]“通过控制和在字符和字符串中的全局源字符实现禁止”(?)

宽/窄字符串的连接

Not VLAs（变量长度数组）(?)

添加了一些扩展的预处理规则

__func__ a 宏扩展到当前函数名(?)

__STDC_HOSTED__

_Pragma: _Pragma(X) 扩展成#pragma X

支持可变长度参数的宏（通过不同数目的参数对宏进行重载）

```
#define report(test, ...) ((test)?puts(#test):printf(_ _VA_ARGS_ _))
```

空的宏参数

标准库中的很多功能组件都是继承自 C99（从本质上来说，所有 C99的库的改变都是从 C89继承而来的）

参考:

Standard: 16.3 Macro replacement.

[N1568=04-0008] P.J. Plauger: PROPOSED ADDITIONS TO TR-1 TO IMPROVE COMPATIBILITY WITH C99.

枚举类——具有类域和强类型的枚举

枚举类（“新的枚举”，“强类型的枚举”）主要用来解决传统的 C++枚举的三个问题：

可转换的枚举类型默认被转换为 int 类型，在那些不需要枚举类型表现为 int 类型的情况下，这可能会导致错误发生可转换的枚举会使得它的所有枚举值在其周围的代码范围内都是可见的，则可能会导致名字冲突

不可以指定枚举的底层数据类型，这可能会导致代码不容易理解，兼容性问题并且不可以进行前向声明

枚举类（enum）（“强类型枚举”）是强类型的，并且具有类域：

```
1    enumAlert { green, yellow, election, red };// 传统枚举
2     enumclassColor { red, blue };  // 新的具有类域和强类型的枚举类
3     // 它的枚举值在类的外部时不可见的
4     //  不会默认地转换成 int
5     enumclassTrafficLight { red, yellow, green };
6
7     Alert a = 7;  //  错误，传统枚举不是强类型的，a 没有数据类型
8     Color c = 7;  // 错误，没有默认的 int 到 Color 的转换
9
10    inta2 = red;          // 正确，Alert 默认转换成 int
11   // 在 C++98中是错误的，但是在 C++0x 中是正确的
12   inta3 = Alert::red;
13   inta4 = blue;           // 错误，blue 并没有在类域中
14    inta5 = Color::blue;// 错误，没有 Color 到 int 的默认转换
15
16    Color a6 = Color::blue;  // 正确
```

正如上面的代码所展示的那样，传统的枚举可以照常工作，但是你现在可以通过提供一个类名来改善枚举的使用，使其成为一个强类型的具有类域的枚举。

因为新的具有类名的枚举具有传统的枚举的功能（命名的枚举值），同时又具有了类的一些特点（具有类域的成员和无法进行默认的类型转换），所以我们将其称为枚举类（enum class）。

因为可以指定枚举的底层数据类型，所以可以进行简单的互通操作以及保证枚举值的体积尺寸大小：

```
1    enumclassColor :char{ red, blue };// compact representation
2
3        // 默认情况下，枚举值的底层数据类型为 int
4        enumclassTrafficLight { red, yellow, green };
5
6         // E 的体积是多大呢？
7        enumE { E1 = 1, E2 = 2, Ebig = 0xFFFFFFF0U };
8         // 不管旧的规则怎么说，比如依赖于实现等
9
10        // 现在我们可以指定枚举值的底层数据类型
11       enumEE : unsignedlong{ EE1 = 1, EE2 = 2, EEbig = 0xFFFFFFF0U };
```

同样的，指定枚举值的底层数据类型，使得前向声明（译注：就是在声明定义一个枚举类之前就使用这个枚举类的名字声明变量）成为可能：

```
1   enumclassColor_code :char;    // (前向) 声明
2        voidfoobar(Color_code* p);      // 前向声明的使用
3        // ...
4        enumclassColor_code :char{ red, yellow, green, blue };// 定义
```

枚举类的底层数据类型必须是有符号或无符号的整型，默认情况下是 int。

标准库中也使用了枚举类。

为了对应特定的系统错误代码，在枚举类 errc 中

为了安全的指针指示标志，在枚举类 pointer_safety { relaxed, preferred, strict }中

为了 I/O 流错误，在枚举类 io_errc { stream = 1 }中

为了处理异步通信错误，在枚举类 future_errc { broken_promise, future_already_retrieved, promise_already_satisfied }中

其中的某些枚举类拥有操作符，比如“==”等。

参考：

the C++ draft section 7.2

[N1513=03-0096] David E. Miller: Improving Enumeration Types (original enum proposal).

[N2347 = J16/07-0207] David E. Miller, Herb Sutter, and Bjarne Stroustrup: Strongly Typed Enums

(revision 3).

[N2499=08-0009] Alberto Ganesh Barbati: Forward declaration of enumerations.

[[carries_dependency]]

Jhkdiy:网页中此链接跳到 属性一节。

复制和重新抛出异常

你怎么捕获一个异常，之后在另外一个线程上重新抛出？使用在标准文档 18.8.5 中描述的异常传递中的方法吧，那将显示标准库的魔力。

exception_ptr current_exception(); 返回一个 **exception_ptr** 变量，它将指向现在正在处理的异常（**15.3**）或者现在正在处理的异常的副本（拷贝），或者有的时候在当前没有遇到异常的时候，返回值为一个空的 **exception_ptr** 变量。只要 **exception_ptr** 指向一个异常，那么至少在 **exception_ptr** 的生存期内，运行时能够保证被指向的异常是有效的。

```
1   voidrethrow_exception(exception_ptr p);
2   templateexception_ptr copy_exception(E e);
```

它的作用如同：

```
1   try{
2       throwe;
3   }catch(...) {
4       returncurrent_exception();
5   }
```

当我们需要将异常从一个线程传递到另外一个线程时，这个方法十分有用.

常量表达式（constexpr）

常量表达式（ constexpr ） — 一般化的受保证的常量表达式

常量表达式机制是为了：

- 提供了更多的通用的值不发生变化的表达式
- 允许用户自定义的类型成为常量表达式
- 提供了一种保证在编译期完成初始化的方法（可以在编译时期执行某些函数调用）

考虑下面这段代码：

```
1    enumFlags { good=0, fail=1, bad=2, eof=4 };
2
3    constexprintoperator|(Flags f1, Flags f2)
4    {returnFlags(int(f1)|int(f2)); }
5
6    voidf(Flags x)
7    {
8        switch(x) {
9        casebad:        /* … */break;
10       caseeof:        /* … */break;
11       casebad|eof:    /* … */break;
12       default:         /* … */break;
13       }
14   }
```

在这里，常量表达式关键字 constexpr 表示这个重载的操作符“|”就应该像一个简单的表单一样，如果它的参数本身就是常量 ，那么这个操作符应该在编译时期就应该计算出它的结果来。(译注： 我们都知道 ,switch 的分支条件要求常量，而使用 constexpr 关键字重载操作符“|”之后，虽然“bad|eof”是一个表达式，但是因为这两个参数都是常量，在编译时期，就可以计算出它的结果，因而也可以作为常量对待。)

除了可以在编译时期被动地计算表达式的值之外，我们希望能够主动地要求表达式在编译时期计算其结果值，从而用作其它用途，比如对某个变量进行赋值。当我们在变量声明前加上 constexpr 关键字之后，可以实现这一功能，当然，它也同时会让这个变量成为常量。

```
constexprintx1 = bad|eof;   // ok

voidf(Flags f3)
{
    // 错误：因为 f3不是常量，所以无法在编译时期计算这个表达式的结果值
    constexprintx2 = bad|f3;
    intx3 = bad|f3;    // ok，可以在运行时计算
}
```

通常，我们希望编译时期计算可以保护全局或者名字空间内的对象，对名字空间内的对象，我们希望它保存在只读空间内。

对于那些构造函数比较简单，可以成为常量表达式（也就是可以使用 constexpr 进行修饰）的对象可以做到这一点(?)

```
structPoint {
    intx,y;
    constexpr Point(intxx,intyy) : x(xx), y(yy){}
};

constexpr Point origo(0,0);
constexprintz = origo.x;

constexpr Point a[] = {Point(0,0), Point(1,1), Point(2,2) };
constexprintx = a[1].x;   // x 变成 1
```

（zwvista 的一段评论，有助于我们理解 constexpr 的意义，感谢 zwvista。constexpr 将编译期常量概念延伸至括用户自定义常量以及常量函数，其值的不可修改性由编译器保证，因而 constexpr 表达式是一般化的，受保证的常量表达式。）

参考：

the C++ draft 3.6.2 Initialization of non-local objects, 3.9 Types [12], 5.19 Constant expressions, 7.1.5 The constexpr specifier

[N1521=03-0104] Gabriel Dos Reis: Generalized Constant Expressions (original proposal).

[N2235=07-0095] Gabriel Dos Reis, Bjarne Stroustrup, and Jens Maurer: Generalized Constant

Expressions — Revision 5 .

decltype – 推断表达式的数据类型

decltype(E)是一个标识符或者表达式的推断数据类型(“declared type”)，它可以用在变量声明中作为变量的数据类型。例如：

```
1    voidf(constvector<int>& a, vector<float>& b)
2    {
3        // 推断表达式 a[0]*b[0]的数据类型，并将其定义为 Temp 类型
4        typedefdecltype(a[0]*b[0]) Tmp;
5        // 使用 Tmp 作为数据类型声明变量，创建对象
6
7        for(inti=0; i < b.size(); ++i) {
8          Tmp* p =newTmp(a[i]*b[i]);
9          // …
10      }
11      // …
12   }
```

这个想法以“typeof”的形式已经在通用语言中流行很久了，但是，现在使用中的 typeof 的实现并没有完成，并有一些兼容性问题，所以新标准将其命名为 decltype。

如果你仅仅是想根据初始化值为一个变量推断合适的数据类型，那么使用 auto 是一个更加简单的选择。当你只有需要推断某个表达式的数据类型，例如某个函数调用表达式的计算结果的数据类型，而不是某个变量的数据类型时，你才真正需要 delctype。

参考

• the C++ draft 7.1.6.2 Simple type specifiers

• [Str02] Bjarne Stroustrup. Draft proposal for “typeof”. C++ reflector message c++std-ext-5364, October 2002. (original suggestion).

• [N1478=03-0061] Jaakko Jarvi, Bjarne Stroustrup, Douglas Gregor, and Jeremy Siek: Decltype and

auto (original proposal).

• [N2343=07-0203] Jaakko Jarvi, Bjarne Stroustrup, and Gabriel Dos Reis: Decltype (revision 7): proposed wording.

控制默认函数——默认或者禁用函数

控制默认函数——默认或者禁用

（译注：我们都知道，在我们没有显式定义类的复制构造函数和赋值操作符的情况下，便编译器会为我们生成默认的复制构造函数和赋值操作符，以内存复制的形式完成对象的复制。虽然这种机制可以为我们节省很多编写复制构造函数和赋值操作符的时间，但是在某些情况下，比如我们不希望对象被复制，这种机制却是多此一举。）

关于类的“禁止复制”，现在可以使用 delete 关键字完美地直接表达：

```
1   classX {
2       // …
3
4       X& operator=(constX&) =delete;  // 禁用类的赋值操作符
5       X(constX&) =delete;
6   };
```

相反地，我们也可以使用 default 关键字，显式地指明我们希望使用默认的复制操作：

```
1   classY {
2       // …
3       // 使用默认的赋值操作符和复制构造函数
4       Y& operator=(constY&) =default;  // 默认的复制操作
5       Y(constY&) =default;
6   };
```

显式地使用 default 关键字声明使用类的默认行为，对于编译器来说明显是多余的，但是对于代码的阅读者来说，使用 default 显式地定义复制操作，则意味着这个复制操作就是一个普通的默认的复制操作。 将默认的操作留给编译器去实现将更加简单，更少的错误发生 ，并且通常会产生更好的目标代码。

“default”关键字可以用在任何的默认函数中，而“delete”则可以用于修饰任何函数。例如，我们可以通过下面的方式排除一个不想要的函数参数类型转换：

```
1   structZ {
2       // …
3
4       Z(longlong);     // 可以通过 long long 初始化
5       // 但是不能使用更短的 long 进行初始化，
6       // 也就不是不允许 long 到 long long 的隐式转型（感谢 abel）
7       Z(long) =delete;
8   };
```

参考：

the C++ draft section ???

[N1717==04-0157] Francis Glassborow and Lois

Goldthwaite:

explicit class and default definitions

(an early proposal).

Bjarne Stroustrup:

Control of class defaults

(a dead end).

[N2326 = 07-0186] Lawrence Crowl:

Defaulted and Deleted Functions

控制默认函数——移动(move)或者复制(copy)

控制默认函数——移动（move）或者复制（copy）

在默认情况下，一个类拥有 5 个默认函数或操作符：

- 复制赋值操作符（ copy assignment ）
- 复制构造函数（ copy constructor ）
- 移动赋值操作符（ move assignment ）
- 移动构造函数（ move constructor ）
- 析构函数（ destructor ）

如果你声明并重新定义了上述 5 个函数或操作符中的任何一个，你必须考虑其余的 4 个，并且显式地定义你需要的操作，或者使用这个操作的默认行为。考虑一下复制，移动和析构，他们是三个密切相关的操作。他们不像那些独立的可以自由地混合和匹配的操作——你可以任意地组合，但是只有少数的组合是有意义的。(?)

如果我们显式地指明（ 声明，定义，=default，或者 =delete ）了移动，复制或者析构函数的行为，将不会产生默认的移动操作（移动赋值操作符和移动构造函数）。同时，未声明的复制操作（复制赋值操作符和复制构造函数）也会被默认生成，但是，这是是应该尽量避免的，不要依赖于编译器的这种行为。

```
1   classX1 {
2       X1& operator=(constX1&) =delete;// 禁用复制操作
3   };
```

这一定义，同时也隐式地禁用了 X1 的移动操作。复制初始化（ Copy initialization ）是被允许，但是这一操作行为是应当尽量避免的。

```
1   classX2 {
2       X2& operator=(constX2&) =default;
3   };
```

这个定义同样隐式地禁用了 X2 的移动操作。复制初始化是被允许，但是这一操作行为是应当尽量避免的。

```
1   classX3 {
2       X3& operator=(X3&&) =delete; //   禁用移动赋值操作符
3   };
```

这一定义也同样隐式地禁用了 X3 的复制操作。

```
1   classX4 {
2       ~X4() =delete;// 禁用析构函数
3   };
```

这个定义同时隐式地禁用了 X4 的移动操作（moving）。复制操作（copying）是被允许的，同样，这一操作也是应当尽量避免的。

如果你声明了上述 5 个默认函数中的任何一个，我强烈建议你显式地声明所有这 5 个默认函数。例如：

```
1    template<classT>
2    classHandle {
3        T* p;
4    public:
5        Handle(T* pp) : p{pp} {}
6        // 用户定义构造函数： 没有隐式的复制和移动操作
7            ~Handle() {deletep; }
8
9        Handle(Handle&& h) :p{h.p}
10           { h.p=nullptr; };// transfer ownership
11       Handle& operator=(Handle&& h)
12           {deletep; p=h.p; h.p=nullptr; }// 传递所有权
13
14       Handle(constHandle&) =delete; // 禁用复制
15       Handle& operator=(constHandle&) =delete;
16
17       // ...
18   };
```

参考：

- the C++

draft section ???

- [N2326==07-0186]

Lawrence Crowl:

- Defaulted and Deleted Functions

.

- [N3174=100164]

B. Stroustrup:

To move or not to move

. An analysis of problems related to generated copy

and move operations. Approved.

委托构造函数（Delegating constructors）

委托构造函数（Delegating constructors）

在 C++98中，如果你想让两个构造函数完成相似的事情，可以实现两个完全相同的构造函数，或者是实现一个 init()函数，两个构造函数都调用这个 init()函数。例如：

```
1    classX {
2            inta;
3            // 实现一个初始化函数
4            validate(intx) {if(0<x && x<=max) a=x;elsethrowbad_X(x); }
5        public:
6            // 三个构造函数都调用初始化函数 validate()，完成初始化工作
7            X(intx) { validate(x); }
8            X() { validate(42); }
9            X(string s) {intx = lexical_cast<int>(s); validate(x); }
10           // …
11       };
```

这样的实现方式重复罗嗦，并且容易出错。并且，这两种方式的可维护性都很差。所以，在 C++0x 中，我们可以在定义一个构造函数时调用另外一个构造函数：

```
1    classX {
2            inta;
3        public:
4            X(intx) {if(0<x && x<=max) a=x;elsethrowbad_X(x); }
5            // 构造函数 X()调用构造函数 X(int x)
6            X() :X{42} { }
```

```
7           // 构造函数 X(string s)调用构造函数 X(int x)
8           X(string s) :X{lexical_cast<int>(s)} { }
9           // …
10      };
```

（译注：在一个构造函数中调用另外一个构造函数，这就是委托的意味，不同的构造函数自己负责处理自己的不同情况，把最基本的构造工作委托给某个基础构造函数完成，实现分工协作。）

参考：

- the C++ draft section 12.6.2
- N1986==06-0056 Herb Sutter and Francis Glassborow: Delegating Constructors (revision 3).

并发性动态初始化和析构

并发性动态初始化和析构

（译注：这部分作者还没有完成，不过一旦英文版出来，中文版将进行同步更新，请读者多多关注！）

参考：

[N2660 = 08-0170] Lawrence Crowl: Dynamic Initialization and Destruction with Concurrency (Final proposal).

（翻译：lianggang jiang）

noexcept – 阻止异常的传播与扩散

noexcept——阻止异常的传播与扩散

如果一个函数不能抛出异常，或者一个程序并没有接获某个函数所抛出的异常并进行处理，那么这个函数

可以用新的 noexcept 关键字对其进行修饰，表示这个函数不会抛出异常或者抛出的异常不会被接获并处理。

例如：

```
1   extern"C"doublesqrt(double) noexcept;// 永远不会抛出异常
2
3       // 在这里，我不准备处理内存耗尽的异常，所以我只是简单地将函数声明为 noexcept
4       vector<double> my_computation(constvector<double>& v) noexcept
5       {
6           vecto<double>r res(v.size()); // 可能会抛出异常
7           for(inti; i<v.size(); ++i) res[i] =sqrt(v[i]);
8           returnres;
9       }
```

如果一个经过 noexcept 修饰的函数抛出异常（异常会尝试逃出这个函数（ ？ ）），程序会通过调用 terminate()来结束执行。通过 terminate()的调用来结束程序的执行会带来很多问题，例如，无法保证对象的析构函数的正常调用，无法保证栈的自动释放，同时也无法在没有遇到任何问题的情况下重新启动程序。所以，它是不可靠的。

我们这样写是故意的，它使得成为一种简单、粗暴但是非常有效的机制（比那种旧的动态地抛出异常的机制要有效得多）。

同时，我们还可以让一个函数根据不同的条件实现 noexcept 修饰或者是无 noexcept 修饰。例如，一个算法可以根据它用作模板参数所使用的操作是否抛出异常，来决定自己是否抛出异常。例如：

```
1       //如果 f(v.at(0))可以抛出异常，则这个函数也可以抛出异常
2       template<classT>
3       voiddo_f(vector<T>& v) noexcept(noexcept(f(v.at(0))))
4   {
5           for(inti; i<v.size(); ++i)
6               v.at(i) = f(v.at(i));
7   }
```

在这里，第一个 noexcept 被用作操作符（operator）：如果 if f(v.at(0))不能够抛出异常，noexcept(f(v.at(0)))则返回 true，也即意味着 f()和 at()是无法抛出异常（noexcept）。

noexcept()操作符是一个常量表达式，并且不计算表达式的值，只是判断这个表达式是否会产生并抛出异常。

声明的通常形式是 noexcept(expression)，并且单独的一个“noexcept”关键字实际上就是的一个noexcept(true)的简化。一个函数的所有声明都必须与 noexcept 声明保持 兼容。

一个析构函数不应该抛出异常；通常，如果一个类的所有成员都拥有 noexcept 修饰的析构函数，那么这个类的析构函数就自动地隐式地 noexcept 声明，而与函数体内的代码没有关系。

通常，将某个抛出的异常进行移动操作是一个很坏的主意，所以，在任何可能的地方都用 noexcept 进行声明。

如果某个类的所有成员都有使用 noexcept 声明的析构函数，那么这个类默认生成的复制或者移动操作（类的复制构造函数，移动构造函数等）都是隐式的 noexcept 声明。（？）

noexcept 被广泛地系统地应用在 C++11的标准库中 ,以此来提供标准库的性能和满足标准库对于简洁性的需求。

参考

Standard: 15.4 Exception specifications [except.spec].

Standard: 5.3.7 noexcept operator [expr.unary.noexcept].

[N3103==10-0093] D. Kohlbrenner, D. Svoboda, and A. Wesie: Security impact of noexcept. (Noexcept must terminate, as it does).

[N3167==10-0157] David Svoboda: Delete operators default to noexcept.

[N3204==10-0194] Jens Maurer: Deducing “noexcept” for destructors

[N3050==10-0040] D. Abrahams, R. Sharoni, and D. Gregor: Allowing Move Constructors to Throw (Rev. 1).

