声明

本手册是 Agner Fog 优化手册系列第一册 "Optimizing software in C++:An optimization guide for Windows,Linux adn Mac." 的中文翻译。可以从www.agner.org/optimize/上获取该手册英文版的最新版本。当前中文版是基于2018.9.5日更新的版本翻译的。版权声明请参考本手册最后一章。

1 简介

本手册适用于那些想要使软件更快的编程人员和软件开发者。本手册假设读者熟练掌握 C++ 编程语言,并了解编译器是如何工作的。至于选择 C++ 作为本手册基础的原因，将在稍后解释.

本手册的内容基于笔者对编译器和微处理器是如何工作的研究。本手册中的建议是针对 x86 家族的微处理器，包括 Intel、AMD 和 VIA 的处理器（包括 64 位版本）。x86 处理器是 Widows，Linux， BSD 和 Mac OS X中最常用的平台，即使这些操作系统也适用于其他微处理器，当然很多设备也使用其他平台和变异语言。

本手册是一个系列五本手册中的第一本:

1. Optimizing software in C++:An optimization guide for Windows,Linux adn Mac.
2. Optimizing subroutines in assembly languague:An optimization guide for x86 platforms.
3. The microarchitecture of Intel,AMD and VIA CPUs:An optimization guide for assembly programmers and compiler makers.
4. Instruction tables:Lists of instruction latencies,throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs.
5. Calling conventions for dirrerent C++ compilers and operating systems.

   这些手册的最新版本可以在www.agner.org/optimize/，版权声明将列在手册的最后一章。

只用高级语言编写软件的读者只需要阅读本书即可。后续的内容是为了那些想要深入了解指令集，汇编语言和编译器，处理器微架构的读者准备的。对 CPU 热点代码，可以通过使用汇编获得更高层次的优化，这将会在后续的内容中进一步讨论。

请注意到有非常多的人使用到我的优化手册。因此我不可能有时间回答每一个人的问题。请不要将你的编程问题发送给我，因为你将得不到任何答案。建议初学者在提高自己的编程经验后，再来尝试手册中所提到的技术。,如果你在相关书籍和手册中找不到答案的话，你可以在互联网上的诸多论坛中找到你问题的答案。

我想要感谢那些给我的优化手册发送修正和建议的人，我很高兴能够收到相关信息。

1.1 优化的代价

如今大学的编程课程在软件开发过程中强调结构化、面向对象、模块化、可重用性、系统化。但是这些要求通常都和优化软件的速度和大小相冲突的。

如今，软件老师更经常建议我们函数或者方法的行数应该尽可能的少。但是在几十年前，建议通常是相反的：如果某些功能只会调用一次，那么就不要把他们封装在分离的子程序中。软件编写风格的建议的变化，是因软件项目变的越来越大、越来越复杂，需要将注意力集中在软件开发中，而且电脑的性能也越来越强大。

软件结构化开发的高优先级和程序性能的低优先级，首先反映在编程语言和接口框架的选择上。这对于最终的用户来说，这通常是一个缺点，他们不得不购买性能更加强大的计算机，来跟上更大的软件包，即使对于简单的任务，响应时间也长的不能接受，这使得他们感到沮丧。

有时候为了使软件更小更快，有必要在软件开发的高级原则上妥协。本手册讨论了如何在这些要求之间取得合理的平衡。讨论了如何识别和隔离程序中的最关键部分，并将优化工作集中在该部分。讨论了在相对原始的编程风格中，如何克服不自动检查数组越界，无效指针等。讨论了在哪些高级编程结构需要更多的执行时间，哪些需要更少的执行时间。

2 选择最优平台

2.1 硬件平台的选择

硬件平台的选择相对于过去来说，变成的更不重要了。RISC（精简指令集）和CISC（复杂指令集）处理器、PC和大型主机（mainframes）以及简单处理器（simple processors）和向量处理器（vector processors）之间的区别，变得越来越模糊。拥有CISC指令集的标准PC处理器也包括了RISC核心、向量处理指令（vector processing instruction）、多核、超过以前大型主机的处理速度。

现如今，对于确定任务的硬件平台的选择通常是由诸如价格、兼容性、第二选择（sencond source）和可用的好的开发工具等因素而不是处理能力决定的。在一个网路中连接几个标准PC可能比投资一个大型主机更便宜、更有效率。具有大规模并行向量处理能力的大型超级计算机在科学计算中人有一席之地，但是对于大多数目的来说，标准PC处理器还是首选，应为它们具有更高的性价比。

从技术角度来看，标准PC处理器的的CISC指令集（也称为x86）不是最佳的。这个指令集还在维护，是为了兼容那些在70年代的软件，而当时RAM和硬盘空间是非常稀缺的资源。然而，CISC指令集实际上要比它的名声要好。紧凑的代码使得缓存的效率在缓存资源依旧非常有限的今天更加高效。CISC指令集实际上在缓存资源非常有限的时候比RISC指令集更好。x86指令集最糟糕的问题是缺少寄存器。这个问题在x86指令集的64位扩展中得到了缓解，其中寄存器的数量翻了一倍。

由于无法控制网络资源的响应时间，对于关键的应用程序，不建议使用依赖网络资源的瘦客户机（Thin clients）。

小型手持设备正变得越来越受欢迎，并被用于越来越多的用途，如电子邮件、浏览网页，这些在以前都需要使用一台PC。类似的，我们正看到有越来越多的设备和机器采用嵌入式处理器。我对使用哪些平台和操作系统更高效，没有任具体的建议。但我们需要认识到这些设备通常情况下，内存和计算能力都是要少于PC的，这一点非常重要。因此在这样的系统上节约资源使用比在PC平台上更加重要。然而，通过良好的软件设计，即使在这样的小型设备上，许多应用程序也可以获得良好的表现，这些将在17章进行讨论。