显式转换操作符

C++98标准提供隐式和显式两种构造函数，也就是说，声明为显式形式的构造函数所定义的转换只能用于

显式转换，而其他形式的构造函数则用于隐式转换。例如：

```
structS { S(int); };   // “普通构造函数”表明是隐式转换
S s1(1);       // 正确
S s2 = 1;   // 正确
voidf(S);
//正确（但是经常会产生意外结果——如果 S 是 vector 类型会怎么样呢？）(?)
f(1);

structE {explicitE(int); };   // 显式构造函数
E e1(1);       // 正确
E e2 = 1;   // 错误（但是常常会让人感到意外——这怎么会错呢？）
voidf(E);
// 错误（不一定会产生意外
// 例如从 int 型到 std::vector 类型转换的构造函数是显式的）(?)
f(1);
```

然而，构造函数不是定义转换的唯一途径。如果不能更改一个类，那么可以从另一个不同的类中定义一个转换操作符。例如：

```
structS { S(int) { }/* … */};

structSS {
    intm;
    SS(intx) :m(x) { }
    // 因为结构体 S 中没有 S(SS)；不存在干扰
    operator S() {returnS(m); }
};

SS ss(1);
S s1 = ss;   // 正确；类似隐式构造函数
S s2(ss);   // 正确；类似隐式构造函数
voidf(S);
f(ss);       // 正确；类似隐式构造函数
```

（译注：这段代码的意义，实际是通过 SS 作为中间桥梁，将 int 转换为 S。）

遗憾的是，这里并没有显式转换操作符（因为有问题的例子很少）。C++11通过允许转换操作符成为显式形式而弥补了这个漏失(?)。例如：

```
structS { S(int) { } };

structSS {
```

```
4        intm;
5        SS(intx) :m(x) { }
6
7        // 因为结构体 S 中没有 S(SS)（译注：因为结构体 S 中没有定义 S(SS)，
8        // 无法将 SS 转换为 S，所以只好在 SS 中定义一个返回 S 的转换操作符，
9        // 将自己转换为 S。转换动作，可以由目标类型 S 提供，也可以由源类型 SS 提供。）
10        explicitoperator S() {returnS(m); }
11   };
12
13   SS ss(1);
14   S s1 = ss;   // 错误；类似显式构造函数
15   S s2(ss);   // 正确；类似显式构造函数
16   voidf(S);
17   f(ss);       // 错误；类似显式构造函数
```

参考:

Standard: 12.3 Conversions

[N2333=07-0193] Lois Goldthwaite, Michael Wong, and Jens Maurer:

Explicit Conversion Operator (Revision 1).

扩展整型

如果扩展整型存在，将有一整套的规则说明应该如何进行（准确的）整型扩展。

参见：

[06-0058==N1988] J. Stephen Adamczyk: Adding extended integer types to C++ (Revision 1) .

（翻译：lianggang jiang）

外部模板声明

限定模板可以被显式声明，这可以作为消除多重实例化的一种方式。例如：

```
1   #include "MyVector.h"
2   externtemplateclassMyVector<int>;// 消除下面的隐式实例化——
3   // MyVector 类将在“其他地方”显式的实例化
4   voidfoo(MyVector<int>& v)
5   {
6       //在这个地方使用 vector 类型
7   }
```

这个”其他地方”可能这样出现：

```
1   #include "MyVector.h"
2   // 使 MyVector 类对客户端（clients）可用（例如，共享库）
3   templateclassMyVector<int>;
```

这是编译器和链接器避免大量冗余工作的一种主要方式。

参见：

Standard 14.7.2 Explicit instantiation

[N1448==03-0031] Mat Marcus and Gabriel Dos Reis: Controlling Implicit Template Instantiation

.

（翻译：lianggang jiang）

序列 for 循环语句

序列 for 循环语句允许重复遍历一组序列，而这组序列可以是任何可以重复遍历的序列，如由 begin()和 end()函数定义的 STL 序列。所有的标准容器都可用作这种序列，同时它也同样可以是 std::string，初始化列表（list），数组，以及任何由 begin()和 end()函数定义的序列，例如输入流。这里是一个序列 for 循环语句的例子：

```
1   voidf(constvector& v)
2   {
3       for(auto x : v) cout << x <<  ‘n’ ;
4       for(auto& x : v) ++x;   // 使用引用，方便我们修改容器中的数据
5   }
```

可以这样理解这里的序列 for 循环语句，“对于 v 中的所有数据元素 x” ,循环由 v.begin()开始 ,循环到 v.end()结束。又如：

```
1   for(constauto x : { 1,2,3,5,8,13,21,34 })
2      cout << x <<  ‘n’ ;
```

begin()函数（包括 end()函数）可以是成员函数通过 x.begin()方式调用，或者是独立函数通过 begin(x)方式调用。

（译注：好像 C#中早就有这种形式的 for 循环语句，用于遍历一个容器中的所有数据很方便，难道 C++是从 C#中借用过来的？）

或参见：

the C++ draft section 6.5.4 (note: changed not to use concepts)

[N2243==07-0103] Thorsten Ottosen:

Wording for range-based for-loop (revision 2).

[N3257=11-0027] Jonathan Wakely and Bjarne Stroustrup: Range-based for statements and ADL (Option 5 was chosen).

返回值类型后置语法

返回类型后置语法

考虑下面这段代码：

```
1   template<classT,classU>
```

```
2    ??? mul(T x, U y)
3    {
4        returnx*y;
5    }
```

函数 mul()的返回类型要怎么写呢？当然，是“x*y 类型”，但是这并不是一个数据类型，我们如何才能一开始就得到它的真实数据类型呢？在初步了解 C++0x 之后，你可能一开始想到使用 decltype 来推断“x*y”的数据类型：

```
1    template<classT,classU>
2    decltype(x*y) mul(T x, U y)// 注意这里的作用域
3    {
4        returnx*y;
5    }
```

但是，这种方式是行不通的，因为 x 和 y 不在作用域内。但是，我们可以这样写：

```
1    template<classT,classU>
2    // 难看别扭，且容易产生错误
3    decltype(*(T*)(0)**(U*)(0)) mul(T x, U y)
4    {
5        returnx*y;
6    }
```

如果称这种用法为“还可以”，就已经是过誉了。

C++11的解决办法是将返回类型放在它所属的函数名的后面：

```
1    template<classT,classU>
2    auto mul(T x, U y) -> decltype(x*y)
3    {
4        returnx*y;
5    }
```

这里我们使用了 auto 关键字，(auto 在 C++11中还有根据初始值推断数据类型的意义)，在这里它的意思变为“返回类型将会稍后引出或指定”。

返回值后置语法最初并不是用于模板和返回值类型缩减的，它实际是用于解决作用域问题的。

```
1    structList {
2        structLink {/* ... */};
3        Link* erase(Link* p);  // 移除 p 并返回 p 之前的链接
4        // ...
```

```
5    };
6
7    List::Link* List::erase(Link* p) {/* ... */}
```

第一个 List::是必需的，这仅是因为 List 的作用域直到第二个 List::才有效。更好的表示方式是：

```
1    auto List::erase(Link* p) -> Link* {/* ... */}
```

现在，将函数返回类型后置，Link*就不需要使用明确的 List::进行限定了。

参考：

the C++ draft section ???

[Str02] Bjarne Stroustrup. Draft proposal for “typeof”. C++ reflector message c++std-ext-5364, October 2002.

[N1478=03-0061] Jaakko Jarvi, Bjarne Stroustrup, Douglas Gregor, and Jeremy Siek: Decltype and auto.

[N2445=07-0315] Jason Merrill: New Function Declarator Syntax Wording.

[N2825=09-0015] Lawrence Crowl and Alisdair Meredith: Unified Function Syntax.

类成员的内部初始化

类内部成员的初始化

在 C++98标准里 ,只有 static const 声明的整型成员能在类内部初始化 ,并且初始化值必须是常量表达式。这些限制确保了初始化操作可以在编译时期进行。例如：

```
1    intvar = 7;
2    classX {
```

```
3       staticconstintm1 = 7;  // 正确
4       constintm2 = 7;   // 错误：无 static
5       staticintm3 = 7;            // 错误：无 const
6       staticconstintm4 = var; // 错误：初始化值不是常量表达式
7       staticconststring m5 = “odd”;//错误：非整型
8       // …
9   };
```

C++11的基本思想是，允许非静态（non-static）数据成员在其声明处（在其所属类内部）进行初始化。这样，在运行过程中，需要初始值时构造函数可以使用这个初始值。考虑下面的代码：

```
1   classA {
2   public:
3       inta = 7;
4   };
```

这等同于：

```
1   classA {
2   public:
3       inta;
4       A() : a(7) {}
5   };
```

从代码来看，这样可以省去一些文字的输入，但是真正受益的是拥有多个构造函数的类。在大多数情况下，所有的构造函数都会使用成员变量的常见初始值：

```
1    classA {
2        public:
3            A(): a(7), b(5), hash_algorithm(“MD5″), s(“Constructor run”) {}
4           A(inta_val) :
5             a(a_val), b(5), hash_algorithm(“MD5″),
6             s(“Constructor run”)
7             {}
8
9           A(D d) : a(7), b(g(d)),
10              hash_algorithm(“MD5″), s(“Constructor run”)
11              {}
12          inta, b;
13      private:
14          // 哈希加密函数可应用于类 A 的所有实例
15          HashingFunction hash_algorithm;
16          std::string s; // 指示的字符串变量将持续整个对象的生命周期
17      };
```

hash_algorithm 和 s 每个都有一个单独的默认值的事实会由于杂乱的代码而不明显，这会在程序维护时造成麻烦。作为替代，可以将数据成员的初始值提取出来：

```
1    classA {
2        public:
3            A(): a(7), b(5) {}
4            A(inta_val) : a(a_val), b(5) {}
5            A(D d) : a(7), b(g(d)) {}
6            inta, b;
7        private:
8            //哈希加密函数可应用于类 A 的所有实例
9            HashingFunction hash_algorithm{“MD5″};
10           //指示的字符串变量将持续整个对象的生命周期
11           std::string s{“Constructor run”};
12       };
```

如果一个成员同时在类内部初始化时和构造函数内被初始化 则只有构造函数的初始化有效(这个初始化值“优先于”默认值)(译注：可以认为，类内部初始化先于构造函数初始化进行，如果是对同一个变量进行初始化，构造函数初始化会覆盖类内部初始化)。因此，我们可以进一步简化：

```
1    classA {
2        public:
3            A() {}
4            A(inta_val) : a(a_val) {}
5            A(D d) : b(g(d)) {}
6            inta = 7;
7            intb = 5;
8        private:
9            //哈希加密函数可应用于类 A 的所有实例
10           HashingFunction hash_algorithm{“MD5″};
11           //指示的字符串变量将持续整个对象的生命周期
12           std::string s{“Constructor run”};
13   };
```

或参见：

the C++ draft section “one or two words all over the place”; see proposal.

[N2628=08-0138] Michael Spertus and Bill Seymour:

Non-static data member initializers.

（翻译：lianggang jiang）

继承的构造函数（inherited constructors）

人们有时会对类成员函数或成员变量的作用域问题感到困惑，尤其是，当基类与派生类的同名成员不在同一个作用域内时：

```
1    structB {
2            voidf(double);
3        };
4        structD : B {
5            voidf(int);
6        };
7        B b;   b.f(4.5);   // 可行
8        // 调用的到底是 B::f(doube)还是 D::f(int)呢？
9        // 实际情况往往会让冉感到意外：调用的 f(int)函数实参为4
10       D d;   d.f(4.5);
```

在 C++98标准里，可以将普通的重载函数从基类“晋级”到派生类里来解决这个问题：

```
1    structB {
2            voidf(double);
3        };
4
5        structD : B {
6            usingB::f;    // 将类 B 中的 f()函数引入到类 D 的作用域内
7            voidf(int);   // 增加一个新的 f()函数
8        };
9
10       B b;   b.f(4.5);   // 可行
11       // 可行：调用类 D 中的 f(double)函数
12       // 也即类 B 中的 f(double)函数
13       D d;   d.f(4.5);
```

普通重载函数可以通过这种方式解决，那么，对于构造函数又该怎么办呢？ 我已经说过“不能像应用于普通成员函数那样，将上述语法应用于构造函数，这如历史偶然一样”。为了解决构造函数的“晋级”问题，C++11 提供了这种能力：

```
classDerived :publicBase {
    public:
        // 提升 Base 类的 f 函数到 Derived 类的作用范围内
        // 这一特性已存在于 C++98标准内
        usingBase::f;
        voidf(char);    // 提供一个新的 f 函数
        voidf(int); // 与 Base 类的 f(int)函数相比更常用到这个 f 函数

        // 提升 Base 类的构造函数到 Derived 的作用范围内
        // 这一特性只存在于 C++11标准内
        usingBase::Base;
        Derived(char);   // 提供一个新的构造函数
        // 与 Base 类的构造函数 Base(int)相比
        // 更常用到这个构造函数
        Derived(int);
        // …
    };
```

如果这样用了，仍然可能困惑于派生类中继承的构造函数，这个派生类中定义的新成员变量需要初始化（译注：

基类并不知道派生类的新增成员变量，当然不会对其进行初始化。）：

```
structB1 {
        B1(int) { }
    };
    structD1 : B1 {
        usingB1::B1;// 隐式声明构造函数 D1(int)
        intx;
    };
    voidtest()
    {
        D1 d(6); // 糟糕：调用的是基类的构造函数，d.x 没有初始化
        D1 e;   // 错误：类 D1没有默认的构造函数
}
```

我们可以通过使用成员初始化（member-initializer）消除以上的困惑：

```
structD1 : B1 {
            usingB1::B1;   // 隐式声明构造函数 D1(int)
            // 注意：x 变量已经被初始化
            // （译注：在声明的时候就提供初始化）
            intx{0};
        };
        voidtest()
        {
```

```
9               D1 d(6);   // d.x 的值是0
10          }
```

或参见：

the C++ draft 8.5.4 List-initialization [dcl.init.list]

[N1890=05-0150] Bjarne Stroustrup and Gabriel Dos Reis:

Initialization and initializers

(an overview of initialization-related problems with suggested solutions).

[N1919=05-0179] Bjarne Stroustrup and Gabriel Dos Reis:

Initializer lists.

[N2215=07-0075] Bjarne Stroustrup and Gabriel Dos Reis :

Initializer lists (Rev. 3) .

[N2640=08-0150] Jason Merrill and Daveed Vandevoorde:

Initializer Lists — Alternative Mechanism and Rationale (v. 2) (final proposal).

初始化列表

考虑如下代码：

```
1    vector<double> v = { 1, 2, 3.456, 99.99 };
2        list<pair<string,string>> languages = {
3            {"Nygaard","Simula"}, {"Richards","BCPL"}, {"Ritchie","C"}
4        };
5        map<vector<string>,vector<int>> years = {
6            { {"Maurice","Vincent","Wilkes"},
7                    {1913, 1945, 1951, 1967, 2000} },
8            { {"Martin","Ritchards"},
9                    {1982, 2003, 2007} },
10           { {"David","John","Wheeler"},
11                   {1927, 1947, 1951, 2004} }
12       };
```

现在，初始化列表不再仅限于数组。可以接受一个“{}列表”对变量进行初始化的机制实际上是通过一个可以接受参数类型为 std::initializer_list 的函数（通常为构造函数）来实现的。例如：

```
1   voidf(initializer_list<int>);
2       f({1,2});
3       f({23,345,4567,56789});
4       f({}); // 以空列表为参数调用 f()
5       f{1,2};// 错误：缺少函数调用符号( )
6
7       years.insert({{"Bjarne","Stroustrup"},{1950, 1975, 1985}});
```

初始化列表可以是任意长度，但必须是同质的（所有的元素必须属于某一模板类型 T，或可转化至 T 类型的)。

一个容器可以以如下方式来实现一个初始化列表：

```
1    template<classE>classvector {
2        public:
3            // 初始化列表构造函数
4                    vector (std::initializer_list<E> s)
5            {
6                   // 获取正确的容器中数据的数量
7                     reserve(s.size());   //
8                // 初始化所有元素
9                    uninitialized_copy(s.begin(), s.end(), elem);
10            sz = s.size();// 设置容器的尺寸（size）
11           }
12
13           // ... 与之前相同 ...
14       };
```

“{}初始化”出现之后，直接初始化与复制初始化之间的差异依然存在， 但由于“{}初始化”的存在，这种差异变得不再那么频繁出现。例如，std::vector 拥有一个参数类型为 int 的显式构造函数及一个带有初始化列表的构造函数：

```
1    vector<double> v1(7);// 正确: v1有7个元素
2        v1 = 9;        // 错误: 无法将 int 转换为 vector
3        vector<double> v2 = 9;   // 错误: 无法将 int 转换为 vector
4
5        voidf(constvector<double>&);
6        f(9);              // 错误: 无法将 int 转换为 vector
7
```

```
8       vector<double> v1{7};    // 正确: v1有一个元素，其值为7
9       v1 = {9};          // 正确，v1现在有一个值为9的元素
10      vector<double> v2 = {9}; // 正确: v2有一个值为9的元素
11      f({9}); // 正确: f 函数将以{ 9 }为参数被调用
12
13      vector<vector<double>> vs = {
14      // 正确: 显式构造（10个元素，值为 double 的默认值）
15          vector<double>(10),
16          vector<double>{10},  // 正确：显式构造（1个元素，值为10）
17          10         //错误 ： vector 的构造函数是显式的
18      };
```

函数可以将 initializer_list 作为一个不可变的序列进行存取。例如：

```
1   voidf(initializer_list<int> args)
2       {
3           for(auto p=args.begin(); p!=args.end(); ++p) cout << *p <<"\n";
4       }
```

仅具有一个 std::initializer_list 的单参数构造函数被称为初始化列表构造函数。

标准库容器，string 类型及正则表达式均具有初始化列表构造函数，以及（初始化列表）赋值函数等。一个初始化列表可被用作 Range，例如，表达式 Range。

初始化列表是一致泛化初始化解决方案的一部分。

参考：

the C++ draft 8.5.4 List-initialization [dcl.init.list]

[N1890=05-0150] Bjarne Stroustrup and Gabriel Dos Reis:

Initialization and initializers

(an overview of initialization-related problems with suggested solutions).

[N1919=05-0179] Bjarne Stroustrup and Gabriel Dos Reis:

Initializer lists.

[N2215=07-0075] Bjarne Stroustrup and Gabriel Dos Reis :

Initializer lists (Rev. 3) .

[N2640=08-0150] Jason Merrill and Daveed Vandevoorde:

Initializer Lists — Alternative Mechanism and Rationale (v. 2) (final proposal).

（翻译：dabaitu）

内联命名空间(inline namespace)

内联命名空间机制是通过提供一种支持版本更新的机制，来支持库的演化。考虑如下代码：

```
1    // 文件：V99.h
2     inlinenamespaceV99 {
3            voidf(int);       // 做一些比 V98版本更好的改进
4            voidf(double);       // 新的特性
5            // …
6     }
7
8     // 文件：V98.h
9     namespaceV98 {
10        voidf(int);       // 做一些事情
11        // …
12    }
13
14    // 文件：Mine.h
15    namespaceMine {
16    #include “V99.h”
17    #include “V98.h”
18    }
```

现在，我们有一个命名空间 Mine，它包含了较新的发行版本(V99)以及早期的版本(V98)，如果你需要显式应用（某个版本的函数），你可以：

```
1    #include “Mine.h”
2    usingnamespaceMine;
3    // …
4    V98::f(1);       // 早期版本
5    V99::f(1);       // 较新的版本
```

```
6    f(1);            // 默认的版本
```

此处的要点在于，inline 描述符使得内联命名空间中的声明看起来就好像是直接在外围的命名空间中进行声明的一样。（译注：我们注意到，这里的 f(1)函数调用并没有使用 V99::f(1)，使用 inline 关键字定义的内联名字空间成为默认名字空间。 < ![endif]–>就像内联函数一样，内联的名字空间被嵌入到它的外围名字空间，成为外围名字空间的一部分。 ）

inline 描述符是一个非常“静态(static)”及面向实现的设施，它由命名空间的设计者放置，即帮助所有的用于进行了选择（译注：即命名空间的作者可以通过放置 inline 描述符来表示当前最新的命名空间是哪个，所以对用户来说，这个选择是“静态”的：用户无权判断哪个命名空间是最新的）。因此，不会存在有 Mine 命名空间的用户说“我想要默认的命名空间为 V98，而非 V99”。

参考：

Standard 7.3.1 Namespace definition [7]-[9].

（翻译：dabaitu）

Lambdas

Lambdas

（译注:目前支持 lambda 的 gcc 编译器版本为4.5，其它详细的编译器对于 C++11新特性的支持请参考 http://wiki.apache.org/stdcxx/C%2B%2B0xCompilerSupport）

Lambda 表达式是一种描述函数对象的机制，它的主要应用是描述某些具有简单行为的函数（译注：Lambda 表达式也可以称为匿名函数，具有复杂行为的函数可以采用命名函数对象，当然,简单和复杂之间的划分依赖于编程人员的选择）。例如：

```
1    vector<int> v = {50, -10, 20, -30};
```

```
2   std::sort(v.begin(), v.end());   // 采用默认排序
3   // 现在 v 中的数据应该是 { -30, -10, 20, 50 }
4
5   // 利用 Lambda 表达式，按照绝对值排序
6   std::sort(v.begin(), v.end(), [](inta,intb) {returnabs(a)<abs(b); });
7   // 现在 v 应该是 { -10, 20, -30, 50 }
```

参数 [&](int a, int b) { return abs(a) < abs(b); }是一个"lambda"(又称为"lambda 函数"或者"lambda 表达式"), 它描述了这样一个函数操作 : 接受两个整形参数 a 和 b , 然后返回对它们的绝对值进行"<"比较的结果。(译注：为了保持与代码的一致性，此处应当为"[] (int a, int b) { return abs(a) < abs(b); }"，而且在这个 lambda 表达式内实际上未用到局部变量，所以 [&] 是无必要的)

一个 Lambda 表达式可以存取在它被调用的作用域内的局部变量。例如：

```
1       voidf(vector<Record>& v)
2   {
3       vector<int> indices(v.size());
4       intcount = 0;
5       generate(indices.begin(),indices.end(),[&count](){returncount++; });
6
7       // sort indices in the order determined by the name field of the records:
8       std::sort(indices.begin(), indices.end(), [&](inta,intb) {returnv[a].name<v[b].name; }
9       // …
10  }
```

有人认为这“相当简洁” , 也有人认为这是一种可能产生危险且晦涩的代码的方式。我的看法是 , 两者都正确。

[&] 是一个“捕捉列表(capture list)” , 用于描述将要被 lambda 函数以引用传参方式使用的局部变量。如果我们仅想“捕捉"参数 v , 则可以写为: [&v]。而如果我们想以传值方式使用参数 v , 则可以写为 : [=v]。如果什么都不捕捉 , 则为 : []。将所有的变量以引用传递方式使用时采用 [&], [=] 则相应地表示以传值方式使用所有变量。(译注 : “所有变量”即指 lambda 表达式在被调用处 , 所能见到的所有局部变量)

如果某一函数的行为既不通用也不简单，那么我建议采用命名函数对象或者函数。例如，如上示例可重写为：

```
1   voidf( vector<Record>& v)
2   {
3           vector<int> indices(v.size() );
4           intcount = 0;
```

```
5           fill(indices.begin(), indices.end(), [&]()
6           {return++count; };
7
8           structCmp_names {
9                   constvector& vr;
10                  Cmp_names(constvector<Record>& r) : vr(r) {}
11                  (译注：原文此处为"Comp_names"，疑是笔误)
12                  booloperator() (Record& a, Record& b)const
13          {returnvr[a] < vr[b]; }
14          };
15
16          //对 indices 按照记录的名字域顺序进行排序
17          std::sort(indices.begin(), indices.end(), Cmp_names(v) );
18  }
```

(译注：此处采用了函数对象 Cmp_names(v)来代替 lambda 表达式，由于 Cmp_names 具有以引用传参方式的构造函数，因此 Cmp_names(v)相当于使用了"[&v]"的 lambda 表达式)

对于简单的函数功能，比如记录名称域的比较，采用函数对象就略显冗长，尽管它与 lambda 表达式生成的代码是一致的。在 C++98中，这样的函数对象在被用作模板参数时必须是非本地的（译注：即你不能在函数对象中像此处的 lambda 表达式那样使用被调用处的局部变量），然而在 C++中（译注：意指 C++0x），这不再是必须的。

为了描述一个 lambda，你必须提供：

它的捕捉列表：它可以使用的变量列表（除了形参之外），如果存在的话（"[&]" 在上面的记录比较例子中意味着“所有的局部变量都将按照引用的方式进行传递”）。如果不需要捕捉任何变量，则使用 []。

（可选的）它的所有参数及其类型（例如： (int a, int b) ）。

组织成一个块的函数行为（例如：{ return v[a].name < v[b].name; }）。

（可选的）采用了新的后缀返回类型符号的返回类型。但典型情况下，我们仅从 return 语句中去推断返回类型，如果没有返回任何值，则推断为 void。

参考：

Standard 5.1.2 Lambda expressions

[N1968=06-0038] Jeremiah Willcock, Jaakko Jarvi, Doug Gregor, Bjarne Stroustrup, and Andrew Lumsdaine:

Lambda expressions and closures for C++

(original proposal with a different syntax)

[N2550=08-0060] Jaakko Jarvi, John Freeman, and Lawrence Crowl:

Lambda Expressions and Closures: Wording for Monomorphic Lambdas (Revision 4) (final proposal).

[N2859=09-0049] Daveed Vandevoorde:

New wording for C++0x Lambdas.

用作模板参数的局部类型

用作模板参数的局部类型

在 C++98中，局部和未命名类型不能作为模板参数，这或许是一个负担，C++11则放宽了这方面的限制：

```
1   voidf(vector<X>& v)
2       {
3           structLess {
4               booloperator()(constX& a,constX& b) {returna.v<b.v; }
5           };
6           // C++98: 错误: Less 是一个本地的类
7             // C++11: 正确
8             sort(v.begin(), v.end(), Less());
9       }
```

在 C++11中，我们还可以采用 Lambda 表达式：

```
1   voidf(vector<X>& v)
2       {
```

```
3           sort(v.begin(), v.end(),
4                [] (constX& a,constX& b) {returna.v<b.v; });// C++11
5       }
```

值得注意的是，为操作行为命名有利于文档化，是一个值得鼓励的设计风格。同时，非局部的函数体（当然也需要命名）还可以被重用。

C++11同时也允许模板参数使用未命名类型的值：

```
1    template<typenameT>voidfoo(Tconst& t){}
2        enumX { x };
3        enum{ y };
4
5        intmain()
6        {
7            foo(x);    // C++98: ok; C++11: ok
8            //（译注：y 是未命名类型的值，C++98无法从这样的值中推断出函数模板参数）
9                   foo(y);    // C++98: error; C++11: ok
10           enumZ { z };
11           foo(z);    // C++98: error; C++11: ok
12                  //（译注：虽然 z 是命名类型的值，
13                  // 但 z 也是局部类型的值，
14                  // 而 C++98不支持从局部类型值推导模板参数）
15       }
```

参考：

Standard: Not yet: CWG issue 757

[N2402=07-0262] Anthony Williams:

Names, Linkage, and Templates (rev 2).

[N2657] John Spicer:

Local and Unnamed Types as Template Arguments.

long long（长长整数类型）

long long（长长整数类型）

这是一个至少为64 bit 的整数类型（译注：实际宽度依赖于具体的实现平台），例如：

```
1    longlongx = 922337203685477580 7LL;
```

不过，不要想当然地认为存在 long long long 或者将 long 拼写为 short long long。

（译注：如同 J. Stephen Adamczyk 在参考文献中所言，“ long long”是一个晦涩的拼写64-bit 整数类型的方式，也不是一个可以解决不断增长的数据类型宽度的有效方法，目前，它仅仅是一个用于表达64bit 整形数的标准。）

参考：

the C++ draft ???.

[05-0071==N1811] J. Stephen Adamczyk:

Adding the long long type to C++ (Revision 3).

内存模型

内存模型

（译注：这一个 item 有相当深的理论深度，原文也比较晦涩难懂，翻译者提醒大家，最好参照原文理解，如果翻译中有什么不恰当的地方，还请批评指出，不胜感谢。）

内存模型是计算机体系结构与编译器作者之间的协议，它使得大多数程序员不用去考虑现代计算机硬件结构的细节。如果没有内存模型，线程机制，锁机制及无锁编程等都将失去意义。

内存模型的最关键保证是：两个运行的线程可以独立地存取和更新各自的内存位置而不会互相影响。那么，什么是“内存位置”呢？ 一个内存位置要么是一个标量类型的对象 ,要么是一个最长连续的具有非零位宽的比特域的序列。例如：此处的 S 具有四个独立的内存位置：

```
1    structS {
```

```
2        chara;          // 位置#1
3        intb:5,          // 位置#2
4        intc:11,
5        // 注意："":0"是一个“特殊符号”
6        //（译注：此处的":0"会将一个 int 对象分割为两个内存位置，
7        // 也就是上面所讲的内存位置的第二种情况）
8        int:0,
9        intd :8;          // 位置#3
10       struct{intee:8; } e;   // 位置#4
11   };
```

内存模型为什么如此重要？为什么它不是显而易见的？(译注 :此处指“为什么应用程序所使用的内存模型与计算机的物理内存结构不是直接一致的，即不用经过任何中间层”，在本例中，应当指为什么 int 不是一个内存位置，而要被分割为两个部分。）难道并非一直如此吗？（译注：此处指“从计算机内存被使用以来，内存模型就与物理内存不一致吗？”（？））。

问题在于，如果多个计算任务真正地并行运行，那么当这些来自不同计算任务中不相关（很明显）的指令代码在同一时刻执行时，内存硬件的诡异行为将会被暴露无遗（译注：一个诡异行为就是，在上述例子中，对变量 b,c,d 的存取，并非一次仅操作5,11或者8个 bit。具体操作与处理器的行为有关）。事实上，如果没有了编译器的支持，应用程序开发人员将完全无法管理指令和数据的流水线安排以及缓存应用问题。即使不存在两个线程之间的数据共享，情况依然糟糕如此！考虑如下两个分开编译的“线程”：

```
1   //线程 1:
2   charc;
3   c = 1;
4   intx = c;
5
6   //线程 2
7   charb;
8   b = 1;
9   inty = b;
```

为了尽量模拟现实状态，我使用了分开编译（对每个线程）来保证编译器/优化器不会对内存进行优化（译注：此处指对每个线程内的代码进行逐行编译，然后连接，以避免代码优化），如去除变量 c 和 b 而直接将 x 和 y 初始化为1。那么现在，x 和 y 的值可能是什么呢？按照 C++11的标准，唯一正确的答案，也是显而易见的那

个，x 和 y 均为1。但如果你采用了一个传统意义上的优秀的带预并发处理(pre-concurrency)的 C 或者 C++ 编译器，那么事情变得有趣起来：x 和 y 的可能值会是0和0，或者1和0，或者0 和1，或者1和1。这些都是“在自然环境下”（译注：指在现实的代码运行环境中）已经观察到的情况。为何如此呢？原因在于，一个链接器可能会将变量 c 和 b 的位置分配在相邻的内存上（在同一个 word 内），而且 C 和 C++ 1990s 的标准并未对此作出说明（译注：指不拒绝这种变量在内存中的存储位置安排方式）。在这种情况下，C++就如同所有那些未考虑实际硬件并发就进行设计的语言一样，由于现代处理器无法读写单个的字符，读写操作在处理中总是以 word 为单位进行的，所以，对变量 c 的赋值实际上是“读取包含变量 c 的一个 word，替换掉其中的 c 部分，然后再写回这个 word”。由于对变量 b 的赋值也是如此进行的，这就造成了两个操作变量 b 和变量 c 的线程有如此多地机会互相干涉、影响（译注：线程1操作变量 c 时，回写操作会改变变量 b 的值，线程2也是如此，这就造成了两个线程之间的干扰，而 b 和 c 也处于不稳定状态），即使它们之间并未共享数据（由它们的源代码来看，的确如此）。

因此，C++11保证了在“独立的内存位置”上，肯定不会发生如上这种问题。注意到一个单 word 内的不同比特域并非属于独立的内存位置（译注：即它们总是被一起读取和写入的），那么在未采用某种形式的线程锁机制之前，不要在线程之间共享带有比特域的结构。除过这个需要特别注意的地方外，C++的内存模型就如同“大家所期待的那样”，简单而且纯洁  （译注：指对象结构的内存模型就按照结构中的声明顺序进行排列，而且不会发生干扰问题）。

但是，对于底层的并行计算问题，要想直接地进行考虑，并不总是那么简单。考虑下面的代码：

```
1   // 开始的时候，  x==0 ，y==0
2
3   if(x) y = 1;  // 线程 1
4
5   if(y) x = 1;  // 线程 2
```

这里是不是有一个问题？或者更直接地，这里是不是有一个资源竞争？（没有，这里没有资源竞争）

（译注：这里也许跟 C++11中的原子操作相关，可以查阅相关资料）

幸运的是，我们已经适应了当前时代，每一个现代的 C++编译器（我所知道的）都给予上面问题正确的答案，而且它们这些年来也是如此做的。毕竟，C++将“永远”被用于并发系统的关键系统编程。

参考：

Standard: 1.7 The C++ memory model [intro.memory]

Paul E. McKenney, Hans-J. Boehm, and Lawrence Crowl:

C++ Data-Dependency Ordering: Atomics and Memory Model.

N2556==08-0066.

Hans-J. Boehm:

Threads basics,

HPL technical report 2009-259.

“what every programmer should know about memory model issues.”

Hans-J. Boehm and Paul McKenney:

A slightly dated FAQ on C++ memory model issues.

move 语义（参见右值引用）

参见右值引用

预防窄转换

预防窄转换

（译注：“窄转换”是我见到过的一个翻译术语，但我忘记是在那本书上看到的。此处也可译为“预防类型截断”或者“预防类型切割”。）

问题：C 和 C++会进行隐式的（类型）截断

```
1   intx = 7.3;   // 啊哦!
2    voidf(int);
3    f(7.3);       // 啊哦!
```

但是，在 C++11中，使用{}进行初始化不会发生这种窄转换（译注：也就是使用{}对变量进行初始化时，不会进行隐式的类型截断，编译器会产生一个编译错误，防止隐式的类型截断的发生。）：

```
1   intx0 {7.3};   // 错误：窄转换
2   intx1 = {7.3};   //错误：窄转换
3   doubled = 7;
4   intx2{d};   //错误：窄转换（double 类型转化为 int 类型）
5   charx3{7}; //OK：虽然7是一个 int 类型，但这不是窄转换
6   vector vi = {1, 2.3, 4, 5.6}; //错误：double 至 int 到窄转换
```

C++11避免许多不兼容性的方法是在进行一个窄转换时尽可能地依赖于初始化变量的实际值（如上例中的7），而非仅仅使用变量的类型声明。如果一个值可以被精确地表达为目标类型，那么就不存在窄转换。

```
1   // OK: 7是一个 int 类型的数据，但是它可以被准确地表达为 char 类型数据
2   charc1{7};
3
4   // error: 发生了窄转换，初始值超出了 char 类型的范围
5   charc2{77777};
```

请注意，double 至 int 类型的转换通常都会被认为是窄转换，即使从7.0转换至7。

（评注：“{}初始化”对于类型转换的处理增强了 C++静态类型系统的安全性。传统的 C/C++中依赖于编程人员的初始化类型安全检查在 C++11中通过“{}初始化”由编译器实施。）

参考：

the C++ draft 8.5.4 List-initialization [dcl.init.list]

[N1890=05-0150] Bjarne Stroustrup and Gabriel Dos Reis:

Initialization and initializers

(an overview of initialization-related problems with suggested solutions).

[N1919=05-0179] Bjarne Stroustrup and Gabriel Dos Reis:

Initializer lists.

[N2215=07-0075] Bjarne Stroustrup and Gabriel Dos Reis :

Initializer lists (Rev. 3) .

[N2640=08-0150] Jason Merrill and Daveed Vandevoorde:

Initializer Lists — Alternative Mechanism and Rationale (v. 2) (final proposal).

[[noreturn]]

jhkdiy：此节依然链接到属性一节

nullptr——空指针标识

nullptr——空指针标识

空指针标识(nullptr)（其本质是一个内定的常量）是一个表示空指针的标识，它不是一个整数。（译注：这里应该与我们常用的 NULL 宏相区别，虽然它们都是用来表示空置针，但 NULL 只是一个定义为常整数0的宏，而 nullptr 是 C++11的一个关键字，一个内建的标识符。下面我们还将看到 nullptr 与 NULL 之间更多的区别。）

```
1    char* p = nullptr;
2    int* q = nullptr;
3    char* p2 = 0; //这里0的赋值还是有效的，并且 p=p2
4
5    voidf(int);
6    voidf(char*);
7
8    f(0);        //调用 f(int)
9    f(nullptr);  //调用 f(char*)
10
11   voidg(int);
12   g(nullptr);      //错误：nullptr 并不是一个整型常量
13   inti = nullptr; //错误：nullptr 并不是一个整型常量
```

（译注：实际上，我们这里可以看到 nullptr 和 NULL 两者本质的差别，NULL 是一个整型数0，而 nullptr 可以看成是一个 char *。）

参考：

the C++ draft section ???

[N1488==/03-0071] Herb Sutter and Bjarne Stroustrup:

A name for the null pointer: nullptr .

[N2214 = 07-0074] Herb Sutter and Bjarne Stroustrup:

A name for the null pointer: nullptr (revision 4) .

对重载(override)的控制: override

对重写（override）的控制：override

我们要在派生类中重写基类的某个虚函数对其行为进行重新定义，并不需要特别的关键字来说明这是函数重写，只需要两个函数的声明相同就可以了。例如：

```
1    structB {
2          virtualvoidf();
3          virtualvoidg()const;
4          virtualvoidh(char);
5          voidk();     // 非虚函数
6    };
7
8    structD : B {
9          voidf();     // 重写 B::f()
10         // 并不是重写 B::g()
11         // 两者的函数声明不同， const 也是函数声明中的一部分
12         voidg();
13
14         virtualvoidh(char); // 重写 B:: h ()
15         voidk();  //   不是重写 B::k() (B::k() 并不是一个虚函数 )
```

```
16    };
```

无须添加关键字的函数重写虽然灵活，但是却很容易让人晕头转向。（程序员这样写是什么意思？并且如果编译器对 suspicious code 代码不产生警告，则可能带来更大的问题。）例如：

程序员是想重写 B::g() 吗？（大多数情况下是的）.

程序员是想重写 B::h(char) 吗？（可能不是，因为他显式地重新指定了 virtual 关键字，而实际上在虚函数重写中并不需要重新指定 virtual 关键字）.

程序员是想重写 B::k() 吗？（可能是，但是那是不可能的，因为它并不是虚函数）.

现在，在 C++11中，我们可以使用新的 override 关键字，来让程序员可以更加明显地表明他对于重写的设计意图，增加代码的可读性。例如：

```
1    structD : B {
2          voidf() override;    // OK: 重写 B::f()
3          voidg() override;    // error: 不同的函数声明，不能重写
4          virtualvoidh(char); // 重写 B:: h ( char ); 可能会有警告
5          voidk() override;    // error: B::k() 不是虚函数
6    };
```

override 的声明只在基类有函数可以重写的时候有效，上面例子中的 g() 并没有基类的函数可供重写，所以添加在其中的 override 关键字会引起一个编译错误。但是，编译器并不保证一定能够发现这个错误，因为它并不是因为语言定义而引起的，但是它很容易被诊断出来（？）。

参考：

Standard: 10 Derived classes [class.derived] [9]

Standard: 10.3 Virtual functions [class.virtual]

[N3234==11-0004] Ville Voutilainen:

Remove explicit from class-head.

[N3151==10-0141] Ville Voutilainen:

Keywords for override control.

Earlier, more elaborate design.

[N3163==10-0153] Herb Sutter:

Override Control Using Contextual Keywords.

Alternative earlier more elaborate design.

[N2852==09-0042] V. Voutilainen, A. Meredith, J. Maurer, and C. Uzdavinis:Explicit Virtual Overrides.

Earlier design based on attributes.

[N1827==05-0087] C. Uzdavinis and A. Meredith:

An Explicit Override Syntax for C++.

The original proposal.