本手册继续标准的PC平台，采用Intel、AMD或者VIA处理器，使用Windows、Linux、BSD或者MAC操作系统。这里给出的很多建议也适用于其它平台，但是都只在PC平台上通过测试。

图形加速器

平台的选择明显受任务要求的影响。例如，较大的图形应用编程序最好在具有图形协处理器或者图形加速卡的平台上实现。一些系统也有专门的物理处理器来处理游戏或者动画中的物理运动。

在某些情况下，可以将图形加速卡的高处理能力用于除了图形渲染之外的其他用途。然而，这样的应用具有非常高的系统依赖性。因此如果可移植性非常重要的话，就不推荐这么做。本手册将不会讨论图形处理器。

可编程逻辑器件

可编程逻辑器件是一种可以使用硬件描述语言（如VHDL、Verilog）进行编程的芯片。常见的有CPLD和FPGA。编程语言（例如C++）和硬件描述语言的区别是：编程语言定义了一个一系列指令的算法，而硬件描述语言定义了由例如*门*、*触发器*、*多路复用器*、*算术单元*等原件和连接它们的导线组成的硬件电路。硬件描述语言天生就是并行的，因为它定义的是电气连接而不是一系列操作序列。

对于一个复杂的数字操作，可编程逻辑器件通常比微处理器中处理的更快，因为硬件可以特定的目的连接。

在FPGA中实现微处理器作为所谓的软核（soft processor）是可能的。然而这样的处理通常比专用微处理器慢的多，因此它本身并没有什么优势。但是在某些情况下，使用硬件描述语言在同一芯片中定义的软核，执行某些关键应用中的特定指令，是一个非常有效的解决方案。当然将专用微处理器和FPGA集成在同一个芯片中是一个更加强大额解决方案。像这样的混合解决方案已经在一些嵌入式系统中被采用了。

我认为这样类似的解决方案，会有一天在PC处理器中采用。应用程序将可以定义由硬件描述语言编码的应用程序专用指令。这样的处理器除了代码缓存和数据缓存外，还将会有用于硬件描述代码的缓存

2.2 微处理器的选择

由于激烈的竞争,不同竞品微处理器的基准性能都非常接近。多核处理器对于那些需要并行运行多个线程的应用程序来说，是好处的。而小型轻量型的低功耗处理器对于非密集型的应用来说也是相当强大的。

一些系统具有图形处理单元，无论是实在图形卡上，亦或是继承在CPU芯片中。这样的单元可以当作协处理器用于一些繁重的图形计算。在某些情况下，也可以将图形处理单元的计算能力用于其它目的，而不是设计它的目的。一些系统还有一个物理处理单元用于计算电脑游戏中物体的运动。

2.3 操作系统的选择

x86家族中，所有较新的处理都可以在16-bit、32-bit以及64-bit模式下运行。

16-bit模式在较早的操作系统 *DOS*和*Windows 3.x*中使用。如果程序或者数据的大小超过64 kbytes，这些系统将使用内存分割。这是非常低效率的。现代微处理器没有针对16-bit模式进行优化，一些系统也没有向后兼容16-bit的程序。除了小型嵌入式系统，是不建议编写16-bit程序的。

如今（2013年）32-bit和64-bit的操作系统非常常见，它们在性能上也没有很大的区别。65-bit软件并没有很大的市场，但可以确定的是64-bit系统将主宰未来。

对于一些具很多函数调用和大量使用CPU资源的应用程序来说，可以提升5-10%的性能。如果性能瓶颈在其它地方，32-bit系统和64-bit系统并没有区别。当然使用大量内存的应用程序可以得益于64-bit系统大地址空间。

软件开发者可以在两个版本中选择需要消耗大量内存的软件：为了与现有系统的兼容的32-bit版本，以及具有最佳性能的64-bit版本。

对于32-bit软件，*Windows操作系统*和*Linux操作系统*的性能几乎相当，因为这两个系统使用同样的函数调用约定（*function calling conventions*）。*FreeBSD*和*Open BSD*在软件优化上，几乎所有方面都是相同的。这里说关于*Linux*的所有建议，同样适用于*BSD 系统*。

基于*Intel*的*Mac OS X操作系统*实在*BSD*的基础上开发的，但是编译器默认使用*位置无关代码*（*position-independent code*）和*延迟绑定*，这会降低它的效率。可以通过使用*静态连接*和关闭*位置无关代码*（选项：**-fno-pic**），来提升性能。

相对于32-bit系统，64-bit系统具有以下几个优点：

1. 两倍的寄存器数量。这样可以在寄存器中而不是内存中存储中间数据和局部变量。
2. 函数参数使用寄存器传递，而不是使用堆栈，这使得函数调用额效率更高。
3. 整数寄存器扩展到64 bits。这样的唯一好处是，应用程序可以使用64位整数。
4. 大内存的分配和释放的效率更高。
5. 所有的64位 CPU和操作系统都支持*SSE2*指令集。
6. 64-bit指令集支持数据的自相关寻址，这使得*位置无关代码*的效率更高。