对重载(override)的控制：final

有时候，程序员可能想要阻止某个虚函数被重写，在这种情况下，他可以为虚函数加上 final 关键字来达到这个目的。例如：

```
1    structB {
2        virtualvoidf()constfinal;   // 不能重写
3        virtualvoidg();// 没有 final 关键字，可以重写
4    };
5
6    structD : B {
7        // 错误: D::f 尝试重写 final 修饰的 B::f 会产生编译错误
8        voidf()const;
9        voidg();    // OK
10   };
```

我们有很多正当的理由来阻止一个虚函数被他的派生类重写，但是我担心我用来展示 C++对 final 的需要的例子，都是基于虚函数非常影响性能（通常是相对于其他语言而言的）的错误假设上。所以，如果为虚函数添加上一个 final 关键字让你感觉到不舒服（某种被驱促的感觉），请再次检查你添加 final 关键字的理由是符合逻

辑的：如果某人重新定义了一个派生类并重写了这个虚函数，会产生语义上的错误吗？

（译注：例如，Human 有一个来自它的父类的虚函数 move()，而当 Girl 派生自 Human 时，就不应该再重写这个虚函数，因为 Human 和 Girl 的 move()是一样的，不需要重新定义，在这种情况下，可以用 final 关键字来阻止 override 的发生。

```
1    classAnimal
2    {
3    public:
4        virtualvoidmove()
5        {
6            cout< <"动物爬行"<<endl;
7        }
8    };
9
10   classHuman :publicAnimal
11   {
12   public:
13       virtualvoidmove() final
14       {
15           cout<<"人只能行走"<<endl;
16       }
17   };
18
19   classGirl :publicHuman
20   {
21   public:
22       //错误，Girl 和 Human 的 move()方式是一样的
23       // 利用编译器来阻止这类错误的发生
24       virtualvoidmove()
25       {
26           cout<<"Girl 也得走路"<<endl;
27       }
28   };
```

）

添加 final 关键字关闭了我们对类的某个函数进行重新定义的可性，而对函数的重新定义，可以更好地实现这个函数。既然你不想让这个函数被重写，那么为什么最初又让它成为虚函数呢？我曾经遇到过的最合理的答案就是：这个函数是某个 framework 的基本功能函数，framework 的设计者需要 override 这些函数，而对于普

通的使用者而言，override 这些函数是非常危险的。我的偏见，这种要求是

可疑的。

如果出于性能（内联）的要求，或者是你只是简单地不希望重写，通常，你最好的做法就是在最开始的地方不要让这个函数成为虚函数。

C++不是 Java。

参考

Standard: 10 Derived classes [class.derived] [9]

Standard: 10.3 Virtual functions [class.virtual]

PODs

POD（大体介绍）

一个 POD（简单旧数据）（译注：你可以将 POD 类型看作是一种来自外太空的用绿色保护层包装的数据类型，POD

意为“Plain Old Data”，如果一定要译成中文，那就叫“彻头彻尾的老数据”怎么样？）指的是能够像 C 语言中的结构体那样进行处理的一种数据类型，比如能够使用 memcpy()进行复制，使用 memset()进行初始化等等。在 C++98标准中，POD 实际上是受限于结构体定义中的语言特性而定义的。

```
1   structS {inta; };//S 就是一个 POD
2    structSS {inta; SS(intaa) : a(aa) { } };// SS 就不是 POD 了
3    structSSS {virtualvoidf();/* ... */};
```

在 C++11的标准中，S 和 SS 都是“标准布置类型”（又叫作 POD），因为 SS 也没什么神奇的地方：构造函数并不影响它的初始变量布局（所以可以使用 memcpy()对其进行操作），只会影响其变量的初始化（memset()就不能用了——不能改变常量。（译注：如果使用 SS 定义一个常量，并使用 memset()函数直接操作内存，也

就是改变了常量的值(?))。而在 SSS 之中，SSS 还将会内嵌一个指向虚函数表的 vptr，所以和 POD 一点也不像，所以即使是新的标准也救不了它。在新的标准 C++0x 中，POD 被定义为可以简简单单复制的，类型普通的，并且拥有可以应对多种 POD 原先就能支持的操作的标准变量地址布局（？）。POD 的定义和以前差不多：

如果你所有的成员变量和基类都是 POD，那么这个类型就是一个 POD

和原先一样，POD 要满足以下的要求（在第9节[10]中有详细内容）

- 没有虚函数
- 没有虚基类
- 没有引用
- 没有多重访问（？）

新标准对 POD 最大的影响就是，对于拥有不会影响数据分配布局的构造函数的数据结构，也可以算作是 POD。

也可以参见：

the C++ draft section 3.9 and 9 [10]

[N2294=07-0154] Beman Dawes:

POD’s Revisited; Resolving Core Issue 568 (Revision 4)

range for statement(参见序列 for 循环语句)

参见序列 for 循环语句

原生字符串标识

在许多情况下，比如，你正在利用标准正则表达式库来写一些正则表达式，但此时，反斜杠(\)事实上却是一个“转义(escape)”操作符，这相当令人讨厌（因为正则表达式中的反斜杠本来是用于引入表示字符的特殊

符号）。考虑如何去写“被反斜杠分隔开的两个词”这样一个模式(w\w):

```
1    string s = “\w\\\w”; // 希望它是对的（译注：不直观、美观，且容易出错）
```

我们注意到，在正则表达式中，反斜杠字符被表达为两个反斜杠的组合。为了表示一个反斜杠，我们必须在正则表达式中使用两个反斜杠。第一个反斜杠表示这是一个转义字符，第二个才表示真正的反斜杠。基本上，在一个使用原生字符串标识 R 修饰的原生字符串中，一个反斜杠仅用一个反斜杠字符就可以表示。因而，上述的例子简化为：

```
1    string s = R"(\w\\\w)"; // 现在我可以确信我是对的
```

引发原生字符串标识提议的是这样一个“惊天地泣鬼神”的例子：

```
1    "('(?:[^\\\\']|\\\\.)*'|\"(?:[^\\\\\"]|\\\\.)*\")|"  // 这五个反斜杠是否正确?
2                // 即使是专家，也很容易被这么多反斜杠搞得晕头转向
```

R”(...)”记法相比于”...”会有一点点的冗长，但当你不必使用烦琐的“转义(escape)”符号时，“多一点”是必要的：如何将一个引用字符串放置于原生字符串内？如果它们不在”)”之后，那么非常简单：

```
1    R"("quoted string")"   // 这个字符串是 “quoted string”
```

但是，我们如何将字符串”)”放入一个原生字符串内呢？幸运地是，这是一个十分罕见的问题，但由于”(...)”是唯一默认的分隔符，我们可以在”(...)”中的(...)前后各加一个分隔符标志。例如：

```
1    // 字符串为: "quoted string containing the usual terminator (")"
2    R"***("quoted string containing the usual terminator (")")***"
```

在符号”)”之后的字符序列必须与符号”(“之前的字符序列相同。通过这种方式，我们可以处理（几乎）任意复杂的模式。

参考：

Standard 2.13.4

[N2053=06-0123] Beman Dawes: Raw string literals . (original proposal)

[N2442=07-0312] Lawrence Crowl and Beman Dawes: Raw and Unicode String Literals; Unified Proposal (Rev. 2) . (final proposal combined with the User-defined literals proposal).

（翻译：张潇，dabaitu）

右角括号

考虑如下代码：

```
1   list<vector<string>> lvs;
```

在 C++98中，这是一个非法的符号表达式，因为两个右角括号(‘>’)之间没有空格（译注：因此，编译器会将它分析为”>>”操作符）。C++0x 可以正确地分辨出这是两个右角括号(‘>’)，是两个模板参数列表的结尾。

为什么之前这会是一个问题呢？一般地，一个编译器前端会按照“分析/阶段”模型进行组织。简要描述如下：

词法分析（从字符中构造 token）

符号分析（检查语法）

类型检测（寻找名称和表达式的类型）

这些阶段在理论上，甚至在某些实际应用中，都是严格独立的。所以，词法分析器会认为”>>”是一个完整的 token（通常意味着右移操作符或是输入），而无法理解它的实际意义（译注：即在具体的上下文环境下，某一个符号的具体意义）。特别地，它无法理解模板或内置模板参数列表。然而，为了使上述示例“正确”，这三个阶段必须进行某种形式的交互、配合。解决这个问题的最关键的点在于，每一个 C++ 编译器已完整理解整个问题（译注：对整个问题进行了全部的词法分析、符号分析及类型检测，然后分析各个阶段的正确性），从而给出令人满意的错误消息。

参考：

the C++ draft section ???

[N1757==05-0017] Daveed Vandevoorde: revised right angle brackets proposal (revision 2) .

（翻译：张潇，dabaitu）

右值引用

左值（可被用在赋值操作符“=”的左侧，通常是一个变量）与右值（对应地，可被用在赋值操作符“=”的右侧，通常是一个常数或函数调用表达式）之间的差别可以追溯到 Christopher Strachey （C++的祖先语言 CPL 与外延语义学之父)时代。在 C++中 ,左值可被绑定到非 const 引用 ,左值或者右值则可被绑定到 const 引用。但是却没有什么可以绑定到非 const 的右值（译注：即右值无法被非 const 的引用绑定），这是为了防止人们修改临时变量的值，这些临时变量在被赋予新的值之前，都会被销毁。例如：

```
1    voidincr(int& a) { ++a; }
2    inti = 0;
3    incr(i);   // i 变为1
4    //错误：0不是一个左值
5    //（译注：0不是左值，无法直接绑定到非 const 引用：int&。
6    // 假如可行，那么在调用时，将会产生一个值为0的临时变量，
7    // 用于绑定到 int&中，但这个临时变量将在函数返回时被销毁，
8    // 因而，对于它的任何更改都是没有意义的，
9    // 所以编译器拒绝将临时变量绑定到非 const 引用，但对于 const 的引用，
10   // 则是可行的）。
11   incr(0);
```

如果 incr(0)被允许的话，那么要么会产生一个任何人都无法看见的临时变量，然后对它进行增加操作，要么，在更糟糕的情况下，0的值会变成1。后者听起来后果相当严重，但事实上，早期的 Fortran 编译器中确实存在这样一个 bug：对于值为0的内存位置分配，将不予理会。

到目前为止，一切都很美好。但，考虑如下函数：

```
1   template<classT> swap(T& a, T& b)// 老式的 swap 函数
2       {
3           T tmp(a);// 现在有两份 a
4           a = b;     // 现在有两份 b
```

```
5           b = tmp;    // 现在有两份 tmp (值同 a)
6       }
```

如果 T 是一个复制其中元素的代价相当昂贵的类型，例如 string 和 vector，swap 将会变成一个十分昂贵的操作（对于标准库来说，我们已经对 string 和 vector 类型的 swap 函数进行了特化来处理这个问题）。注意这个奇怪的现象 :事实上 ,我们并不需要任何变量的拷贝操作。我们仅仅是想将变量 a,b 和 tmp 的值做一个移动(译注：即通过 tmp 来交换 a,b 的值)。

在 C++11 中，我们可以定义“移动构造函数(move constructors)”和“移动赋值操作符(move assignments”来移动而非复制它们的参数：

```
1   template<classT>classvector {
2           // …
3           vector(constvector&); // 复制构造函数
4           vector(vector&&);  // 移动构造函数
5           vector& operator= (constvector&);// 复制赋值函数
6           vector& operator =(vector&&); // 移动赋值函数
7   };//注意：移动构造函数和移动赋值操作符接受
8   // 非 const 的双重引用(&&)，它们可以，而且经常这样，
9   // 修改它们的参数。
```

”&&”表示“右值引用”。一个右值引用可以绑定到一个右值（但不能绑定到左值）：

```
1   X a;
2   X f();
3   X& r1 = a;        // 将 r1绑定到 a(一个左值)
4   X& r2 = f();    // 错误：f()的值是一个右值，无法绑定
5
6   X&& rr1 = f();    // 没问题：将 rr1绑定到临时变量
7   X&& rr2 = a;    // 错误：无法将右值引用 rr2绑定到左值 a
```

存在于移动赋值操作背后的思想是，并非一定需要产生一份拷贝，你还可以构造一个源对象的代表，然后简单地替换掉它。例如：在表达式 s1 = s2 中，字符串 s1并不产生字符串 s2的拷贝，而是让 s1将 s2中的字符序列当做是它自己的，同时删除掉 s1中原有的字符串（也可能把它们遗留在 s2中，但大多数情况下，它们也面临被销毁的命运）。（译注：仔细体会 copy 与 move 的区别。）

我们如何知道简单地对源对象进行移动是否正确呢？我们可以告诉编译器：

```
1   template<classT>
2   voidswap(T& a, T& b) //“完美 swap”（大多数情况下）
3   {
4           T tmp = move(a);    // 可以使变量 a 失效（？）
5           a = move(b);        // 可以使变量 b 失效
6           b = move(tmp);    // 可以使变量 tmp 失效
7   }
```

move(x) 意味着“你可以把变量 x 当做一个右值”，把 move()称为 rval()或许会更好，但是如今，move()已经使用很多年了。现在，在 C++0x 中，也可以使用 move()模板函数（参考“brief introduction”），以及右值引用了。

右值引用同时也可以用作完美转发(perfect forwarding)。

在 C++11的标准库中，所有的容器都提供了移动构造函数和移动赋值操作符，那些插入新元素的操作，如 insert()和 push_back()，也都有了可以接受右值引用的版本。最终的结果是，在没有用户干预的情况下，标准容器和算法的性能都提升了，而这些都应归功于复制操作的减少。

参考：

the C++ draft section ???

N1385 N1690 N1770 N1855 N1952

[N2027==06-0097] Howard Hinnant, Bjarne Stroustrup, and Bronek Kozicki:

A brief introduction to rvalue references

[N1377=02-0035] Howard E. Hinnant, Peter Dimov, and Dave Abrahams:

A Proposal to Add Move Semantics Support to the C++ Language (original proposal).

[N2118=06-0188] Howard Hinnant:

A Proposal to Add an Rvalue Reference to the C++ Language Proposed Wording (Revision 3)

(final proposal).

（翻译：dabaitu，感谢：dave）

Simple SFINAE rule

Stroustrup 先生尚未完成这个主题，请稍后再来。

静态（编译期）断言 — static_assert

一个静态（编译期）断言由一个常量表达式及一个字符串文本构成：

```
1   static_assert(expression, string);
```

编译期对 expression 进行求值，当表达式为 false（例如：断言失败）时，将 string 作为错误消息输出。例如：

```
1   static_assert(sizeof(long) >= 8,
2      “64-bit code generation requiredforthislibrary.” );
3
4   structS { X m1; Y m2; };
5
6   static_assert(sizeof(S)==sizeof(X)+sizeof(Y),
7       ” unexpected padding in S” );
```

一个 static_assert 断言在判断某种假设是否成立（译注：比如判断当前平台是否是64位平台）并提供相应的解决方法时十分有用。必须注意到，由于 static_assert 在编译期进行求值，它不能用于依赖于运行时变量值的假设检验。例如：

```
1   intf(int* p,intn)
2   {
3         //错误：表达式“p == 0”不是一个常量表达式
4         static_assert(p == 0,
5             “p is not null” );
6   }
```

（作为替代方案，可以进行运行时测试，然后在测试失败时抛出异常）

参考：

the C++ draft 7 [4].

[N1381==02-0039] Robert Klarer and John Maddock: Proposal to Add Static Assertions to the Core Language .

[N1720==04-0160] Robert Klarer, John Maddock, Beman Dawes, Howard Hinnant:Proposal to Add Static Assertions to the Core Language (Revision 3) .

（翻译：张潇，dabaitu）

模板别名（正式的名称为”template typedef”）

如何构造一个与另外一个模板“类似”，但某一些模板参数已被特化（绑定）的模板？

考虑如下代码：

```
template<classT>
// 采用了自己设计的空间配置器（一般为内存分配器）的标准 vector
usingVec = std::vector<T,My_alloc<T>>;

// 使用自定义的空间配置器为元素分配存储空间
Vec<int> fib = { 1, 2, 3, 5, 8, 13 };

// verbose 与 fib 是相同类型
vector<int,My_alloc<int>> verbose = fib;
```

关键词 using 用于表示一个相当直观的概念：“引用跟在名字之后的东西”（译注：即将’=’前后的事物等同起来）。我们也曾尝试过传统且相当复杂的 typedef 方法，但除非我们选择一个不太晦涩的符号，否则始终无法得到一个完整且一致的解决方案。

这(using 关键字)对模板特化同样有效。(可以为一组特化模板设定一个别名，但无法直接去特化一个别名（所代表的模板）(译注：即模板特化操作不能通过别名进行）。例如：

```
template<int>
// idea: int_exact_trait::type 是一个拥有 N 个 bit 的类型
    structint_exact_traits {
        typedefinttype;
```

```
5       };
6
7       template<>
8       structint_exact_traits<8> {
9           typedefchartype;
10      };
11
12      template<>
13      structint_exact_traits<16> {
14          typedefchar[2] type;
15      };
16
17      // ...
18
19      template<intN>
20      // 为了方便，为这种转换定义一个别名
21          usingint_exact =typenameint_exact_traits<N>::type;
22          // int_exact<8> 是一个拥有8个 bit 的 int 类型
23      int_exact<8> a = 7;
```

除了在联系模板的过程中有重要作用，类型别名也可以用于普通类型，以简化它们（以我的观点，这样更好）。

```
1   typedefvoid(*PFD)(double);   // C 样式
2   usingPF =void(*)(double);   // using 加上 C 样式的类型
3   usingP = [](double)->void; // using 和 后缀返回类型
```

参考：

参考：

the C++ draft: 14.6.7 Template aliases; 7.1.3 The typedef specifier

[N1489=03-0072] Bjarne Stroustrup and Gabriel Dos Reis: Templates aliases for C++ .

[N2258=07-0118] Gabriel Dos Reis and Bjarne Stroustrup: Templates Aliases (Revision 3) (final proposal).

（翻译：张潇，dabaitu）

template typedef(参见模板别名)

线程本地化存储 (thread_local)

抱歉，目前我还没有完成这个主题，请稍后再来。

如果你对这一主题感兴趣，可以参考本站的：

C++小品：井水不犯河水的 thread_specific_ptr，C++11线程库中的本地存储

C++小品：井水不犯河水在 PPL 中的实现：combinable 以及 task_group,task

参考：

[N2659 = 08-0169] Lawrence Crowl:

Thread-Local Storage (Final proposal).

unicode 字符

抱歉，目前我尚未完成这个主题，请稍后再来。

（译注：C++对 unicode 的支持不是特别重视）

统一初始化的语法和语义

C++提供了多种按照对象的类型以及初始化上下文对一个对象进行初始化的方法。在被错用的时候，错误产生的效果会十分惊人，而错误的（调试）信息却十分模糊（想要找出来也十分困难）。考虑如下的代码：

```
1   string a[] = { “foo”, ” bar” }; //正确：初始化数组变量
2   //错误：初始化列表应用在了非聚合的向量上
3   vector<string> v = { “foo”, ” bar” };
4   voidf(string a[]);
5   f( { “foo”, ” bar” } );  //语法错误，把一个块（block）作为了参数
```

还有以下代码：

```
1   inta = 2;         //“赋值风格”的初始化
2   intaa[] = { 2, 3 };//用初始化列表进行的赋值风格的初始化
3   complex z(1,2);  //“函数风格”的初始化
4   x = Ptr(y);     //  “函数风格”的转换/赋值/构造操作
```

还有以下的代码：

```
1   inta(1); //变量的定义
2   intb();   //函数的声明
3   intb(foo);   // 变量的定义，或者函数的声明
```

要记住变量的初始化规则并且寻找一个最佳的方式去初始化变量是比较困难的。

C++11的解决方法是对于所有的初始化，允许使用“{}-初始化变量列表”：

```
1    X x1 = X{1,2};
2    X x2 = {1,2};    //这个=是可选的，不必要
3    X x3{1,2};
4    X* p =newX{1,2};
5
6    structD : X {
7        D(intx,inty) :X{x,y} {/* … */};
8    };
9
10   structS {
11       inta[3];
12       //对于老问题的解决方案
13       S(intx,inty,intz) :a{x,y,z} {/* … */};
14   };
```

重要的是，X{a}在所有的执行代码中都创建了一个相同的值，所以使用“{}”进行的初始化，在合法使用的时候，在任何地方都产生相同的结果。例如：

```
1   X x{a};
2   X* p =newX{a};
3   z = X{a};         // 使用了类型转换
4   f({a});           // 函数的实际参数（X 类型的）
5   return{a};       //函数的返回值（函数返回类型为 X）
```

还有其他的文档可以参考：

the C++ draft section ???

[N2215==07-0075] Bjarne Stroustrup and Gabriel Dos Reis: Initializer lists (Rev. 3) .

[N2640==08-0150] Jason Merrill and Daveed Vandevoorde: Initializer Lists — Alternative Mechanism and Rationale (v. 2) (final proposal).

（翻译：张潇）

（广义的）联合体

在 C++98版本中（一个 C++标准的较早版本），拥有用户自定义的构造函数，析构函数或者赋值操作符的成员，不能成为一个联合体成员。

```
1   unionU {
2    intm1;
3    complex m2;   //错误（明显的）：complex 拥有构造函数
4    //错误（不那么明显）：string 是由构造函数，
5    // 拷贝构造函数，和析构函数维护的共同维护的
6    string m3;
7    };
```

特殊情况下：

```
1   U u;       // 使用成员的哪个构造函数呢？
2   u.m1 = 1;    // 给整形成员赋值
3   string s = u.m3;   //灾难：从字符串成员中读取数据
```

显而易见，把值写入一个成员，之后又读取另外一个成员的做法是非法的。然而人们往往因为一些失误而这么干 。

C++11改变了联合体对成员的限制，以使更多的成员类型可以在其中声明；特别的是，新的标准允许有构造函数和析构函数的成员类型。新的标准还添加了一个规则，通过鼓励创建可识别的联合体(?)，使得联合体更加灵活而产生更少的错误。

联合体的成员类型有如下的限制：

没有虚函数（和以往一样）

没有引用（和以往一样）

没有基类（和以往一样）

如果一个联合体成员有用户定义的构造函数，拷贝构造函数，或者是析构函数，那么这些特殊的函数将导致联合体的相应函数被删除掉（译注：在 C++0x 中，我们可以使用 delete 关键字删除掉某个类的默认函数）；导致的结果是，我们将不能创建这种联合体类型的对象。这条规则是新添加的。

例如：

```
1   unionU1 {
2       intm1;
3       complex m2;   // ok//正确
4    };
5
6    unionU2 {
7       intm1;
8       string m3;   // ok//正确
9    };
```

这看起来比较容易出错，但是在新的规定下是正确的。特殊的：

```
1   U1 u;         // ok//正确
2   u.m2 = {1,2};   //正确：给 complex 成员赋值
3   U2 u2;  //错误：string 的析构函数导致 U 的析构函数被删除
4   U2 u3 = u2;   //错误：string 的拷贝构造函数导致 U 的拷贝构造函数被删除
```

基本上，U2是没有什么用的，除非它被嵌入在一个能够确定它的哪个成员（固定的）是一直被使用的结构体中。

所以，创建一个可识别的联合体是必要的，例如：

```
1    // 包含三种可供选择的实现联合体的方法
2    classWidget {
3    private:
4       //使用枚举判别
5       enumclassTag { point, number, text } type;
6       union{       //替代
7           point p;     //point 类有构造函数
8           inti;
9           //string 有默认的构造函数，拷贝构造函数和析构函数
10          string s;
```

```
    };
    // …
    //由于 string 中存在拷贝构造函数，所以这个是必要的。
    widget& operator=(constwidget& w)
    {
        if(type==Tag::text && w.type==Tag::text) {
            s = w.s;       //使用 string 的赋值操作符
            return*this;
        }

        if(type==Tag::text) s.~string();   // 析构（显式地）

        switch(type==w.type) {
        caseTag::point: p = w.p;break;   //普通的拷贝
        caseTag::number: i = w.i;break;
        caseTag::text:new(&s)(w.s);break;   //放置 new，创建新的对象
        }
        return*this;
    }
};
```

也可以参照：

[N2544=08-0054] Alan Talbot, Lois Goldthwaite, Lawrence Crowl, and Jens Maurer:Unrestricted unions (Revison 2)

用户定义数据标识（User-defined literals）

C++提供了许多内建数据类型的数据标识（2.14节变量）：

```
123   // int 整型
1.2   // double 双精度型
1.2F   // float 浮点型
’a'   // char 字符型
1ULL   // unsigned long long64位无符号长整型
0xD0   // hexadecimal unsigned 十六进制无符号整型
"as"    // string 字符串
```

但是在 C++98中并没有为用户自定义的变量类型提供数据标识。这就违反甚至冲突于“用户自定类型应该和

内建类型一样得到支持”的原则。在特殊情况下，人们有以下的需求：

```
"Hi!"s            //字符串，不是“以零字符为终结的字符数组”
1.2i              //虚数
123.4567891234df    //十进制浮点型（IBM）
101010111000101b    //二进制
123s              //秒
123.56km          //不是英里（单位）
12345678901234567890123456789012345678901234567890x        //扩展精度
```

C++11通过在变量后面加上一个后缀来标定所需的类型以支持“用户定义数据标识”，例如：

```
constexpr complex operator "" i(long double d)    // 设计中的数据标识
{
    return {0,d};    //complex 是一个数据标识
}

// 将 n 个字符构造成字符串 std::string 对象的数据标识
std::string operator""s (const char* p, size_t n)
{
    return string(p,n);    // 需要释放存储空间
}
```

这里需要注意的是，constexpr 的使用可以进行编译时期的计算。使用这一功能，我们可以这样写：

```
template <class T> void f(const T&);
f("Hello");    // 传递 char*指针给 f()
f("Hello"s);    // 传递（5个字符的）字符串对象给 f()
f("Hello n"s);    // 传递（6个字符的）字符串对象给 f()

auto z = 2+1i;    // 复数 complex(2,1)
```

基本（实现）方法是编译器在解析什么语句代表一个变量之后，再分析一下后缀。用户自定义数据标识机制只是简简单单的允许用户制定一个新的后缀，并决定如何对它之前的数据进行处理。要想重新定义一个内建的数据标识的意义或者它的参数、语法是不可能的。一个数据标识操作符可以使用它（前面）的数据标识传递过来的处理过的值（如果是使用新的没有定义过的后缀的值）或者没有处理过的值（作为一个字符串）。

要得到一个没有处理过的字符串，只要使用一个单独的 const char*参数即可，例如：

```
Bignum operator"" x(const char* p)
{
    return Bignum(p);
```

```
4    }
5
6    void f(Bignum);
7    f(12345678901234567890123456789012345678901234567890x);
```

这个 C 语言风格的字符串“12345678901234567890123456789012345678901234567890”被传递给了操作符 operator”" x()。注意，我们并没有明确地把数字转换成字符串。

有以下四种数据标识的情况，可以被用户定义后缀来使用用户自定义数据标识：

整型标识：允许传入一个 unsigned long long 或者 const char*参数

浮点型标识：允许传入一个 long double 或者 const char*参数

字符串标识：允许传入一组(const char*,size_t)参数

字符标识：允许传入一个 char 参数。

注意，你为字符串标识定义的标识操作符不能只带有一个 const char*参数（而没有大小）。例如：

```
1    //警告，这个标识操作符并不能像预想的那样子工作
2    string operator"" S(const char* p);
3    "one two"S;     //错误，没有适用的标识操作符
```

根本原因是如果我们想有一个“不同的字符串”，我们同时也想知道字符的个数。后缀可能比较短（例如，s 是字符串的后缀，i 是虚数的后缀，m 是米的后缀，x 是扩展类型的后缀），所以不同的用法很容易产生冲突，我们可以使用 namespace（命名空间）来避免这些名字冲突：

```
1    namespaceNumerics {
2        // …
3        classBignum {/* … */};
4        namespaceliterals {
5            operator""X(charconst*);
6        }
7    }
8
9    usingnamespaceNumerics::literals;
```

参考：

Standard 2.14.8 User-defined literals

[N2378==07-0238] Ian McIntosh, Michael Wong, Raymond Mak, Robert Klarer, Jens Mauer, Alisdair Meredith, Bjarne Stroustrup, David Vandevoorde:

User-defined Literals (aka. Extensible Literals (revision 3)).

可变参数模板（Variadic Templates）

要解决的问题：

怎么创建一个拥有1个、2个或者更多的初始化器的类？

怎么避免创建一个实例而只拷贝部分的结果？

怎么创建一个元组？

最后的问题是关键所在：考虑一下元组！如果你能创建并且访问一般的元组，那么剩下的问题也将迎刃而解。

这里有一个例子（摘自“可变参数模板简述（A brief introduction to Variadic templates）”（参见参考）），要构建一个广义的、类型安全的 printf()。这个方法比用 boost::format 好的多，但是考虑一下：

```
1   conststring pi = “pi”;
2   constchar* m =
3       “The value of %s is about %g (unless you live in %s).n”;
4   printf(m,  pi, 3.14159,  “Indiana”);
```

这是除了格式字符串之外，没有其它参数的情况下调用 printf()的一个最简单的例子了，所以我们将要首先解决：

```
1   voidprintf(constchar* s)
2    {
3        while(s && *s) {
4            if(*s==’ %’  && *++s!=’ %') //保证没有更多的参数了
5    //%%（转义字符，在格式字符串中代表%
6                throwruntime_error(“格式非法: 缺少参数”);
7            std::cout << *s++<<endl;
8        }
9    }
```

这个处理好之后，我们必须处理有更多参数的 printf()：

```
1    template<typenameT,typename... Args>   // 注意这里的"..."
2        voidprintf(constchar* s, T value, Args... args)  // 注意"..."
3        {
4            while(s && *s) {
5                //一个格式标记（避免格式控制符）
6                            if(*s=='%'&& *++s!='%') {
7                    std::cout << value;
8                    returnprintf(++s, args...);//使用第一个非格式参数
9                }
10               std::cout << *s++;
11           }
12           throwstd::runtime error("extra args provided to printf");
13       }
```

这段代码简单地“去除”了开头的无格式参数，之后递归地调用自己。当没有更多的无格式参数的时候，它调用第一个（很简单）printf()（如上所示）。这也是标准的函数式编程在编译的时候做的(?)。注意,< <的重载代替了在格式控制符当中（可能会有错误）的花哨的技巧。（译注：我想这里可能指的是使用重载的<<输出操作符，就可以避免使用各种技巧复杂的格式控制字符串。）

Args...定义的是一个叫做“参数包”的东西。这个“参数包”仅仅是一个（有各种类型的值的）队列，而且这个队列中的参数可以从头开始进行剥离（处理）。如果我们使用一个参数调用 printf()，函数的第一个定义(printf(const char*))就被调用。如果我们使用两个或者更多的参数调用 printf()，那么函数的第二个定义(printf(const char*, T value, Args... args))就会被调用，把第一个参数当作字符串，第二个参数当作值，而剩余的参数都打包到参数包 args 中，用做函数内部的使用。在下面的调用中：

```
1    printf(++s, args…);
```

参数包 args 被打开，所以参数包中的下一个参数被选择作为值。这个过程会持续进行，直到 args 为空（所以第一个 printf()最终会被调用）。

如果你对函数式编程很熟悉的话，你可能会发现这个语法和标准技术有一点不一样。如果发现了，这里有一些小的技术示例可能会帮助你理解。首先我们可以声明一个普通的可变参数函数模板（就像上面的 printf()）：

```
1    template<class... Types>
```

```
2          // 可变参数模板函数
3                //（补充：一个函数可以接受若干个类型的若干个参数）
4                voidf(Types ... args);
5
6       f();       // OK: args 不包含任何参数
7       f(1);      // OK: args 有一个参数: int
8       f(2, 1.0); // OK: args 有两个参数: int 和 double
```

我们可以建立一个具有可变参数的元组类型：

```
1    template<typenameHead,typename... Tail>
2        //这里是一个递归
3    //一个元组最基本要存储它的 head（第一个（类型/值））对
4                //并且派生自它的 tail（剩余的（类型/值））对
5                //注意，这里的类型被编码，而不是按一个数据来存储
6    classtuple<Head, Tail...>
7            :privatetuple<Tail...> {
8            typedeftuple<Tail...> inherited;
9        public:
10           tuple() { }// 默认的空 tuple
11
12           //从分离的参数中创建元组
13           tuple(typenameadd_const_reference<Head>::type v,
14        typenameadd_const_reference<Tail>::type... vtail)
15               : m_head(v), inherited(vtail...) { }
16
17           // 从另外一个 tuple 创建 tuple:
18           template<typename... VValues>
19           tuple(consttuple<VValues...>& other)
20           :    m_head(other.head()), inherited(other.tail()) { }
21
22           template<typename... VValues>
23           tuple& operator=(consttuple<VValues...>& other) // 等于操作
24           {
25               m_head = other.head();
26               tail() = other.tail();
27               return*this;
28           }
29
30           typenameadd_reference<Head>::type head()
31                   {returnm_head; }
32           typenameadd_reference<constHead>::type head()const
33                   {returnm_head; }
34
35           inherited& tail() {return*this; }
```

```
36          constinherited& tail()const{return*this; }
37      protected:
38          Head m_head;
39      }
```

有了定义之后，我们可以创建元组（并且复制和操作它们）：

```
1   tuple<string,vector,double> tt("hello",{1,2,3,4},1.2);
2       string h = tt.head();  // "hello"
3       tuple<vector<int>,double> t2 = tt.tail();
```

要实现所有的数据类型可能会比较乏味，所以我们经常减少参数的类型，例如，可以使用标准库中的

make_tuple()函数：

```
1    template<class... Types>
2       // 这个定义十分简单（参见标准20.5.2.2）
3           tuple<Types...> make_tuple(Types&&... t)
4       {
5           returntuple<Types...>(t...);
6       }
7
8       string s ="Hello";
9       vector<int> v = {1,22,3,4,5};
10      auto x = make_tuple(s,v,1.2);
```

参考：

Standard 14.6.3 Variadic templates

[N2151==07-0011] D. Gregor, J. Jarvi:

Variadic Templates for the C++0x Standard Library.

[N2080==06-0150] D. Gregor, J. Jarvi, G. Powell:

Variadic Templates (Revision 3).

[N2087==06-0157] Douglas Gregor:

A Brief Introduction to Variadic Templates.

[N2772==08-0282] L. Joly, R. Klarer:

Variadic functions: Variadic templates or initializer lists? -- Revision 1.

[N2551==08-0061] Sylvain Pion:

A variadic std::min(T, ...) for the C++ Standard Library (Revision 2) .

Anthony Williams:

An introduction to Variadic Templates in C++0x.

DevX.com, May 2009.

我经常从提案中借用一些例子。所以，我要感谢这些提案的作者们。另外，我也从自己的访谈和论文中借用了很多例子。

关于标准库的问题：

abandoning_a_process

Stroustrup 尚未完成此主题，期待中。

参考：

Abandoning a process

（翻译：interma）

算法方面的改进

标准库的算法部分进行了如下改进：新增了一些算法函数；通过新语言特性改善了一些算法实现并且更易于使用。下面分别来看一些例子：

新算法:

```
1       boolall_of(Iter first, Iter last, Pred pred);
```

```
    boolany_of(Iter first, Iter last, Pred pred);
    boolnone_of(Iter first, Iter last, Pred pred);

    Iter find_if_not(Iter first, Iter last, Pred pred);

    OutIter copy_if(InIter first, InIter last,
           OutIter result, Pred pred);
    OutIter copy_n(InIter first, InIter::difference_type n,
           OutIter result);

    OutIter move(InIter first, InIter last, OutIter result);
    OutIter move_backward(InIter first, InIter last, OutIter result);

    pair<OutIter1, OutIter2> partition_copy(InIter first, InIter last,
           OutIter1 out_true, OutIter2 out_false, Pred pred);
    Iter partition_point(Iter first, Iter last, Pred pred);

    RAIter partial_sort_copy(InIter first, InIter last,
           RAIter result_first, RAIter result_last);
    RAIter partial_sort_copy(InIter first, InIter last,
           RAIter result_first, RAIter result_last, Compare comp);
    boolis_sorted(Iter first, Iter last);
    boolis_sorted(Iter first, Iter last, Compare comp);
    Iter is_sorted_until(Iter first, Iter last);
    Iter is_sorted_until(Iter first, Iter last, Compare comp);

    boolis_heap(Iter first, Iter last);
    boolis_heap(Iter first, Iter last, Compare comp);
    Iter is_heap_until(Iter first, Iter last);
    Iter is_heap_until(Iter first, Iter last, Compare comp);

    T min(initializer_list<T> t);
    T min(initializer_list<T> t, Compare comp);
    T max(initializer_list<T> t);
    T max(initializer_list<T> t, Compare comp);
    pair<constT&,constT&> minmax(constT& a,constT& b);
    pair<constT&,constT&> minmax(constT& a,
           constT& b,
            Compare comp);
    pair<constT&,constT&> minmax(initializer_list<T> t);
    pair<constT&,constT&> minmax(initializer_list<T> t,
            Compare comp);
    pair<Iter, Iter> minmax_element(Iter first, Iter last);
    pair<Iter, Iter> minmax_element(Iter first, Iter last, Compare comp);
```

```
46
47   // 填充[first,last]范围内的每一个元素
48       // 第一个元素为 value，第二个为++value，以此类 ui
49       // 等同于
50       // *(d_first)   = value;
51       // *(d_first+1) = ++value;
52       // *(d_first+2) = ++value;
53       // *(d_first+3) = ++value; ...
54       // 注意函数名，是 iota 而不是 itoa 哦
55       voidiota(Iter first, Iter last, T value);
```

更有效的 move：more 操作比 copy 操作更有效率（参看 move semantics（译注：实际上是一个右值(rval)问题，核心是减少创建不必要的对象））。基于 move 的 std::sort()和 std::set::insert()要比基于 copy 的对应版本快15倍以上。不过它对标准库中已有操作的性能改善不多，因为它们的实现中已经使用了类似的方法进行优化了（例如 string，vector 使用了调优过的 swap 操作来代替 copy 了）。当然如果你自己的代码中包含了 move 操作的话，就能自动从新标准库中获益了。试着用 move 操作来替代下边这个 sort 函数中的智能指针（unique_ptr）吧（译注：可以通过一个 move 版 swap 来搞定，参看之前 move semantics 文章）：

```
1   template<classP>structCmp<P> {//比较 *P 的值
2           booloperator() (P a, P b)const
3                       {return*a<*b; }
4       }
5
6       vector<std::unique_ptr<Big>> vb;
7       //  用指向大对象的 unique_ptr 填充 vb
8
9       sort(vb.begin(),vb.end(),Cmp<Big>());// 不要像这样使用时用 auto_ptr
```

对 lambda 表达式的使用：对于为标准库算法写函数/函数对象（function object，推荐）这个事儿大家已经抱怨很久了（例如 Cmp）。特别是在 C++98标准中，这会令人更加痛苦，因为无法定义一个局部的函数对象。不过现在好多了，lambda 表达式允许用”inline”的方式来写函数了：

```
1   sort(vb.begin(),vb.end(),
2         [](unique_ptr a, unique_ptr b) {return*a< *b; });
```

我希望大家尽量多用用 lambda 表达式（它真的是一种很好很强大的机制）（译注：原文是：”I expect lambdas to be a bit overused initially (like all powerful mechanisms)”，非字面翻译）

对初始化列表（initializer lists）的使用：初始化表有时可以像参数那样方便的使用。看下边这个例子（x,y,z 是 string 变量，Nocase 是一个大小写不敏感的比较函数）：

```
1    auto x = max({x,y,z},Nocase());
```

参考：

25 Algorithms library [algorithms]

26.7 Generalized numeric operations [numeric.ops]

Howard E. Hinnant, Peter Dimov, and Dave Abrahams:

A Proposal to Add Move Semantics Support to the C++ Language.

N1377=02-0035.

array(数组)

std::array 是一个支持随机访问且大小（size）固定的容器（译注：可以认为是一个紧缩版的 vector 吧）。它有如下特点：

不预留多余空间，只分配必须空间（译注：size() == capacity()）。

可以使用初始化表（initializer list）的方式进行初始化。

保存了自己的 size 信息。

不支持隐式指针类型转换。

换句话说，可以认为它是一个很不错的内建数组类型。一些代码片段：

```
1       array<int,6> a = { 1, 2, 3 };
2    a[3]=4;
3    int x = a[5];    // array 的默认数据元素为0，所以 x 的值变成0
4    int* p1 = a; // 错误: std::array 不能隐式地转换为指针
5    int* p2 = a.data(); // 正确，data()得到指向第一个元素的指针
```

不过要注意：是可以定义一个长度为0的 array 的；但是无法从初始化表中推导出 size 信息：

```
array<int> a3 = { 1, 2, 3 }; //错误：没有 size 信息
    array<int,0> a0;  // 正确：没有任何元素
    int* p = a0.data(); // 为定义行为，不要这样做
```

array 非常适合在嵌入式系统（和有类似限制/性能敏感/安全关键系统等）中使用。它提供了序列型容器该有的大部分通用函数（和 vector 很像）：

```
template<classC> C::value_type sum(constC& a)
 {
        returnaccumulate(a.begin(),a.end(),0);
 }

 array<int,10> a10;
 array<double,1000> a1000;
 vector<int> v;
 // …
 intx1 = sum(a10);
 intx2 = sum(a1000);
 intx3 = sum(v);
```

但是，它是不支持由子类到基类的自动类型转换的（注意这个潜在陷阱）：

```
structApple : Fruit {/* … */};
structPear : Fruit {/* … */};

voidnasty(array<fruit *,10>& f)
{
        f[7] =newPear();
};

array<apple ,10> apples;
// …
nasty(apples); // 错误：不能将 array 转换为 array;
```

如果支持这种转换的话，apple array 中就能放 pear 啦。

参考：

Standard: 23.3.1 Class template array

（翻译：interma）

async()函数

async()函数是一个简单任务的”启动”（launcher）函数，它是本 FAQ 中唯一一个尚未在标准草案中投票通过的特性。我希望它能在调和两个略微不同的意见之后最终于10月份获得通过（记得随时骚扰你那边的投票委员，一定要为它投票啊 ）。

下边是一种优于传统的线程+锁的并发编程方法示例(译注：山寨 map-reduce 哦)：

```
template<classT,classV>structAccum  {// 简单的积函数对象
        T* b;
        T* e;
        V val;
        Accum(T* bb, T* ee,constV& v) : b{bb}, e{ee}, val{vv} {}
        V operator() ()
        {returnstd::accumulate(b,e,val); }
};

voidcomp(vector<double>& v)
        // 如果 v 够大，则产生很多任务        {
        if(v.size()<10000)
            returnstd::accumulate(v.begin(),v.end(),0.0);

        auto f0 {async(Accum{&v[0],&v[v.size()/4],0.0})};
        auto f1 {async(Accum{&v[v.size()/4],&v[v.size()/2],0.0})};
        auto f2 {async(Accum{&v[v.size()/2],&v[v.size()*3/4],0.0})};
        auto f3 {async(Accum{&v[v.size()*3/4],&v[v.size()],0.0})};

        returnf0.get()+f1.get()+f2.get()+f3.get();
}
```

尽管这只是一个简单的并发编程示例（留意其中的”magic number “），不过我们可没有使用线程，锁，缓冲区等概念。f*变量的类型（即 async()的返回值）是”std::future”类型。future.get()表示如果有必要的话则等待相应的线程(std::thread)运行结束。async 的工作是根据需要来启动新线程，而 future 的工作则是等待新线程运行结束。”简单性”是 async/future 设计中最重视的一个方面；future 一般也可以和线程一起使用，不过不要使用 async()来启动类似 I/O 操作，操作互斥体（mutex），多任务交互操作等复杂任务。async()背后的理念和 range-for statement 很类似：简单事儿简单做，把复杂的事情留给一般的通用机制来搞定吧。

async()可以启动一个新线程或者复用一个它认为合适的已有线程（非调用线程即可）(译注：语义上并发即可，不关心具体的调度策略。和 go 语义中的 goroutines 有点像)。后者从用户视角看更有效一些（只对简单任务而言）。

参考：

Standard: ???

Lawrence Crowl:

An Asynchronous Call for C++.

N2889 = 09-0079.

Herb Sutter :

A simple async()

N2901 = 09-0091 .

（翻译：interma）

atomic_operations

Stroustrup 尚未完成此主题，期待中

对此主题感兴趣的朋友，可以参考

C++小品：榨干性能：C++11中的原子操作（atomic operation）

VC11有点甜：原子操作和< atomic>头文件

Condition variables(条件变量)

Condition variables provide synchronization primitives used to block a thread until notified by some other thread that some condition is met or until a system time is reached.

Sorry, I have not had time to write this entry. Please come back later.