相对于32-bit系统，64-bit系统具有以下几个缺点：

1. 指针、引用和堆栈入口使用64 bits而不是32 bits，这导致数据缓存的效率更低。
2. 在64 bit模式下，如果装载地址不能保证小于2^{31}, 访问静态或者全局数组将会需要几个额外的指令来计算地址。这些额外的成本在64位的Windows和Mac程序中可以看到，但是在Linux中很少见。
3. 在大内存模型（代码和数据的大小超过2 Gbytes）中，地址的计算将更加的复杂。虽然这种大内存模型很少能用到。
4. 一些指令的长度，64-bit模式下的长度要比32-bit模式下要长1字节。
5. 一些64位编译器要不如它们的32位版本。

总的来说，如果程序有很多函数调用、大量大内存快的分配、或者可以利用64位整数的优势，那么你可以期待64位程序会比32位程序跑的略微快一点。当程序使用超过2 gigabytes的数据时，就非常有必要使用64位的系统了。

当在64位模式下运行时，操作系统之间的相似性将会消失，因为函数的调用约定时不同的。64位的*Windows*只允许4个函数参数通过寄存器传递，而64位的*Linux*、*BSD*、*Mac*允许通过寄存器传递14个参数（6个整数和8个浮点数）。还有其他的细节使得64位*Linux*的函数调用比64位*Windows*的效率更高（详见第五册**Calling conventions for different compilers and operating systems**）。一个具有很多函数调用的的程序，有可能在64位的*Linux*上，比在64位的*Windows*运行的更快。64位*Windows* 的这个缺点可以通关是关键函数为内联的或者静态的，或者通过使用可以使进行这个程序优化的编译器来减轻。

2.4 编程语言的选择

在开始一个新的软件项目之前，决定哪种编程语言最适合手上的项目是非常重要的。低级语言有利于优化程序执行速度，而高级语言则有利于开发出清晰和结构良好的代码，以及快速和容易的开发用户界面，利用网络资源和数据库的接口等。

最终应用程序的效率取决于编程语言是如何实现的。当代码被编程并翻译成二进制可以行代码时，效率最高。**C++**、**Pascal**以及**Fortran**的绝大多数实现都是通过编译器的。

其它一些编程语言通过解释器实现。代码按原样分发（distribute），运行时逐行解释。例如*JavaScript*、*PHP*、*ASP*以及*UNIX shell script*。解释代码是是非常没有效率的，因为循环的每一次迭代，被一次又一次的解释位一个循环的主体。

有些是通过即时编译（*just-in-time compilation*）实现的。程序代码按照原样存储，一边编译一边执行，例如*Perl*。

一些现代编程语言使用一种中间代码（*byte code*，字节码），源码被编程成中间代码，这是分发的代码。中间代码不能按照原样立即执行，在执行之前，它必须经过第二步的解释或者编译。*Java*的一些实现基于解释器，解释器通过模拟所谓的Jave虚拟机来解释中间代码。最好的Jave虚拟机对于代码的最常用部分使用即时编译。*C#*、*托管*C++以及MicroSoft .Net FrameWork的一些其它一些语言都是基于中间代码的即时编译。

使用中间代码的原因在为了独立于平台且紧凑。使用中间代码的最大缺点是：为了解释或者编译中间代码，用户必须安装庞大的*runtime framework*。而这个*framework*通常需要使用比代码本身多的多的资源。中间代码的另一个是：它增加了额外的抽象层，这使得更加具体的优化更加困难。另一方面，即时编译器可以针对它所运行的CPU进行专门的优化，而在预编译代码中进行特定的优化择更加复杂。

编程语言及其实现的历史揭示了一个曲折的过程，反映了效率、平台独立性和易于开发的等相关冲突的考量。例如，第一台PC有一个*Basic*的解释器，而由于*Basic*解释器实在太慢了，很快就有了*Basic*编译器。如今，最受欢迎的*Basic*版本，是基于中间代码和即时编译的*Visual Basic .NET*。一些早期的*Pascal*实现使用类似今天*Java*的中间代码，但从有了真正的可用的编译器后，该语言获得了显著的欢迎。

从本文的讨论中可以清楚的看到，编程语言的选择需要在效率、可移植性和开发时间等原因进行妥协。当效率很重要的时候，解释类编程语言就不再考虑范围内。而当可移植性和易于开发比速度更重要时，基于中间代码和即使编译的语言可能是一种可行的这种方案。这包括*C#*、*Visual Basic*以及最好的*Java*实现。然而，这些语言的缺点时运行时框非常庞大，而每次运行程序时都必须加载该框架。加载框架和编译程序的时间有可能比执行程序所要的时间还长。而且运行时框架所消耗的资源可能比运行程序本身还多。程序使用这样的框架，对于简单的任务例如按下按钮或者移动鼠标，优势会有难以接受的长响应时间。当速度很关键时就应该避免使用*.Net framework*。

毫无疑问，使用完全编译的代码可以获得最快的执行速度。编译语言包括*C*、*C++*、*D*、*Pascal*、*Fortan*以及其它几种非著名语言。由于一些原因，我更喜欢*C++*。一些非常好的编译器和优化的函数库都支持*C++*。*C++*是一种高级的高级语言（advancd high-level language），具有其

他语言中少见的丰富的高级特性。但是*C++*还将低级的*C*语言作为一个自己，因此可以进行低层次的优化。但多数*C++*编译器都支持生成汇编语言，这对于检查编译器对代码的优化成都非常有用。此外，但多数C++编译器允许类似会汇编的函数指令、内联汇编或者易于链接汇编语言模块，但需要最高级别的优化是必要的。*C++*编译器存在于所有主流平台，在这个意义上，*C++*语言是可移植的。*Pascal*相对于*C++*具有很多优势。但是不是很通用。*Fortran*也相当有效率，但是语法相当的过时。