Stroustrup 先生尚未完成这一主题。

对这一主题感兴趣的朋友可以参考

C++小品：她来听我的演唱会——C++11中的随机数、线程(thread)、互斥(mutex)和条件变量(condition)

参考：

Standard: 30.5 Condition variables [thread.condition]

容器(Container)方面的改进

标准库中容器方面的改进

新的语言特性和近10年来的经验会给标准库中的容器带来啥改进呢？首先，新容器类型：array(大小固定容器)，forward_list（单向链表），unordered containers（哈希表，无序容器）。其次，新特性：initializer lists（初始化列表），rvalue references（右值引用），variadic templates（可变参数模板），constexpr（常量表达式）。下边均以 vector 为例加以介绍：

初始化列表（Initializer lists）：最显著的改进是容器的构造函数可以接受初始化列表来作为参数：

```
1    vector<string> vs = {"Hello",", ","World!","\n"};
2           for(auto s : vs ) cout << s;
```

move 操作：容器新增了 move 版的构造和赋值函数（作为传统 copy 操作的补充）。它最重要的内涵就是允许

我们高效的从函数中返回一个容器：

```
vector<int> make_random(intn)
        {
            vector<int> ref(n);
            // 产生0-255之间的随机数
                for(auto x& : ref) x = rand_int(0,255);

                returnref;
        }

        vector<int> v = make_random(10000);
        for(auto x : make_random(1000000)) cout << x <<'\n';
```

上边代码的关键点是 vector 没有被拷贝操作(译注：vector ref 的内存空间不是应该在函数返回时被 stack 自动回收吗？move assignment 通过右值引用精巧的搞定了这个问题)。对比我们现在的两种惯用法：在自由存储区来分配 vector 的空间，我们得负担上内存管理的问题了；通过参数传进已经分配好空间的 vector，我们得要写不太美观的代码了（同时也增加了出错的可能）。

改进的 push 操作：作为我最喜爱的容器操作函数，push_back()允许我们优雅的增大容器：

```
vector<pair<string,int>> vp;
string s;
inti;
while(cin>>s>>i) vp.push_back({s,i});
```

如上代码通过 r 和 i 构造了一个 pair 对象，然后将它 move 到 vp 中。注意这里是”move”而不是”copy”。这个 push_back 版本接受了一个右值引用参数，因此我们可以从 string 的移动构造函数（move constructor）（译注：直接由拷贝构造函数（copy ctor）对应而来）中获益。同时使用了[统一初始化语法]（unified initializer syntax）来避免哆嗦。

原地安置操作（Emplace operations）：在大多数情况下，push_back()使用移动构造函数（而不是拷贝构造函数）来保证它更有效率，不过在极端情况下我们可以走的更远。为何一定要进行拷贝/移动操作？为什么不能在 vector 中分配好空间，然后直接在这个空间上构造我们需要的对象呢？做这种事儿的操作被叫做”原地安置”（emplace，含义是：putting in place）。举一个 emplace_back()的例子：

```
vector<pair<string,int>> vp;
string s;
```

```
3    inti;
4    while(cin>>s>>i) vp.emplace_back(s,i);
```

emplace_back()接受了可变参数模板变量并通过它来构造所需类型。至于 emplace_back()是否比 push_back()更有效率，取决于它和可变参数模板的具体实现。如果你认为这是一个重要的问题，那就实际测试一下。否则，就从美感上来选择它们吧。到目前为止，我更喜欢 push_back()，不过只是目前哦。

Scoped allocators：现在容器中可以持有”拥有状态的空间分配对象(allocationobjects)”了，并通过它来进行”nested/scoped”方式的空间分配（译注：原文：use those to control nested/scoped allocation ）（举例：为容器中的元素分配空间）。

显然，容器不是唯一从新语言特性中获益的标准库部分：

编译期计算（Compile-time evaluation）：常量表达式

为 bitset, duration, char_traits, array, atomic types,

random numbers,

complex 等类型引入了编译期计算。对于某些情况，这意味着性能上的改善；而对于其它情况（无法编译期优化的情况下），则意味着可以减少晦涩代码和宏的使用了。

元组（Tuples）：如果没有可变参数模板

，它就不存在了。

（翻译：interma）

std::function 和 std::bind

标准库函数 bind()和 function()定义于头文件<functional>中（该头文件还包括许多其他函数对象），用于处理函数及函数参数。bind()接受一个函数（或者函数对象，或者任何你可以通过”(...)”符号调用的事物），生成一个其有某一个或多个函数参数被“绑定”或重新组织的函数对象。（译注：顾名思义，bind()函数的意义

就像它的函数名一样，是用来绑定函数调用的某些参数的。）例如：

```
intf(int,char,double);
// 绑定 f()函数调用的第二个和第三个参数，
// 返回一个新的函数对象为 ff，它只带有一个 int 类型的参数
auto ff = bind(f, _1, ‘c’, 1.2);
intx = ff(7);              //  f(7, ‘c’, 1.2);
```

参数的绑定通常称为”Currying”（译注：Currying—“烹制咖喱烧菜”，此处意指对函数或函数对象进行加工修饰操作），“_1”是一个占位符对象，用于表示当函数 f 通过函数 ff 进行调用时，函数 ff 的第一个参数在函数 f 的参数列表中的位置。第一个参数称为”_1”，第二个参数为”_2”，依此类推。例如：

```
intf(int,char,double);
auto frev = bind(f, _3, _2, _1);       // 翻转参数顺序
intx = frev(1.2, ‘c’, 7);          // f(7, ‘c’, 1.2);
```

此处，auto 关键字节约了我们去推断 bind 返回的结果类型的工作。

我们无法使用 bind()绑定一个重载函数的参数，我们必须显式地指出需要绑定的重载函数的版本：

```
intg(int);
doubleg(double);

auto g1 = bind(g, _1);   // 错误：调用哪一个 g() ?
// 正确，但是相当丑陋
auto g2 = bind( (double(*)(double))g, _1);
```

bind()有两种版本：一个如上所述，另一个则是“历史遗留”的版本：你可以显式地描述返回类型。例如：

```
auto f2 = bind<int> (f, 7, ‘c’, _1);     // 显式返回类型
intx = f2(1.2);                  // f(7, ‘c’, 1.2);
```

第二种形式的存在是必要的，并且因为第一个版本（(?) “and for a user simplest “，此处请参考原文)）无法在 C++98中实现。所以第二个版本已经被广泛使用。

function 是一个拥有任何可以以”(...)”符号进行调用的值的类型。特别地，bind 的返回结果可以赋值给 function 类型。function 十分易于使用。（译注：更直观地，可以把 function 看成是一种表示函数的数据类型，就像函数对象一样。只不过普通的数据类型表示的是数据，function 表示的是函数这个抽象概念。）例如：

```
// 构造一个函数对象，
```

```
2    // 它能表示的是一个返回值为 float，
3    // 两个参数为 int，int 的函数
4    function<float(intx,inty)> f;
5
6    // 构造一个可以使用"()"进行调用的函数对象类型
7    structint_div {
8           floatoperator() (intx,inty)const
9                {return((float)x)/y; };
10   };
11
12   f = int_div();                    // 赋值
13   cout<< f(5,3) <<endl; // 通过函数对象进行调用
14   std::accumulate(b, e, 1, f);       // 完美传递
```

成员函数可被看做是带有额外参数的自由函数：

```
1    structX {
2           intfoo(int);
3    };
4
5    // 所谓的额外参数，
6    // 就是成员函数默认的第一个参数，
7    // 也就是指向调用成员函数的对象的 this 指针
8    function<int(X*,int)> f;
9    f = &X::foo;           // 指向成员函数
10
11   X x;
12   intv = f(&x, 5); // 在对象 x 上用参数5调用 X::foo()
13   function<int(int)> ff = std::bind(f, &x, _1);   // f 的第一个参数是&x
14   v = ff(5);              // 调用 x.foo(5)
```

function 对于回调函数、将操作作为参数传递等十分有用。它可以看做是 C++98 标准库中函数对象 mem_fun_t, pointer_to_unary_function 等的替代品。同样的，bind()也可以被看做是 bind1st()和 bind2nd()的替代品，当然比他们更强大更灵活。

参考：

Standard: 20.7.12 Function template bind, 20.7.16.2 Class template function

Herb Sutter:Generalized Function Pointers

.

August 2003.

Douglas Gregor:

Boost.Function

.

Boost::bind

（翻译：dabaitu）

forward_list – a singly-liked list

std::forward_list 是一个基本的单向链表。它只提供了前向迭代器（forward iteration）；并在执行插入/删除操作后，其他节点也不会受到影响（译注：其它迭代器不失效）。它尽可能减少所占用空间的大小（空链表很可能只占用一个 word2（译注：2Byte））且不提供 size()操作（所以也没有存储 size 的数据成员），简略原型如下：

```
template<ValueType T, Allocator Alloc = allocator<T> >
        requires NothrowDestructible<T>
    classforward_list {
    public:
        // the usual container stuff
        // no size()
        // no reverse iteration
        // no back() or push_back()
    };
```

参看:

Standard: 23.3.3 Class template forward_list

（翻译：interma）

future and promise

并行开发挺复杂的，特别是在试图用好线程和锁的过程中。如果要用到条件变量或 std-atomics（一种无锁开发方式），那就更复杂了。C++0x 提供了 future 和 promise 来简化任务线程间的返回值操作；同时为启动任务线程提供了 packaged_task 以方便操作。其中的关键点是允许2个任务间使用无（显式）锁的方式进行值传递；标准库帮你高效的做好这些了。基本思路很简单：当一个任务需要向父线程（启动它的线程）返回值时，它把这个值放到 promise 中。之后，这个返回值会出现在和此 promise 关联的 future 中。于是父线程就能读到返回值。更简单点的方法，参看 async()。

标准库中提供了3种 future：普通 future 和为复杂场合使用的 shared_future 和 atomic_future。在本主题中，只展示了普通 future，它已经完全够用了。如果我们有一个 future

f，通过 get()可以获得它的值：

```
1   X v = f.get(); // if necessary wait for the value to get computed
```

如果它的返回值还没有到达，调用线程会进行阻塞等待。要是等啊等啊，等到花儿也谢了的话，get()会抛出异常的（从标准库或等待的线程那个线程中抛出）。

如果我们不需要等待返回值（非阻塞方式），可以简单询问一下 future，看返回值是否已经到达：

```
1   if(f.wait_for(0))
2   {
3       // there is a value to get()
4       // do something
5   }
6   else
7   {
8       // do something else
9   }
```

但是，future 最主要的目的还是提供一个简单的获取返回值的方法：get()。

promise 的主要目的是提供一个”put”（或”get” ,随你)操作 ,以和 future 的 get()对应。future 和 promise

的名字是有历史来历的，是一个双关语。感觉有点别扭？请别怪我。

promise 为 future 传递的结果类型有2种：传一个普通值或者抛出一个异常

```
try{
        X res;
        // compute a value for res
        p.set_value(res);
}
catch(…) {  // oops: couldn’t compute res
        p.set_exception(std::current_exception());
}
```

到目前为止还不错，不过我们如何匹配 future/promise 对呢？一个在我的线程，另一个在别的啥线程中吗？

是这样：既然 future 和 promise 可以被到处移动（不是拷贝），那么可能性就挺多的。最普遍的情况是父子线程配对形式，父线程用 future 获取子线程 promise 返回的值。在这种情况下，使用 async()是很优雅的方法。

packaged_task 提供了启动任务线程的简单方法。特别是它处理好了 future 和 promise 的关联关系，同时提供了包装代码以保证返回值/异常可以放到 promise 中，示例代码：

```
voidcomp(vector& v)
{
        // package the tasks:
        // (the task here is the standard
        //  accumulate() for an array of doubles):
        packaged_task pt0{std::accumulate};
        packaged_task pt1{std::accumulate};

        auto f0 = pt0.get_future();    // get hold of the futures
        auto f1 = pt1.get_future();

        pt0(&v[0],&v[v.size()/2],0);   // start the threads
        pt1(&[v.size()/2],&v[size()],0);

        returnf0.get()+f1.get();      // get the results
}
```

参看：

Standard: 30.6 Futures [futures]

Anthony Williams:

Moving Futures – Proposed Wording for UK comments 335, 336, 337 and 338.

N2888==09-0078.

Detlef Vollmann, Howard Hinnant, and Anthony Williams

An Asynchronous Future Value (revised)

N2627=08-0137.

Howard E. Hinnant:

Multithreading API for C++0X – A Layered Approach.

N2094=06-0164. The original proposal for a complete threading package..

（翻译：interma）

垃圾回收（应用程序二进制接口）

在 C++中，垃圾回收机制（自动回收没有被引用的内存区域）是可选的；也就是说在编译器中并不是一定要实现垃圾回收器。尽管如此，C++0x 还是定义了垃圾回收器的功能。与此同时，C++0x 还提供了应用程序二进制接口（ABI: Application Binary Interface）来辅助控制垃圾回收器的行为。

我们用“safely derived pointer”（3.7.3.3）（译注：我搜索后发现，是在3.7.3.3而不是3.7.4.3讲了 safely derived pointer。这里可能是原文作者的笔误）来表示指针和生存时间的规则；粗略地说就是 “指向由 new 分配的对象或者指向相应的子对象的指针”。 下面是一些关于“not safely derived pointers”又名 “disguised pointers”，或者说是为便于正常人理解和认可你的程序从而你应该注意的一些问题。

将指针暂时指到别处：

```
1    int* p =newint;
2    p+=10;
3    //…垃圾回收器可能在这里运行…
```

```
4    p-=10;
5    //在此，我们是否可以肯定开始为 p 分配的那块 int 内存还存在？
6    *p = 10;
```

将指针隐藏到一个 int 变量中：

```
1    int* p =newint;
2    intx =reinterpret_cast(p);// non-portable (不可移植)
3    p=0;
4    //…垃圾回收器可能在这里运行…
5    p =reinterpret_cast(x);
6    //在此，我们是否可以肯定开始为 p 分配的那块 int 内存还存在？
7    *p = 10;
```

还有更多甚至更危险的陷阱。比如 I/O，以及将存储不同的数据位打散，..

虽然我们也有一些合理的理由来伪装指针（例如：xor 有可能在 exceptionally memory-constrained applications 中导致错误），但是理由并不像一些程序员所认为的那么多。

程序员可以在代码中声明哪些地方不会有指针被发现（比如在一个图像中），也可以声明哪些内存区域不能被回收，即使垃圾回收器发现没有任何指针指向这块内存区域。以下是相关的例子：

void declare_reachable(void* p); //以 p 起始的内存区域

//（用能够记住所分配内存区域大小

//的内存分配操作符分配）不能被回收

```
1    template<classT> T* undeclared_reachable(T* p);
2
3     voiddeclare_no_pointers(char* p,size_tn);       //p[0..n] 中没有指针
4     voidundeclared_no_poitners(char* p,size_tn);
```

程序员要能够查询到关于指针安全和回收的规则是否是强制性的：

```
1    enumclasspointer_saftety {relaxed, preferred, strict};
2           pointer_safety get_pointer_safety();
```

3.7.4.3[4]：满足以下三个条件的行为没有被定义：如果一个非”safely-derived pointer”值被释放，并且它所引用的对象是动态存储期（由 new 操作符动态创建并由 delete 销毁的对象拥有动态存储期），与此同时这个指针之前也没被声明为可到达的（20.7.13.7）。

relaxed: safely-derived pointer 和 not safely-derived pointer 被认为是等同的。这和 C 以及 C++98中是一样的。但这并不是我的初衷。我的想法是用户如果没有使用有效的指针指向一个对象则应启用垃圾回收。

Preferred：和 relaxed 类型一样。只不过垃圾回收器可能被用作内存泄露检测以及（或者）检测对象是否被一个错误的指针解引用。

strict: safely-derived 和 not safely-derive 这两种指针可能不再被等同。也就是说，垃圾回收器可能被启用而且将会忽略那些 not safely derived pointer。

并不存在任何标准化的方法以供你选择任何一种。这是一个实现的质量（quality of implementation）和编程环境（programming environment）的问题。

另外，可以参考以下文献：

the C++ draft 3.7.4.3

the C++ draft 20.7.13.7

Hans Boehm’s

GC page

Hans Boehm’s

Discussion of Conservative GC

final proposal

Michael Spertus and Hans J. Boehm:

The Status of Garbage Collection in C++0X

.

ACM ISMM’09.

（翻译：Yibo Zhu）

无序容器（Unordered containers）

一个无序容器实际上就是某种形式的蛤希表。C++0x 提供四种标准的无序容器：

unordered_map

unordered_set

unordered_multimap

unordered_multiset

实际上，它们应该被称为 hash_map 等。但是因为有很多地方已经在使用 hash_map 这样的名字了，为了保证其兼容性，标准委员会不得不选择新的名字。而 unordered_map 是我们所能够找到的最好的名字了。无序（“unordered”）代表着 map 和 unordered_map 之间一个最本质的差别：当你使用 map 容器的时候，容器中的所有元素都是通过小于操作（默认情况下使用“<”操作符）排好序的，但是 unordered_map 并没有对元素进行排序，所以它并不要求元素具有小于操作符。并且，一个哈希表也并添言地提供排序的功能。相反，map 容器中的元素也并不要求具有哈希函数。

基本上，当代码的优化是可能的并且我们有理由对其进行优化时，我们可以把 unordered_map 当作一个优化之后的 map 容器来使用。例如：

```
1    map<string,int> m {
2        { “Dijkstra” ,1972}, { “Scott” ,1976},
3        { “Wilkes” ,1967}, { “Hamming” ,1968}
4    };
5    m["Ritchie"] = 1983;
6    for(auto x : m)
7        cout << ‘{ ‘ << x.first << ‘,’ << x.second << ‘}’ ;
8
9    // 使用优化之后的 unordered_map
10   unordered_map<string,int> um {
11       { “Dijkstra” ,1972}, { “Scott” ,1976},
12       { “Wilkes” ,1967}, { “Hamming” ,1968}
13   };
14   um["Ritchie"] = 1983;
```

```
15    for(auto x : um)
16        cout << '{' << x.first << ',' << x.second << '}';
```

map 容器 m 的迭代器将以字母的顺序访问容器中的所有元素，而 unordered_map 容器 um 则并不按照这样的顺序（除非通过一些特殊的操作）。m 和 um 两者的查找功能的实现机制是非常不同的。对于 m，它使用的是复杂度为 log2(m.size())的小于比较，而 um 只是简单地调用了一个哈希函数和一次或多次的相等比较。如果容器中的元素比较少（比如几打），很难说哪一种容器更快，但是对于大量数据（比如数千个）而言，unordered_map 容器的查找速度要比 map 容器快很多。

更多内容稍后提供。

参考：

• Standard: 23.5 Unordered associative containers.

(翻译：Yibo Zhu)

锁（locks）

锁是这样一个对象，它能够保持对一个 mutex 对象的引用并且可能会在自身销毁时（比如离开一个 block 域时）来对 mutex 对象进行解锁 unlock 操作。作为一种良好的异常处理习惯，可以在线程中使用锁来帮助管理 mutex 的拥有权。也就是说，锁为互斥实现了 RAII（Resource Acquisition Is Initialization）。下面是一个关于锁的用法的例子：

```
1     std::mutex m;
2     intsh; // 共享数据
3     // ...
4     voidf()
5     {
6     // ...
7     std::unique_lock lck(m);
8     // 对共享数据进行操作
9     sh+=1;
10    }
```

锁可以被移动（这是由设计锁的目的决定的：锁是用于在局部环境下表示对非局部资源的拥有权），但是不可以被复制（这是因为无法确定哪一个副本应该拥有目前的资源）。

可以使用 unique_lock 来很直接明了地给出关于锁的描述。unique_lock 能够更加安全的执行任何 mutex 所能达到的功能。如下例所示，我们可以用一个 lock 对象进行 try lock 操作：

```
1    std::mutex m;
2    intsh; // 共享数据
3    // ...
4    voidf()
5    {
6    // ...
7    std::unique_lock lck(m,std::defer_lock);//对 m 使用锁，但是没有获取 mutex
8    // ...
9    if(lck.try_lock()) {
10   // 操作共享数据
11   sh+=1;
12   }
13   else{
14   // 可能做一些其它操作
15   }
16   }
```

类似地，unique_lock 也支持 try_lock_for()以及 try_lock_until()。与使用 mutex 相比，使用锁可以提供异常处理并且能够在忘记 unlock()时提供相应的保护。在并发编程中，我们可以通过锁来得到所有需要的功能。

现在考虑一个新的问题：如果我们需要两个分别用一个 mutex 表示的资源，应该如何实现？一种最简单的办法就是如下例所示的那样依次获取这两个 mutex：

```
1    std::mutex m1;
2    std::mutex m2;
3    intsh1; // 共享数据
4    intsh2
5    // ...
6    voidf()
7    {
8    // ...
9    std::unique_lock lck1(m1);
10   std::unique_lock lck2(m2);
```

```
11    //操作共享数据
12    sh1+=sh2;
13    }
```

然而，上例中的方法有着一个致命缺陷：如果其它线程试图以相反的次序来获取 m1和 m2，便会出现死锁现象。对于一个含有很多锁的系统来说，这一做法相当有风险。为了解决这类问题，锁提供了两个方法来安全地尝试获取两个或者两个以上的锁。下面是一个相应的例子：

```
1    voidf()
2    {
3    // ...
4    std::unique_lock lck1(m1,std::defer_lock); //使用锁但并不试图获取 mutex
5    std::unique_lock lck2(m2,std::defer_lock);
6    std::unique_lock lck3(m3,std::defer_lock);
7    lock(lck1,lck2,lck3);
8    // 操作共享数据
9    }
```

很明显，必须非常并且精巧地设计上例中的 lock()才能避免死锁。从本质上来讲，它和小心使用 try_lock()s 所达到的效果是一样的。当 lock()没有成功获取所有所锁时，它会抛出一个异常。实际上，由于 lock()和 try_lock(), unlock()接受任何参数，所以我们无法分清 lock()到底抛出了哪个异常。这依赖于它的参数。

如果你倾向于自己使用 try_lock()来实现上例所示的相应功能，下面的例子可能对你有一定的帮助：

```
1     voidf()
2     {
3     // ...
4     std::unique_lock lck1(m1,std::defer_lock); // 使用锁但是并不试图去获取 mutex
5     std::unique_lock lck2(m2,std::defer_lock);
6     std::unique_lock lck3(m3,std::defer_lock);
7     intx;
8     if((x = try_lock(lck1,lck2,lck3))==-1) { // 欢迎来到 C 的地界
9     // 操作共享数据
10    }
11    else{
12    // x 拥有正在拥有一个 mutex，因此我们无法获取
13    // 比如，如果 lck2.try_lock()失效了，x 的值就等于1
14    }
15    }
```

同时可参考：

Standard: 30.4.3 Locks [thread.lock]

（翻译：Yibo Zhu）

metaprogramming（元编程） and type traits

m etaprogramming and type traits

（译注：原作者还没有完成这个小节，等原作者完成后中文版随后奉上。这里补充一点关于元编程(metaprogramming)）的基础知识，供大家参考

1、何谓“元编程(Metaprogramming)”？

具备如下特征之一的程序编写称为元编程：

利用或者编写其它语言程序来作为所编写程序的数据

在程序运行时完成一些编程工作，而不是在编译时—具备这种特征的语言，我们也常成为动态语言

2、元编程的优势

元编程为程序开发人员提供了在与手动编写所有代码相同时间内，完成更多工作的可能，同时，由于

在面对新情况时，不需要重新编译，因此元编程为程序提供了更大的灵活性和可用性。

3、元编程语言

用于元编程的语言，称作元语言，被元语言所利用的语言通常称作对象语言。元语言具有自反性。所谓

自反性就是自己描述自己的特性。自反性是元编程中非常重要特点。有些语言中 ,将自身作为 First-class object

也是非常有用的。对于有些可以调用元编成机制的程序语言，也具备自反性。

4、元编程方式

一般来说，实现元编程有两种方式。

第一种方法是调用 API 将运行时引擎中的代码展示为程序代码

第二种方法是动态执行包含在程序中的字符表达式，也就是我们常说的“程序生成程序”。

虽然，两种方法都可以实现元编程，但是大多数语言都只是支持其中的一种方法。

5、实例

1）bash script

#!/bin/bash # metaprogram echo ‘#!/bin/bash’ >program for ((I=1; I< =992; I++)) do echo "echo $I" >>program done chmod +x program

该脚本产生993行代码，这便是用程序生成程序的一个例子

2）像 Lisp， Python 和 Javascript 等语言可以在程序运行期间修改或者增量编译，这些程序语言也可以在不产生新代码的

情况下进行元编程。

3）我们常见的编译器其实也是元编码的代表，编译器通常是用相对简约的高级语言来产生汇编或者机器代码。

（翻译：Yibo Zhu）

互斥

互斥是多线程系统中用于控制访问的一个原对象（primitive object）。下面的例子给出了它最基本的用法：

```
1   std::mutex  m;
2   intsh;//共享数据
3   // …
4   m.lock();
5   // 对共享数据进行操作：
6   sh += 1;
7   m.unlock();
```

在任何时刻，最多只能有一个线程执行到 lock()和 unlock()之间的区域（通常称为临界区）。当第一个线程正在临界区执行时，后续执行到 m.lock()的线程将会被阻塞直到第一个进程执行到 m.unlock()。这个过程比较简单，

但是如何正确使用互斥并不简单。错误地使用互斥将会导致一系列严重后果。大家可以设想以下情形所导致的后果：一个线程只进行了 lock()而没有执行相应 unlock()； 一个线程对同一个 mutex 对象执行了两次 lock()操作；一个线程在等待 unlock()操作时被阻塞了很久；一个线程需要对两个 mutex 对象执行 lock()操作后才能执行后续任务。可以在很多书（译者注：通常操作系统相关书籍中会讲到）中找到这些问题的答案。在这里（包括 Locks section 一节）所给出的都是一些入门级别的。

除了 lock()，mutex 还提供了 try_lock()操作。线程可以借助该操作来尝试进入临界区，这样一来该线程不会在失败的情况下被阻塞。下面例子给出了 try_lock()的用法：

```
1    std::mutex m;
2    intsh;//共享数据
3    // …
4    if(m.try_lock()) {
5        //操作共享数据
6        sh += 1;
7        m.unlock();
8    }
9    else{
10       //可能在试图进入临界区失败后执行其它代码
11   }
```

recursive_mutex 是一种能够被同一线程连续锁定多次的 mutex。下面是 recursive_mutex 的一个实例：

```
1    std::recursive_mutex m;
2     intsh;//共享数据
3     //..
4     voidf(inti)
5     {
6         //…
7         m.lock();
8         //对共享数据进行操作
9         sh += 1;
10        if(－i>0) f(i); //注意：这里对 f(i)进行了递归调用，
11        //将导致在 m.unlock()之前多次执行 m.lock()
12        m.unlock();
13        //…
14    }
```

对于这点，我曾经夸耀过并且用 f()调用它自身。一般地，代码会更加微妙。这是因为代码中经常会有间接递归

调用。比如 f()调用 g()，而 g()又调用了 h()，最后 h()又调用了 f()，这样就形成了一个间接递归。

如果我想在未来的10秒内进入到一个 mutex 所划定的临界区，该如果实现？ timed_mutex 类可以解决这个问题。事实上，关于它的使用可以被看做是关联了时间限制的 try_lock()的一个特例。

```
1    std::timed_mutex m;
2    intsh;//共享数据
3    //…
4    if( m.try_lock_for(std::chrono::seconds(10))) {
5    //对共享数据进行操作
6    sh += 1;
7    m.unlock();
8    }
9    else{
10       //进入临界区失败，在此执行其它代码
11   }
```

try_lock_for()的参数是一个用相对时间表示的 duration。如果你不想这么做而是想等到一个固定的时间点：一个 time_point，你可以使用 try_lock_until()：

```
1    std::timed_mutex m;
2    intsh;//共享数据
3    // …
4    if( m.try_lock_until(midnight)) {
5    //对共享数据进行操作
6    sh += 1;
7    m.unlock();
8    }
9    else{
10       //进入临界区失败，在此执行其它代码
11   }
```

这里使用 midnight 是一个冷笑话：对于 mutex 级别的操作，相应的时间是毫秒级别的而不是小时。

当然地，C++0x 中也有 recursive_timed_mutex。

mutex 可以被看做是一个资源（因为它经常被用来代表一种真实的资源），并且当它对至少两个线程可见时它才是有用的。必然地，mutex 不能被复制或者移动（正如你不能复制一个硬件的输入寄存器）。

令人惊讶地，实际中经常很难做到 lock()s 与 unlock()s 的匹配。设想一下那些复杂的控制结构，错误以及异常，

要做到匹配的确比较困难。如果你可以选择使用 locks 去管理你的互斥，这将为你和你的用户节省大量的时间，再也不用熬夜通宵彻夜无眠了。（that will save you and your users a lot of sleep？？）。

同时可参考：

Standard: 30.4 Mutual exclusion [thread.mutex]

H. Hinnant, L. Crowl, B. Dawes, A. Williams, J. Garland, et al.:

Multi-threading Library for Standard C++ (Revision 1)

???

（翻译：Yibo Zhu）

随机数的产生

随机数有着广泛的用途。比如测试、游戏、仿真以及安全等领域都需要用到随机数。标准库所提供的多种可供选择的随机数产生器也恰恰反应了随机数应用范 围的多样性。随机数产生器由引擎（engine）和分布（distribution）两部分组成。其中，engine 用于产生一个随机数序列或者伪随机数 序列；distribution 则将这些数值映射到位于固定范围的某一数学分布中。关于分布的例子有：unifrom_int (所有的整数倍都被以相等的概率产生)以及 normal_distribution (分布的概率密度函数曲线呈钟形)；每一种分布都处于某一特定的范围之内。例如：

```
1    //distribution 将产生的随机数映射到整数1..6
2    uniform_int_distribution<int> one_to_six {1,6};
3
4    default_random_engine re {};        //默认的 engine
```

如果想获得一个随机数，你可以用一个随机引擎为参数调用 distribution 来产生一个随机数：

```
1    intx = one_to_six(re);// x 是 [1:6]这个范围内的一个随机数
```

在每次调用的时候都需要提供一个引擎作为参数非常繁琐，所以我们可将引擎和 distribution 邦定成一个函数

对象，然后直接通过这个函数对象的调用来产生随机数，而不用每次调用都提供参数了。

```
auto dice {bind(one_to_six,re)};//  产生一个新的随机数生成器
intx = dice();// 调用 dice 函数对象，x 是一个分布在 [1:6]范围的随机数
```

多亏在设计它是对一般性和性能的关注。在这方面，一位专家曾在评价标准库中随机数模块时说道：“在扩充的过程中，每一个随机数库想变成什么”。然而，它很 难真正让一个新手感觉到容易上手。在性能方面，我从没有见过随机数的接口成为性能的瓶颈。另外 我也一直会使用一个简单的随机数生器来教新手(具有一定的 基础)。下面的就是这样的一个可以说明问题的例子。

```
intrand_int(intlow, high);  //按照均匀分布在区间[low: high]中产生一个随机数
```

然而我们如何实现 rand_int() ? 我们必须在 rand_int()中使用 dice()之类的函数：

```
intrand_int(intlow,inthigh)
{
     staticdefault_random_engine re {};
    usingDist = uniform_int_distribution<int>;
    staticDist uid {};
    returnuid(re, Dist::param_type{low,high});
}
```

关于 rand_int()的定义依然是属于“专家级”的，但是应该把关于它的使用安排在 C++课程的第一周。

在这里，我们举一个不太琐碎的关于随机数生成器的例子。这个例子中代码的功能是生成和打印一个正态分布。

```
        default_random_engine re;  //
 默认引擎
    normal_distribution<double> nd(31/* mean */,
          8/* sigma */);

    auto norm = std::bind(nd, re);

    vector<int> mn(64);

    intmain()
    {
        for(inti = 0; i<1200; ++i)
                ++mn[round(norm())];// 产生随机数


        for(inti = 0; i<mn.size(); ++i)
```

```
17                 {
18             cout << i <<'\t';
19             for(intj=0; j<mn[i]; ++j)
20                         cout <<'*';
21
22                     cout <<'\n';
23         }
24     }
```

我运行了一个支持 boost::random 的版本并把它编辑到 C++0x 中，然后得到了下面的结果。

```
0

1

2

3

4 *

5

6

7

8

9 *

10 ***

11 ***

12 ***

13 *****

14 *******

15 ****

16 **********
```

17 **********

18 **********

19 **********

20 **********

21 **********

22 **********

23 **********

24 **********

25 **********

26 **********

27 **********

28 **********

29 **********

30 **********

31 **********

32 **********

33 **********

34 **********

35 **********

36 **********

37 **********

38 **********

```
39 ********************************
40 ******************************************
41 ***********************
42 *************************
43 ******************************
44 ******************
45 **************
46 *********
47 ********
48 *****
49 *****
50 ****
51 ***
52 ***
53 **
54 *
55 *
56
57 *
58
59
60
```

61

62

63

另外，可以参考以下文献：

Standard 26.5: Random number generation

正则表达式（Regular expressions）

抱歉，本主题尚未完成，请稍后再来。

（翻译：Yibo Zhu）

具有作用域的内存分配器

为了使容器对象小巧和简单起见，C++98没有要求容器支持具有状态的内存分配器，即不用把分配器对象存储在容器对象中。在 C++11中，这仍然默认做法。但是，在 C++0x 中，也可以使用具有状态的内存分配器：这种内存分配器拥有一个指向分配区域的指针。例如：

```
1    template<classT>classSimple_alloc {  // C++98 style
2          // no data
3          // usual allocator stuff
4       };
5
6       classArena {
7          void* p;
8          ints;
9       public:
10         Arena(void* pp,intss);
11         // allocate from p[0..ss-1]
12      };
13
```

```
14      template<classT>structMy_alloc {
15          Arena& a;
16          My_alloc(Arena& aa) : a(aa) { }
17          // usual allocator stuff
18      };
19
20      Arena my_arena1(newchar[100000],100000);
21      Arena my_arena2(newchar[1000000],1000000);
22
23      vector<int> v0;  // allocate using default allocator
24
25      vector<int,My_alloc<int>> v1(My_alloc<int>{my_arena1});  // allocate from my_arena1
26
27      vector<int,My_alloc<int>> v2(My_alloc<int>{my_arena2});  // allocate from my_arena2
28
29      vector<int,Simple_alloc<int>> v3;  // allocate using Simple_alloc
```

通常我们可以使用 typedef 来简化上述例子中繁冗的表达。

虽然并不能保证默认内存分配器和 Simple_alloc 不在一个 vector 对象中占用空间，但是在实现库的时候可以使用一些模板的元程序设计方法来做到这点。因此，如果对象拥有状态的话（比如 My_alloc），内存分配器将会带来一定的空间开销。

在同时使用容器和用户自定义内存分配器时存在一个更为隐蔽的问题：一个内存分配器成员是否应该和它的容器处于相同的分配区域中？例如，你使用

Your_allocator 来给 Your_string 的成员分配内存，而我用 My_allocator 来给 My_vector 的成员分配内存。在这种

情况下，My_vector 会使用哪个分配器。要解决这问题的话就需要告诉容器使用什么分配器来传递成员。下面的例子将给出实现这一点的方法。在这个例子

中我将使用分配器 My_alloc 来分配 vector 成员和 string 成员。首先，我必须要有一个能够接受 My_alloc 对象的 string 版本：

```
1   //使用 My_alloc 的一个 string 类型
2   usingxstring = basic_string<char, char_traits<char>, My_alloc<char>>;
```

然后，我要有一个能够接受这种 string 类型以及 My_alloc 对象的 vector 版本。同时，该 vector 版本需要能够把 My_alloc 对象传递给 string。

```
1    usingsvec =  vector<xstring,scoped_allocator_adaptor<My_alloc<xstring>>>;
```

最后，我们可以按照下面的示例实现 My_alloc 类型的分配器：

```
1    svec v(svec::allocator_type(My_alloc{my_arena1}));
```

在此，svec 是一个成员为 string 类型的 vector，并且该 vector 使用 My_alloc 来为其 string 类型的成员分配内存。另外，标准

库中新提供了”adaptor”（“wrapper type”）

scoped_allocator_adaptor。它可以用来指明使用 My_alloc 来为 string 类型变量分配内存。需要注意的是，scoped_allocator_adaptor 可以将 My_alloc 转换为 My_alloc。而这正是 xstring 所需要提供的功能。

于是，我们又有了4种可用于替换的方法：

```
1    // vector and string use their own (the default) allocator:
2    usingsvec0 = vector<string>;
3    svec0 v0;
4
5    // vector (only) uses My_alloc and string uses its own (the default) allocator:
6    usingsvec1 = vector<string,My_alloc<string>>;
7    svec1 v1(My_alloc<string>{my_arena1});
8
9    // vector and string use My_alloc (as above):
10   usingxstring = basic_string<char, char_traits<char>, My_alloc<char>>;
11       usingsvec2 = vector<xstring,scoped_allocator_adaptor<My_alloc<xstring>>>;
12       svec2 v2(scoped_allocator_adaptor<My_alloc<xstring>>{my_arena1});
13
14   // vector uses My_alloc and string uses My_string_alloc:
15   usingxstring2 = basic_string<char, char_traits<char>, My_string_alloc<char>>;
16   usingsvec3 = vector<xstring2,scoped_allocator_adaptor<My_alloc<xstring>, My_string_alloc<c
17   svec3 v3(scoped_allocator_adaptor<My_alloc<xstring2>, My_string_alloc<char>>{my_arena1,my_
```

很明显，第一种方法 svec0是最常用的。但是在内存会严重影响性能的系统中，其它版本（特别是 svec2）将会显得尤为重要。当然了，我们可以使用一些

typedef 来增强代码的可读性。不过还好，毕竟你不用每天都写这些。另外，我们提供了

scoped_allocator_adaptor2的一个变

种：scoped_allocator_adaptor2。它用于两个非默认内存分配器不一样的情况。

同时可参考：

Standard: 20.8.5 Scoped allocator adaptor [allocator.adaptor]

Pablo Halpern:

The Scoped Allocator Model (Rev 2).

N2554=08-0064.

（翻译：Yibo Zhu）

共享资源的智能指针 shared_ptr

shared_ptr 被用来表示共享的拥有权。也就是说，当两段代码都需要访问一些数据，而它们又都没有独占该数据的所有权（从某种意义上来说就是该段代码负责销毁该对象）。这是我们就需要 shared_ptr。shared_ptr 是一种计数指针。当引用计数变为0时，shared_ptr 所指向的对象就会被删除。下面我们用一段代码来说明这点。

```
1    voidtest()
2    {
3        shared_ptr p1(newint);   // 计数是1
4        {
5            shared_ptr p2(p1);   //计数是2
6            {
7                shared_ptr p3(p1);   // 计数是3
8            }   //计数变为2
9        }//计数变为1
10   }   // 在此，计数变为0。同时 int 对象被删除
```

现在来看一个更为实际的例子。在这个例子中，我们用指针指向图中的节点。一个需要解决的问题是当从一个节点上移除一个指针时并不知道时候还有其它指针指向这个节点。如果节点能够拥有一些资源，从而需要析构

器采取一些行动（一个典型的例子就是文件句柄。当节点被检测到时，文件句柄相应的文件就会被关闭）。这样以来，通过 shared_ptr 就可以解决这个问题。你可以认为使用 shared_ptr 的目的和你使用垃圾回收器的目的是一样的。只是出处于经济性的考虑，你没有足够的垃圾，或者执行环境不允许那么做，或者所管理的资源不仅仅是内存（如文件句柄）。例如：

```
1   structNode {   // 注意：其它的节点也可能指向该节点
2       shared_ptr left;
3       shared_ptr right;
4       File_handle f;
5       // …
6   };
```

这里 Node 的析构器（隐式的析构器即可）删除了它的子节点。也就是说 Node 的析构器调用了 left 和 right 的析构器。因为 left 是一个 shared_ptr，所以当 left 是最后一个指向该 Node 的指针时，该节点将会被删除。处理 right 的方式和 left 的类似。f 的析构器将按照 f 的要求执行。

需要注意的是，当仅需要将一个指针从一个拥有者传个另一个时，你不应使用 shared_ptr。这是 unique_ptr 的用途。unique_ptr 会以更为小的开销来更好的实现这个功能。如果你曾经使用计数指针作为工厂函数的返回值或者类似的情形，可以考虑升级使用 unique_ptr 而不是 shared_ptr。

另外，不要不加思考地把指针替换为 shared_ptr 来防止内存泄露。shared_ptr 并不是万能的，而且使用它们的话也是需要一定的开销的：

环状的链式结构 shared_ptr 将会导致内存泄露（你需要一些逻辑上的复杂化来打破这个环。比如使用 weak_ptr）。

共享拥有权的对象一般比限定作用域的对象生存更久。从而将导致更高的平均资源使用时间。

在多线程环境中使用共享指针的代价非常大。这是因为你需要避免关于引用计数的数据竞争。

共享对象的析构器不会在预期的时间执行。

与非共享对象相比，在更新任何共享对象时，更容易犯算法或者逻辑上的错误。

shared_ptr 用于表示共享拥有权。然而共享拥有权并不是我的初衷。在我看来，一个更好的办法是为对象指明拥有者并且为对象定义一个可以预测的生存范围。

同时可参考：

the C++ draft: Shared_ptr (20.7.13.3)

（翻译：Yibo Zhu）

smart pointers 请参考 shared_ptr, weak_ptr, 和 unique_ptr

线程（thread）

线程（译注：大约是 C++11中最激动人心的特性了）是一种对程序中的执行或者计算的表述。跟许多现代计算一样，C++11中的线程之间能够共享地址空间。从这点上来看，它不同于进程：进程一般不会直接跟其它进程共享数据。在过去，C++针对不同的硬件和操作系统有着不同的线程实现版本。如今，C++将线程加入到了标准件库中：一个标准线程 ABI。

许多大部头书籍以及成千上万的论文都曾涉及到并发、并行以及线程。在这一条 FAQ 里几乎不涉及这些内容。事实上，要做到清楚地思考并发非常难。如果你想编写并发程序，请至少看一本书。不要依赖于一本手册、一个标准或者一条 FAQ。

在用一个函数或者函数对象（包括 lambda）构造 std::thread 时，一个线程便启动了。

```
1    #include <thread>
2
3        voidf();
4
5        structF {
6            voidoperator()();
```

```
7       };
8
9       intmain()
10      {
11          std::threadt1{f};   // f() 在一个单独的线程中执行
12          std::threadt2{F()};   // F()() 在一个单独的线程中执行
13      }
```

然而，无论 f()和 F()执行任何功能，都不能给出有用的结果。这是因为程序可能会在 t1执行 f()之前或之后以及

t2执行 F()之前或之后终结。我们所期望的是能够等到两个任务都完成，这可以通过下述方法来实现：

```
1   intmain()
2   {
3           std::threadt1{f};   // f() 在一个单独的线程中执行
4           std::threadt2{F()};   // F()()在一个单独的线程中执行
5
6       t1.join();   // 等待 t1
7       t2.join();   // 等待 t2
8   }
```

上面例子中的 join()保证了在 t1和 t2完成后程序才会终结。这里”join”的意思是等待线程返回后再终结。

通常我们需要传递一些参数给要执行的任务。例如：

```
1    voidf(vector<double>&);
2
3    structF {
4    vector<double>& v;
5    F(vector<double>& vv) :v{vv} { }
6    voidoperator()();
7    };
8
9    intmain()
10   {
11       // f(some_vec) 在一个单独的线程中执行
12       std::threadt1{std::bind(f,some_vec)};
13
14       // F(some_vec)() 在一个单独的线程中执行
15       std::threadt2{F(some_vec)};
16
17       t1.join();
18       t2.join();
19   }
```