由于有强大的开发工具可用，*C++*开发非常高效。Microsoft Visual Studio是一种非常流行的开发工具。这个工具可以实现*C++*的两种不同实现，直接编译和基于*.NET framework*公共语言运行时的中间代码。显然，当速度很重要时，直接编译的版本更受青睐。

*C++*的一个重要缺点与安全性相关。它没有对数组越界、整数溢出以及无效指针的检查。这些检查的缺席使得代码执行的比那些拥有这些检查的编程语言更快。由于程序规则无法排除这些错误情况，这使得程序员有责任对这些错误进行显示的检查。后面将会有关于这些检查的指导。

当性能优化具有很高优先级时，*C++*绝对时首选的编程语言。与其他编程语言相比，性能上的提升是相当可观的。当性能对最终的用户很重要时，这种性能上的提升，可以很容易证明在开发时间上可能会有细微的提高。

由于其他一些原因，可能需要基于中间代码的高级框架，但是部分代码仍需要仔细优化。在这种情况下，混合实现可能是一个可行的解决档案。代码中最重要的部分可以由基于编译的*C++*或者汇编语言实现，而剩余的部分包括用户界面等，可以使用高级框架实现。被优化的代码部分可以被编译位动态链接库（*DLL*），供其他代码调用。这不是一个最佳的解决档案，因为高级框架任然消耗大量的资源，而这两种代码之间的转换也会产生额外耗费CPU时间的消耗。但是当对时间要求高的部分可以完全包含在DLL中时，这种解决方案也可以显著的提高性能。

另一个值得考虑的选择时*D语言*。*D语言*具有*Java*和*C++*的许多特性，同时避免了很多*C++*的缺点。而且，*D语言*编译成的二进制代码可以与*C*或者*C++*代码链接在译器，但是*D语言*的IDS和编译器没有*C++*的好。

2.5 编译器的选择

市面上有几种不同的*C++*编译器可供选择。很难预测哪一个编译器对于一段特定的代码可以做到最佳的优化。每一个编译器都会做一些非常聪明和非常愚蠢的事情。下面将列举一些常见的编译器。 能都非常接近。多核处理器对于那些需要并行运

Microsoft Visual Studio

这是一个非常友好的编译器，具有许多特性。完整的版本非常昂贵，但是有限制的非商业版本是免费的。Visual Studio可以为. Net框架构建代码，也可以直接编译代码(编译时不使用公共语言运行时，CLR，生成二进制代码)。支持32位和64位 Windows。集成开发环境(IDE)支持多种编程语言的分析和调试。支持多核处理的OpenMP指令。Visual Studio的优化相当好，但它不是最好的优化器。

Borland/CodeGear/Embarcadero C++ builder

它的IDE具有很多和VS相同的特性，只支持32位 Windows。不支持最新的指令集。优化做的没有Microsoft、Intel 和Gnu的编译器好。

Intel C++ compiler (parallel composer)

*Intel*编译器没有它自己的IDE。它可以作为VS和Eclipse的插件。当使用命令行或者make工具时，它也可以作为一个独立的编译器。支持32位和64位 的*Windows*和*Linux*，也支持基于Intel的*Mac OS* 和 *Itaniumx系统*。Intel编译器支持向量指令、自动矢量化、OpenMP和自动并行化。支持CPU调度，为不同的CPU生成不同版本的代码。在所有的平台上，对于内联汇编都有非常好的支持，使得在*Windows*和*Linux*上使用相同的内联汇编语法成为可能。编译还提供了一些具有最佳优化的数学函数库。

*Intel*编译器最重要的缺点是：它编译的代码在*AMD*和*VIA*的处理器上运行的较慢或者根本不运行。可以通过绕过所谓的CPU调度机制来避免这个问题,该调度机制检查代码是否运行在*Intel CPU* 上。

就从代码可以从它众多的优化特性中受益和可以移植到众多平台上的来说，*Intel*编译器是一个很好额选择。

Gnu

虽然缺少用户友好，但这是可以使用的最佳编译器之一。它是免费并且开源的。它支持大多数*Linux*发行版本、*BSD*、*Mac OS X*，无论是32位的还是64位的。支持OpenMP、自动并行化和自动矢量化。Gnu的函数库至今还没有完全优化过。同时支持*AMD*和*Intel*的向量数学库（vector math libraries)。*Gnu C++*编译器在众多的平台上可用，包括32位和64位的*Linux*、*BSD*、*Windows*以及*Mac*。对于所有的平台来说*Gnu*编译器都是一个非常不错的选择。它是使用命令行运行的独立编译器，但是可以用于很多IDE，包括*Eclipse*、*NetBeans*、*CodeBlocks*和*BloodShed*。

Clang

*Clang*编译器基于*LLVM*（Low Level Virtual Machine）。它和*Gnu*编译器在很多方面都相似，并与*Gnu*编译器高度兼容。这是*Mac*平台上最常用的编译器，也支持*Linux*和*Windows*平台。对于所有的平台*Clang*编译器都是一个不错的选择。它可以和*Eclipse IDE*一起使用。

PGI

该编译器支持32位和64位的*Windows*、*Linux*和*Mac*。之前并行编程、OpenMP和自动矢量化。优化相当不错。但是向量指令的效率很低。

7.Digital Mars

这是一个便宜的编译器，用于32位 *Windows*，包含IDE。优化的不是很好。

Open Watcom

另一个32位的Windows开源编译器。默认情况下不符合标准的调用约定，优化合理。

Codeplay VectorC

一个32位的Windows商业编译器。集成到Microsoft Visual Studio IDE中。显然已经不再更新了。可以做自动矢量化。优化适度。支持三种不同的目标文件格式。

总结

在没有IDE的情况下，所有这些编译器都可以作为命令行版本使用。商业编译器有免费的试用版本提供。

在*Linux*平台上，通常可以混合来自不同编译器的目标文件（*Object File*），在某些情况下，也可以在Windows平台上也可以。*Microsoft*和*Intel*的Windows编译器在目标文件级别上完全兼容，而*Digital Mars*编译器基本上与它们兼容。*Embarcadero*、*Codeplay*和*Watcom*编译器在目标文件级别上与其他编译器不兼容。

为了良好的代码性能，我建议在Unix应用程序中使用Gnu、Clang或Intel编译器，在Windows应用程序中使用Gnu、Clang、Intel或Microsoft编译器。如果您希望您的代码在AMD微处理器上高效运行，请不要使用Intel编译器。

编译器的选择可能在某些情况下由遗留代码兼容的要求，IDE具体的参数选择，调试工具，简单的GUI开，数据库集成web应用程序集成，混合语言编程等决定。如果所选择的编译器不提供最好的优化，在这种情况下，最关键的模块使用不同的编译器可能是非常有帮助的。在大多数情况下，如果包含必要的库文件，那么由*Intel*编译器生成的目标文件可以毫无问题地链接到使用*Microsoft*或*Gnu*编译器生成的项目中。或者，使用最好的编译器生成DLL，并从使用另一个编译器构建的项目中调用它。

2.6 函数库的选择

有些应用程序将大部分执行时间花在执行库函数上。Tim-econsuming库函数通常属于以下类别之一：

1. 文件输入/输出
2. 图形和声音处理
3. 内存和字符串操作
4. 数学函数
5. 加密，解密和数据压缩

大多数编译器都包含用于这些目的的标准库。不幸的是，标准库并不总是完全优化的。

库函数通常是许多用户在许多不同应用程序中使用的一小段代码。因此，与优化特定于应用程序的代码相比，值得在优化库函数方面投入更多的精力。最好的函数库是使用*汇编语言*和*自动cpu调度*以及最新的*指令集扩展*高度优化的。

如果分析显示库函数在某个特定应用程序占用了大量CPU时间，或者如果这是显而易见的，那么可以通过使用不同的函数库来显著提高性能。如果应用程序在库函数中花费了大部分时间，那么除了寻找最有效的库和节省库函数调用之外，可能不需要优化其他任何地方。建议尝试不同的库，看看哪个最好。

下面将讨论一些常见的函数库。还有许多用于特殊目的的库。

Microsoft

微软编译器自带。有些函数优化得很好，有些则没有。支持32位和64位 *Windows*。

Borland / CodeGear / Embarcadero

*Borland C++ builder*自带。未针对*SSE2*和后续指令集进行优化。只支持32位 *Windows*。

Gnu

*Gnu*编译器自带的。没有像编译器本身优化的好。64位版本比32位版本好。Gnu编译器经常插入内置代码，而不是最常见的内存和字符串指令。内置代码不是最优的。使用选项*-fno-builtin*使用库版本替代内置版本。*Gnu*库支持32位和64位*Linux*和*BSD*。Windows版本目前还不是最新的。

Mac

Mac OS X（*Darwin*）* Gnu*编译器中包含的库是*Xnu*项目的一部分。在所谓的**commpage**中，操作系统内核中包含了一些最重要的函数。这些功能针对*Intel Core*和稍后的*Intel*处理器版本进行了高度优化。*AMD*处理器和早期的英特尔处理器根本不被支持。只能在Mac平台上运行。

Intel

*Intel*编译器包含标准函数库。还有一些特殊用途的库，如"*Intel Math Kernel Library*"和"*"ntegrated Performance Primitives*"。这些函数库针对大型数据集进行了高度优化。然而，英特尔的库在AMD和VIA处理器上并不能总是运行良好。有关解释和可能的解决方法，请参见后面的章节。支持所有x86和x86-64平台。

AMD

*AMD*数学核心库包含优化的数学函数。它也适用于英特尔处理器。性能不如*Intel*库。支持32位和64位*Windows*和*Linux*。

AsmLib

我自己的函数库是为了演示而创建的。可以从www.agner.org/optimize/asmlib.zip获得。目前包括内存和字符串函数的优化版本，以及其他一些很难在其他地方找到的函数。在最新的处理器上运行时，比大多数其他库都要快。支持所有x86和x86-64平台。

Test	Processor	Microsoft	CodeGear	Intel	Mac	Gnu 32-bits	Gnu 32-bits -fno-builtin	Gnu 64-bits -fno-builtin	Asmlib
memcpy16kB aligned operands	Intel Core 2	0.12	0.18	0.12	0.11	0.18	0.18	0.18	0.11
memcpy16kB unaligned op.	Intel Core 2	0.63	0.75	0.18	0.11	1.21	0.57	0.44	0.12
memcpy16kB aligned operands	AMD Opteron K8	0.24	0.25	0.24	n.a.	1.00	0.25	0.28	0.22
memcpy16kB unaligned op.	AMD Opteron K8	0.38	0.44	0.40	n.a.	1.00	0.35	0.29	0.28
strlen128 bytes	Intel Core 2	0.77	0.89	0.40	0.30	4.5	0.82	0.59	0.27
strlen128 bytes	AMD Opteron K8	1.09	1.25	1.61	n.a.	2.23	0.95	0.6	1.19