上例中的标准库函数 bind 会将一个函数对象作为它的参数。

通常我们需要在执行完一个任务后得到返回的结果。对于那些简单的对返回值没有概念的，我建议使用 std::future。另一种方法是，我们可以给任务传递一个参数，从而这个任务可以把结果存在这个参数中。例如：

```
1    voidf(vector<double>&,double* res);   // 将结果存在 res 中
2
3    structF {
4        vector<double>& v;
5        double* res;
6        F(vector<double>& vv,double* p) :v{vv}, res{p} { }
7        voidoperator()();   //将结果存在 res 中
8    };
9
10   intmain()
11   {
12       doubleres1;
13       doubleres2;
14
15       // f(some_vec,&res1) 在一个单独的线程中执行
16       std::threadt1{std::bind(f,some_vec,&res1)};
17       // F(some_vec,&res2)() 在一个单独的线程中执行
18       std::threadt2{F(some_vec,&res2)};
19
20       t1.join();
21       t2.join();
22
23       std::cout << res1 <<" "<< res2 <<  '\n' ;
24   }
```

但是关于错误呢？如果一个任务抛出了异常应该怎么办？如果一个任务抛出一个异常并且它没有捕获到这个异常，这个任务将会调用 std::terminate()。调用这个函数一般意味着程序的结束。我们常常会为避免这个问题做诸多尝试。std::future 可以将异常传送给父线程（这正是我喜欢 future 的原因之一）。否则，返回错误代码。

除非一个线程的任务已经完成了，当一个线程超出所在的域的时候，程序会结束。很明显，我们应该避免这一点。

没有办法来请求（也就是说尽量文雅地请求它尽可能早的退出）一个线程结束或者是强制（也就是说杀死这个

线程）它结束。下面是可供我们选择的操作：

设计我们自己的协作的中断机制（通过使用共享数据来实现。父线程设置这个数据，子线程检查这个数据（子线程将会在该数据被设置后很快退出））。

使用 thread::native_handle()来访问线程在操作系统中的符号

杀死进程（std::quick_exit()）

杀死程序(std::terminate())

这些是委员会能够统一的所有的规则。特别地，来自 POSIX 的代表强烈地反对任何形式的“线程取消”。然而许多 C++的资源模型都依赖于析构器。对于每种系统和每种可能的应有并没有完美的解决方案。

线程中的一个基本问题是数据竞争。也就是当在统一地址空间的两个线程独立访问一个对象时将会导致没有定义的结果。如果一个（或者两个）对对象执行写操作，而另一个（或者两个）对该对象执行读操作，两个线程将在谁先完成操作方面进行竞争。这样得到的结果不仅仅是没定义的，而且常常无法预测最后的结果。为解决这个问题，C++0x 提供了一些规则和保证从而能够让程序员避免数据竞争。

C++标准库函数不能直接或间接地访问正在被其它线程访问的对象。一种例外是该函数通过参数（包括 this）来直接或间接访问这个对象。

C++标准库函数不能直接或间接修改正在被其它线程访问的对象。一种例外是该函数通过非 const 参数（包括 this）来直接或间接访问这个对象。

C++标准函数库的实现需要避免在同时修改统一序列中的不同成员时的数据竞争。

除非已使用别的方式做了声明，多个线程同时访问一个流对象、流缓冲区对象，或者 C 库中的流可能会导致数据竞争。因此除非你能够控制，绝不要让两个线程来共享一个输出流。

你可以

等待一个线程一定的时间

通过互斥来控制对数据的访问

通过锁来控制对数据的访问

使用条件变量来等待另一个线程的行为

通过 future 来从线程中返回值

同时可参考：

Standard: 30 Thread support library [thread]

17.6.4.7 Data race avoidance [res.on.data.races]

???

H. Hinnant, L. Crowl, B. Dawes, A. Williams, J. Garland, et al.:

Multi-threading Library for Standard C++ (Revision 1)

N2497==08-0007

H.-J. Boehm, L. Crowl:

C++ object lifetime interactions with the threads API

N2880==09-0070.

L. Crowl, P. Plauger, N. Stoughton:

Thread Unsafe Standard Functions

N2864==09-0054.

WG14:

Thread Cancellation N2455=070325.

（翻译：Yibo Zhu）

时间工具程序

在编写程序时，我们常常需要定时执行一些任务。例如，标准库 mutexes 和 locks 提供了的一些选项就需要这一定时功能：线程等待一段时间(duration)或者等到某一给定时刻(time_point)。

如果你需要得到当前时刻，你可以调用 system_clock、monotonic_clock、high_resolution_clock 中任何一个时钟的 now()方法。例如：

```
1   monotonic_clock::time_point t = monotonic_clock::now();
2       // 执行一些代码
3       monotonic_clock::duration d = monotonic_clock::now() - t;
4       // 一些需要 d 个单位时间的任务
```

在上面例子中，一个时钟返回一个 time_point 和一个 duration。其中 duration 是该时钟它返回的两个 time_point 的差值。如果你对细节不感兴趣，你可以使用 auto 类型。

```
1   auto t = monotonic_clock::now();
2       // 执行一些代码
3       auto d = monotonic_clock::now() - t;
4       // 一些需要 d 个单位时间的任务
```

这里提供的时间工具是为了高效支持系统内部的应用。它们不会提供便捷的工具来帮助你维护你的社交日历。事实上，这些时间工具源自于高能物理对时间度量的高精度要求。为了能够表达所有的时间尺度（比如世纪和皮秒），同时避免单位、打字排版以及舍入时的混淆，使用编译时的有理数包来表示 duration 和 time_point。一个 duration 由两部分组成：一个数字时钟”tick”（滴答）和能够表示一个 tick 期望(一秒还是一毫秒？)的事物（一个 period）。这里的 period 是 duration 类型的一部分。下面的表格摘自标准头文件中，它定义了国际单位系统中的 period。这或许会帮助你明白它们的使用范围。

```
1    // 为方便起见，对国际单位做的 typedef:
2    typedefratio<1, 1000000000000000000000000> yocto; // 有条件的支持
3    typedefratio<1,    1000000000000000000000> zepto; // 有条件的支持
4    typedefratio<1,       1000000000000000000> atto;
5    typedefratio<1,          1000000000000000> femto;
6    typedefratio<1,             1000000000000> pico;
7    typedefratio<1,                1000000000> nano;
```

```
8    typedefratio<1,                  1000000> micro;
9    typedefratio<1,                     1000> milli;
10   typedefratio<1,                      100> centi;
11   typedefratio<1,                       10> deci;
12   typedefratio<                       10, 1> deca;
13   typedefratio<                      100, 1> hecto;
14   typedefratio<                     1000, 1> kilo;
15   typedefratio<                  1000000, 1> mega;
16   typedefratio<               1000000000, 1> giga;
17   typedefratio<            1000000000000, 1> tera;
18   typedefratio<         1000000000000000, 1> peta;
19   typedefratio<      1000000000000000000, 1> exa;
20   typedefratio<   10000000000000000000, 1> zetta; //有条件的支持
21   typedefratio<100000000000000000000000, 1> yotta; //有条件的支持
```

编译时有理数提供的常用算术操作符(+, -, *, and /)和比较操作符（==, !=, < , <=, >, >=)适用于合理的duration 和 time_point 组合（比如你不能对两个 time_point 进行加法运算）。系统会对这些运算进行溢出以及除数为0的检查。由于这是编译时的工具，所以不用担心它在的运行时的性能。另外，你还可以使用++、–、+=、-=以及/=来操作 duration，同时可以使用 tp+=d 和 tp-=d 来操作 time_point tp 和 duration d。

下面是一些使用定义在中的 duration 类型的例子：

```
1    microseconds mms = 12345;
2    milliseconds ms = 123;
3    seconds s = 10;
4    minutes m = 30;
5    hours h = 34;
6
7    auto x = std::chrono::hours(3);   // 显式使用命名空间
8    auto x = hours(2)+minutes(35)+seconds(9);// 假设合适的”using”
```

你不能用一个分数来初始化 duration。比如，不要用2.5秒，而应该用2500毫秒。这是因为 duration 被解释为若干个 tick，而每个 tick 表示一个 duration 时间段的单位，比如上面例子中所定义的 milli 和 kilo。所以必须用整数来初始化。Duration 的默认单位是秒。也就是说，时间段为1的 tick 被解释为1秒。在程序中，我们也可以指明 duration 的单位。

```
1    duration d0 = 5;       // 秒 (默认值)
2    duration d1 = 99;       // 千秒
3    duration > d2 = 100;   // d1和 d2的类型相同
```

如果我们想利用 duration 来做一些事（比如打印出这个 duration 的值），那么我们必须给出它的单位。如：

分钟或者微妙。例如：

```
1    auto t = monotonic_clock::now();
2    // 执行一些代码
3    nanoseconds d = monotonic_clock::now() - t;   // 我们希望结果的单位是纳秒
4    cout << “something took ” << d << “nanosecondsn”;
```

或者，我们可以将 duration 转换成一个浮点数

```
1    auto t = monotonic_clock::now();
2    //执行一些代码
3    auto d = monotonic_clock::now() - t;
4    cout << “something took ” << duration(d).count() << “secondsn”;
```

这里的 count()返回 tick 的数量。

同时可参考：

Standard: 20.9 Time utilities [time]

Howard E. Hinnant,

Walter E. Brown,

Jeff Garland,

and Marc Paterno:

A Foundation to Sleep On.

N2661=08-0171.

Including “A Brief History of Time” (With apologies to Stephen Hawking).

(翻译：Yibo Zhu)

标准库中的元组（std::tuple）

在标准库中，tuple（一个 N 元组：N-tuple）被定义为 N 个值的有序序列。在这里，N 可以是从0到文件

中所定义的最大值中的任何一个常数。你可以认为 tuple 是一个未命名的结构体，该结构体包含了特定的 tuple 元素类型的数据成员。特别需要指出的是，tuple 中元素是被紧密地存储的（位于连续的内存区域），而不是链式结构。

可以显式地声明 tuple 的元素类型，也可以通过 make_tuple()来推断出元素类型。另外，可以使用 get()来通过索引（和 C++的数组索引一样，从0而不是从1开始）来访问 tuple 中的元素。

```
1   tuple<string,int> t2(“Kylling”,123);
2
3   // t 的类型被推断为 tuple
4   auto t = make_tuple(string(“Herring”),10, 1.23);
5   // 获取元组中的分量
6   string s = get<0>(t);
7   intx = get<1>(t);
8   doubled = get<2>(t);
```

有时候，我们需要一个编译时异构元素列表（a heterogeneous list of elements at compile time），但又不想定义一个有名字的结构来保存。这种情况下，我们就可以使用 tuple（直接地或间接地）. 例如，我们在 std::function and std::bind 中使用 tuple 来保存参数。

最常用的 tuple 是2-tuple(二元组)，也就是一个 pair。但是标准库已经定义了 pair：std::pair (20.3.3 Pairs)。我们可以使用 pair 来初始化一个 tuple，然而反之则不可。

另外，需要为 tuple 中的元素类型定义比较操作（==, !=, < , <=, >, 和 >=）.

参考：

Standard: 20.5.2 Class template tuple

Variadic template paper

Boost::tuple

unique_ptr

unique_ptr（定义在中）提供了一种严格的语义上的所有权

- 拥有它所指向的对象
- 无法进行复制构造，也无法进行复制赋值操作（译注：也就是对其无法进行复制，我们无法得到指向同一个对象的两个 unique_ptr），但是可以进行移动构造和移动赋值操作
- 保存指向某个对象的指针，当它本身被删除释放的时候（例如，离开某个作用域），会使用给定的删除器（deleter）删除释放它指向的对象。

unique_ptr 的使用能够包括：

- 为动态申请的内存提供异常安全
- 将动态申请内存的所有权传递给某个函数
- 从某个函数返回动态申请内存的所有权
- 在容器中保存指针

“所有 auto_ptr 应该已经具有的（但是我们无法在 C++98中实现的）功能”

unique_ptr 十分依赖于右值引用和移动语义。

下面是一段传统的会产生不安全的异常的代码：

```
1   X* f()
2   {
3       X* p =newX;
4       // 做一些事情 - 可能会抛出某个异常
5       returnp;
6   }
```

解决方法是，用一个 unique_ptr 来管理这个对象的所有权，由其进行这个对象的删除释放工作：

```
1   X* f()
2   {
3       unique_ptr p(newX); // 或者使用{new X}，但是不能 = new X
4       // 做一些事情 - 可能会抛出异常
```

```
5       returnp.release();
6   }
```

现在，如果程序执行过程中抛出了异常，unique_ptr 就会（毫无疑问地）删除释放它所指向的对象，这是最基本的 RAII。但是，除非我们真的需要返回一个内建的指针，我们可以返回一个 unique_ptr，让事情变得更好。

```
1   unique_ptr f()
2   {
3       unique_ptr p(newX);       // 或者使用{new X}，但是不能 = new X
4       //做一些事情 – 可能会抛出异常
5       returnp;   // 对象的所有权被传递出 f()
6   }
```

现在我们可以这样使用函数 f()：

```
1   voidg()
2   {
3       unique_ptr q = f();      // 使用移动构造函数（move constructor）
4       q->memfct(2);              // 使用 q
5       X x = *q;                  // 复制指针 q 所指向的对象
6       // …
7   }   // 在函数退出的时候，q 以及它所指向的对象都被删除释放
```

unique_ptr 拥有“移动意义（move semantics）”，所以我们可以使用函数 f() 返回的右值对 q 进行初始化，这样就简单地将所有权传递给了 q。

在那些要不是为了避免不安全的异常问题（以及为了保证指针所指向的对象都被正确地删除释放），我们不可以使用内建指针的情况下，我们可以在容器中保存 unique_ptr 以代替内建指针：

```
1   vector<unique_ptr<string>> vs {newstring{“Doug”},
2       newstring{“Adams”} };
```

unique_ptr 可以通过一个简单的内建指针构造完成，并且与内建指针相比，两者在使用上的差别很小。特殊情况下，unique_ptr 并不提供任何形式的动态检查(?)。

参考

the C++ draft section 20.7.10

Howard E. Hinnant:

unique_ptr Emulation for C++03 Compilers.

无序容器（Unordered containers）

一个无序容器实际上就是某种形式的蛤希表。C++0x 提供四种标准的无序容器：

unordered_map

unordered_set

unordered_multimap

unordered_multiset

实际上，它们应该被称为 hash_map 等。但是因为有很多地方已经在使用 hash_map 这样的名字了，为了保证其兼容性，标准委员会不得不选择新的名字。而 unordered_map 是我们所能够找到的最好的名字了。无序（“unordered”）代表着 map 和 unordered_map 之间一个最本质的差别：当你使用 map 容器的时候，容器中的所有元素都是通过小于操作（默认情况下使用“<”操作符）排好序的，但是 unordered_map 并没有对元素进行排序，所以它并不要求元素具有小于操作符。并且，一个哈希表也并添言地提供排序的功能。相反，map 容器中的元素也并不要求具有哈希函数。

基本上，当代码的优化是可能的并且我们有理由对其进行优化时，我们可以把 unordered_map 当作一个优化之后的 map 容器来使用。例如：

```
1    map<string,int> m {
2        {“Dijkstra”,1972}, {“Scott”,1976},
3        {“Wilkes”,1967}, {“Hamming”,1968}
4    };
5    m["Ritchie"] = 1983;
6    for(auto x : m)
7        cout << ‘{‘ << x.first << ‘,’ << x.second << ‘}’;
8
9    // 使用优化之后的 unordered_map
10   unordered_map<string,int> um {
11       {“Dijkstra”,1972}, {“Scott”,1976},
```

```
12        { “Wilkes” ,1967}, { “Hamming” ,1968}
13    };
14    um["Ritchie"] = 1983;
15    for(auto x : um)
16        cout <<  ‘{ ‘ << x.first <<  ‘,’  << x.second <<  ‘}’ ;
```

map 容器 m 的迭代器将以字母的顺序访问容器中的所有元素，而 unordered_map 容器 um 则并不按照这样的顺序（除非通过一些特殊的操作）。m 和 um 两者的查找功能的实现机制是非常不同的。对于 m，它使用的是复杂度为 log2(m.size())的小于比较，而 um 只是简单地调用了一个哈希函数和一次或多次的相等比较。如果容器中的元素比较少（比如几打），很难说哪一种容器更快，但是对于大量数据（比如数千个）而言，unordered_map 容器的查找速度要比 map 容器快很多。

更多内容稍后提供。

参考：

• Standard: 23.5 Unordered associative containers.

(翻译：Yibo Zhu)

weak_ptr

弱指针（weak pointer）经常被解释为用来打破使用 shared_ptr 管理的数据结构中循环(?)。但是我认为，将 weak_ptr 看成是指向具有下列特征的对象的指针更好一些。

只有当对象存在的时候，你才需要对其进行访问

并且它可能被其他人删除释放

并且在最后一次使用之后调用其析构函数（通常用于释放那些不具名的内存(anon-memory)资源

（译注：weak_ptr 可以保存一个“弱引用”，指向一个已经用 shared_ptr 进行管理的对象。为了访问这个对象，一个 weak_ptr 可以通过 shared_ptr 的构造函数或者是 weak_ptr 的成员函数 lock()转化为一个

shared_ptr。当最后一个指向这个对象的 shared_ptr 退出其生命周期并且这个对象被释放之后，将无法从指向这个对象的 weak_ptr 获得一个 shared_ptr 指针，shared_ptr 的构造函数会抛出异常，而 weak_ptr::lock 也会返回一个空指针。）

我们来考虑一下如何实现一个老式的“星盘棋”（ asteroid game ）游戏，所有星星（ asteroid ）都属于游戏（ the game ），但是所有星星都必须与它周围的 星星保持联系，并且与之保持相反的状态。要维持一个相反的状态，通常会消去一个或者多个星星，也就是会调用其析构函数。每个星星必须有一个列表来保存记录它周围的星星。这里需要注意的是，在这样一个相邻星星列表中的星星不应该是具有完整生命的(?)，所以 shared_ptr 在这种情况下并不适合。另外一方面，当另外一个星星正看着某个星星时，（ 例如，依赖于这个星星计算其相反状态），这个星星就不能被析构。 当然，星星的析构函数必须被调用以释放其占用的资源（ 比如与图形系统的连接 ）。我们所需要的是一个在任何时间都应该保持完整无缺，并且随时都可以从中获取一个星星的星星列表(?)。weak_ptr 可以帮我们做到这一切：

```
1    voidowner()
2    {
3        // …
4        vector<shared_ptr<Asteroid>> va(100);
5        for(inti=0; i<va.size(); ++i) {// 访问相邻的星星，计算相反状态
6            va[i].reset(newAsteroid(weak_ptr(va[neighbor]));
7            launch(i);
8        }
9        // …
10   }
```

reset() 可以让一个 shared_ptr 指向另外一个新的对象。

当然，我对 ower 类作了相当大的简化，并且只给了每个星星一个邻居。这里的关键是，我们使用了 weak_ptr 指向其邻居星星。在计算相反状态的时候，ower 类则使用 shared_ptr 来代表星星与 owner 之间的所属关系(?)。一个星星的相反关系的计算应该是这样的：

```
1    voidcollision(weak_ptr<Asteroid> p)
2    {
```

```
3        // p.lock 返回一个指向 p 所指对象的 shared_ptr
4        if(auto q = p.lock()) {
5            // … p 以及 q 指向的星星对象依然存在:进行计算…
6        }
7        else{
8            // … oops: 星星对象已经被析构，我们可以忘掉它了(?)…
9        }
10   }
```

注意，即使 owner 决定关闭整个游戏并释放所有的星星对象（通过删除代表所属关系的多个 shared_ptr），每一个正在计算过程中的星星对象仍然可以正确地结束，因为 p.lock()将维持一个 shared_ptr，直到计算过程结束。（译注，也即是说，如果正在计算过程中关闭游戏并通过 shared_ptr 释放对象，那么 p.lock()会维持一个 shared_ptr，这样可以使得 shared_ptr 不会变成0，在计算过程中，星星对象也就不会被错误地释放。当整个计算过程结束后，shared_ptr 的引用计数变为0，星星对象被正确释放。）

我期望看到 weak_ptr 比简单的 shared_ptr 更少地被用到，并且我希望 unique_ptr 可以比 shared_ptr 更加流行，因为 unique_ptr 的所属关系更简单一些（译注：只能有一个 unique_ptr 指针指向某个对象，不向 shared_ptr,可以同时有多个 shared_ptr 指向同一个对象）并且性能更高，因而可以让局部的代码更容易理解。

参考

the C++ draft: Weak_ptr (20.7.13.3)

system error

Stroustrup 先生尚未完成这个主题，请稍后再来。

既然没有完成，我们也可以额外补充一点，了解一下 C++11的这个新特性

sysytem_error 这实际上是 C++1l 为了更好地描述和处理系统错误（异常，）而提供的一个类，定义在〈system_error〉头文件中。

std::system_error 是多个库函数（典型的，与操作系统打交道的函数，比如，std::thread 的构造函数）所抛

出异常的类型，同时，这个异常拥有一个 std::error_code，我们可以根据这个错误码对异常进行处理。一个典型的例子如下:

```
1    #include <thread>
2    #include <iostream>
3    #include <system_error>
4
5    intmain()
6    {
7        try{
8            std::thread().detach();// attempt to detach a non-thread
9        }catch(conststd::system_error& e) {
10           std::cout <<"Caught system_error with code "<< e.code()
11                     <<" meaning "<< e.what() <<'\n';
12       }
13   }
```

更多关于 C++11异常处理以及 system error 的信息，可以参考

std::system_error

error handling