表2.1. 不同函数库性能对比

表中的数字是每字节数据的核心时钟周期(低数字意味着良好的性能)。对齐的操作数意味着源和目标的地址都可以被16整除。 用于测试库的版本（不是最新的） * Microsoft Visual studio 2008, v. 9.0 * CodeGear Borland bcc, v. 5.5 * Mac: Darwin8 g++ v 4.0.1. * Asmlib: v. 2.00 * Intel C++ compiler, v. 10.1.020. 使用库*libircmt.lib* 中的 $\_\int el_f *_m emcpy$ 和 $_\int el_{\neq} w_s tr \leq n$函数。函数名没有文档。

2.7 用户界面框架的选择

典型软件项目中的大多数代码都进入用户界面。不需要大量计算的应用程序很可能在用户界面上花费的CPU时间比在程序的基本任务上花费的还要多。

程序员很少从头开始编写自己的图形用户界面。这不仅浪费了程序员的时间，也给最终用户带来了不便。出于可用性的考虑，菜单、按钮、对话框等应该尽可能地标准化。程序员可以使用操作系统附带的标准用户界面元素或编译器和开发工具附带的库。

*Microsoft Foundation Classes*是一个流行的*Windows C++*用户界面库(**MFC**)。与之竞争的产品是Borland现已停止继续维护的*Object Windows Library*（**OWL**）。*Linux*系统有几个可用的图形界面框架。用户界面库可以作为运行时DLL或静态库链接。除非多个应用程序同时使用同一个DLL，运行时DLL比静态库占用更多的内存资源。

用户界面库可能比应用程序本身更大，需要更多的时间来加载。一个轻量级的替代方案是*Windows Template Library*（**WTL**）。*WTL*应用程序通常比*MFC*应用程序更快、更紧凑。由于糟糕的文档、缺乏高级开发工具，WTL应用程序可能会花费更多的时间去开发。

通过放弃使用图形用户界面并使用控制台模式程序，可以获得最简单的用户界面。控制台模式程序的输入通常在命令行或输入文件中指定。输出到控制台或文件。控制台模式的程序是快速、紧凑和易于开发的。方便移植到不同的平台，因为它不依赖于系统特定的图形界面调用。可用性可能很差，因为它缺少图形用户界面的自解释菜单。控制台模式程序对于从其他应用程序(如实现工具库)调用非常有用。

结论是，用户界面框架的选择必须是开发时间、可用性、程序紧凑性和运行时间之间的折衷。没有一个通用的解决方案对所有应用程序都是最好的。

2.8 克服C++语言的缺点

虽然c++在优化方面有很多优点，但它也有一些缺点，这使得开发人员不得不选择其他编程语言。本节将讨论在选择*C++*进行优化时如何克服这些缺点。

<u>可移植性</u>

=== c++是完全可移植的，因为它的语法在所有主要平台上都是完全标准化和受支持的。然而，c++也是一种允许直接访问硬件接口和系统调用的语言。这些当然是系统特有的。为了方便在平台之间进行移植，建议将用户界面代码和其他系统特定部分放在一个单独的模块中，并将代码的任务特定部分(应该是系统独立的)放在另一个模块中。

整数的大小和其他硬件相关细节取决于硬件平台和操作系统。详情见第29页（TODO）。

<u>开发时间</u>

=== 一些开发人员认为特定的编程语言和开发工具比其他语言和开发工具使用起来更快。虽然有些区别仅仅是习惯的问题，但确实有些开发工具具有强大的功能，可以自动完成许多琐碎的编程工作。通过一致的模块化和可重用类，可以降低*C++*项目的开发时间并提高可维护性。

<u>安全性</u>

=== *C++*语言最严重的问题与安全性有关。标准*C++*的实现没有检查数组边界违规和无效指针。这是*C++*程序中常见的错误来源，也是黑客可能的攻击点。有必要遵守某些编程原则，以防止在涉及安全性的程序中出现此类错误。

无效指针的问题可以通过使用引用代替指针，通过初始化指针为0，通过将指针指向的对象无效时将指针设置为0来避免，还可以通过避免指针算术和指针类型转换来避免。通常使用指针的链表和其他数据结构可以使用更高效的容器类模板替代，如第95页（TODO：）所述。避免使用`scanf`函数。

数组越界可能是*C++*程序错误的最常见原因。对数组边界外的赋值操作，可能会重写其他变量，更糟糕的是，它可能会重写定义数组的函数的返回地址。这会导致各种奇怪和意想不到的行为。数组通常用作存储文本或输入数据的缓冲区。缺少对输入数据缓冲区溢出的检查是黑客经常利用的一个常见错误。

防止此类错误的一个好方法是使用经过良好测试的容器类来替换数组。标准模板库（**STL**）是此类容器类的一个有用来源。不幸的是，许多标准容器类以一种低效的方式来使用动态内存分配。有关如何避免动态内存分配的示例，请参见第92页（TODO：）。有关高效容器类的讨论，请参见第95页（TODO：）。www.agner.org/optimize/cppexamples.zip上本手册的附录含带有边界检查和各种高效容器类的数组示例。

文本字符串尤其有问题，因为字符串的长度可能没有特定的限制。在字符数组中存储字符串的旧式*C*风格方法快速有效，但不安全，除非在存储之前检查每个字符串的长度。这个问题的标准解决方案是使用*string*类，例如**string**或**CString**。这是安全且灵活的，但在大型应用程序中效率非常低。每次创建或修改字符串时，string类都会分配一个新的内存块。这可能会导致内存碎片化，并涉及堆管理和垃圾收集的高开销成本。一个不影响安全性的更有效的解决方案是将所有字符串存储在一个内存池中。有关如何在内存池中存储字符串，请参见附录中的示例（参见www.agner.org/optimize/cppexamples.zip）。

整数溢出是另一个安全问题。官方的*C*标准说，在溢出的情况下，有符号整数的行为是"未定义的"。这允许编译器忽略溢出或假设它没有发生。在*Gnu*编译器的情况下，不发生带符号整数溢出的假设的不幸后果是，它允许编译器优化掉溢出检查。对于这个问题，有许多可能的补救措施：（1）在溢出前进行，（2）使用无符号整数——它们是保证回绕（*wrap around*），（3）使用选项`-ftrapv`捕获整数溢出,但这是非常低效的，（4）使用选项`-Wstrict-overflow = 2`，对这样的优化进行警告，(5)使用选项`-fwrapv`或`-fno-strict-overflow`明确定义溢出行为。

在代码中速度很重要的关键部分，您可能会偏离上述安全建议。如果不安全的代码仅限于经过良好测试的函数、类、模板或模块，并且与程序的其余部分有良好定义的接口，那么这是被允许的。

3 找到消耗时间最多的地方

3.1 一个时钟周期是多少?

在本手册中，我使用CPU时钟周期而不是秒或微秒作为时间度量单位。这是因为不同计算机有不同的速度。今天，如果我写下某事需要10μs,那么在下一代的电脑，它可能只需要5μs，而我的手册将很快被淘汰。但是如果我写下某事需要10个时钟周期，即使CPU时钟频率加倍，那么它仍然需要10个时钟周期。

时钟周期的长度是时钟频率的倒数。例如，如果时钟频率是2 GHz，那么时钟周期的长度是：

$$\frac{1}{2GHz} = 5ns$$

一台计算机上的时钟周期并不总是可以与另一台计算机上的时钟周期相比较。奔腾4 （*NetBurst*） CPU的被设计为具有比其他CPU更高的时钟频率，但是总的来说，在执行同一段代码时，它比其他CPU耗费更多的时钟周期。

假设程序中的一个循环重复1000次，循环中有100个浮点运算（加法、乘法等）。如果每个浮点运算需要5个时钟周期,然后我们可以大致估计，循环将1000 * 100 * 5 * 0.5 ns = 250μs 2 GHz CPU。我们应该尝试优化这个循环吗?当然不！250μs小于1/50的时间刷新屏幕。用户不可能看到延迟。但是在这个循环中还存在另一个循环，另一个循环也重复1000次，那么我们估计计算时间为250毫秒。这种延迟的时间足够长，足以引起注意，但也不够长，足以令人厌烦。我们可能决定做一些测量，看看我们的估计是否正确，或者计算时间是否实际超过250毫秒。如果响应时间太长，用户实际上必须等待结果，那么我们将考虑是否有可以改进的地方。

3.2 使用分析器查找热点（hot spots）

在开始优化任何东西之前，必须先识别程序的关键部分。在一些程序中，超过99%的时间花在最内部的循环中进行数学计算。在其他程序中，99%的时间花在读取和写入数据文件上，只有不到1%的时间花在实际操作这些数据上。优化重要部分的代码，而不是优化只占总时间的一小部分代码，这一点非常重要。优化代码中不太重要的部分不仅会浪费时间，还会使代码不太清晰，更难于调试和维护。

大多数编译器包都包含一个分析器，它可以告诉每个函数被调用的次数和时间。也有第三方剖析器，如*AQtime*、*Intel VTune*和*AMD CodeAnalyst*。

有几种不同的分析方法:

- 植入：编译器在每次函数调用时插入额外的代码，以计算调用函数的次数和时间。
- 调试：分析器在每个函数或每一行代码中插入临时调试断点。
- 基于时间的采样：分析器告诉操作系统生成一个中断，例如每毫秒一次。分析器计算在程序的每个部分中发生中断的次数。这不需要修改被测程序，但可靠性较差。
- 基于事件的采样：分析器告诉CPU在某些事件上生成中断，例如每次发生1000次缓存丢失。这使得查看程序的哪个部分有最多的缓存丢失、分支错误预测、浮点异常等等成为可能。基于事件的采样需要基于CPU的分析其。对于Intel CPU使用*Intel VTune*，对于AMD CPU使用*AMD CodeAnalyst*。

不幸的是，分析器通常是不可靠的。它们有时会给出误导的结果，或者完全因为技术问题而失败。分析器的一些常见问题是：

- 粗糙的时间测量。如果时间是以毫秒级的分辨率测量的，并且关键函数的执行需要几微秒，那么测量可能变得不精确，或者干脆为零。
- 执行时间过短或过长。如果被测程序在短时间内完成，那么采样生成的数据太少，无法进行分析。如果程序执行时间太长，那么采样生成的数据太多，超出分析器的分析能力。
- 等待用户输入。许多程序将大部分时间用于等待用户输入或网络资源。这个时间包含在分析文件中。为了使分析可行，可能需要修改程序以使用一组测试数据而不是用户输入。
- 来自其他过程的干扰。分析器不仅测量被测试程序中所花费的时间，而且还测量在同一台计算机上运行的所有其他进程（包括分析器本身）所使用的时间。
- 函数地址在优化的程序中是模糊的。分析器通过地址识别程序中的任何热点，并尝试将这些地址转换为函数名。但是，高度优化的程序经常以这样一种方式重新组织:函数名和代码地址之间没有明确的对应关系。内联函数的名称对于分析器可能根本不可见。其结果将是关于哪些功能花费的时间最多的误导性报告。
- 使用调试版本的代码。一些分析器要求您正在测试的代码包含调试信息，以便识别单个函数或代码函数。代码的调试版本没有优化。
- 在CPU内核之间跳转。进程或线程不一定停留在多核CPU上的同一处理器内核中，但事件计数器可以。这导致在多个CPU内核之间跳转的线程的事件计数没有意义。您可能需要通过设置线程关联掩码将线程锁定到特定的CPU内核。
- 再现性不好。程序执行中的延迟可能是由不可重现的随机事件引起的。诸如任务切换和垃圾收集之类的事件可以在随机时间发生，这使得程序的某些部分看起来比正常情况下花费的时间更长。

有多种方法可以替代分析器。一个简单的替代方法是在调试器中运行程序，并在程序运行时按下**break**。如果有一个热点，占用90%的CPU时间，那么中断有90%的机会发生在这个热点。重复中断几次可能足以确定一个热点。在调试器中使用调用堆栈来识别热点周围的情况。

有时，识别性能瓶颈的最佳方法是将度量工具直接放入代码中，而不是使用现成的分析器。这并不能解决与概要分析相关的所有问题，但通常会提供更可靠的结果。如果您不满意分析器的工作方式，那么您可以将所需的测量仪器插入程序本身。您可以添加计数器变量来计算程序的每个部分执行了多少次。此外，您可以读取程序中每个最重要或关键部分前后的时间，以度量每个部分所花费的时间。有关此方法的进一步讨论，请参阅第157页（TODO）。

您的测量代码应该包含`#if`指令，以便在代码的最终版本中禁用它。在代码中插入自己的分析工具是z在程序开发过程中跟踪程序性能的一种非常有用的方法。

如果时间间隔很短，时间测量可能需要很高的分辨率。在Windows中，您可以使用GetTickCount或QueryPerformanceCounter函数获得毫秒级的分辨率。使用CPU中的时间戳计数器可以获得更高的分辨率，它以CPU时钟频率计数（在Windows中: __rdtsc()）。

如果线程在不同的CPU内核之间跳转，时间戳计数器将失效。在时间度量期间，您可能必须将线程固定到特定的CPU核心，以避免这种情况。（在*Windows*中是SetThreadAffinityMask，在*Linux*中是sched_setaffness)。

程序应该用一组真实的测试数据进行测试。测试数据应该具有一个典型的随机性，以便获得真实数量的缓存丢失和分支错误预测。

当发现程序中最耗时的部分时，重要的是将优化工作集中在耗时的部分上。关键代码片段可以使用第157页中（TODO）描述的方法进行进一步测试和研究。

分析器对于查找与CPU密集型代码相关的问题最有用。但是许多程序在加载文件或访问数据库、网络和其他资源时所花费的时间比算术运算要多。下面几节将讨论最常见的时间消耗器。

3.3 安装程序

安装程序包所需的时间通常不被认为是软件优化问题。但这肯定会占用用户的时间。如果软件优化的目标是为用户节省时间，那么安装软件包并使其正常工作所花费的时间是不能忽略的。由于现代软件的高度复杂性，安装过程花费一个多小时是很正常的。为了找到并解决兼容性问题，用户必须多次重新安装软件包，这种情况也很常见。

软件开发人员在决定软件包是否使用需要安装许多文件的复杂框架时，应该考虑安装时间和兼容性问题。安装过程应该始终使用标准化的安装工具。应该可以在开始时选择所有安装选项，以便在无人参与的情况下继续安装过程的其余部分。卸载也应该以标准化的方式进行

3.4 自动更新

许多软件程序通过互联网定期自动下载更新。有些程序在每次计算机启动时都会搜索更新，即使该程序从未被使用过。安装了许多这类程序的计算机需要几分钟才能启动，这完全是在浪费用户的时间。其他一些程序在每次启动时使用时间搜索更新。如果当前版本满足用户的需求，则用户可能不需要更新。搜索更新应该是可选的，默认情况下是关闭的，除非有令人信服的安全理由进行更新。更新过程应该在低优先级线程中运行，并且只有在程序实际使用时才运行。任何程序在不使用时都不应该在后台进程运行。下载的程序更新的安装应该延迟到程序关闭并重新启动。

操作系统的更新尤其耗时。有时安装操作系统的自动更新需要几个小时。这是非常有问题的，因为这些耗时的更新可能在不方便的时候出现。如果用户在离开工作场所之前出于安全原因必须关闭或注销计算机，并且系统禁止用户在更新过程中关闭计算机，那么这将是一个非常大的问题。

3.5 程序加载

很多时候，加载一个程序要比执行它花费更多的时间。对于基于大型运行时框架、中间代码、解释器、即时编译器等的程序，加载时间可能会非常长，这是使用*Java*, *C#*， *Visual Basic*等编程语言编写的程序的常见情况。

但是，即使是用编译的*C++*实现的程序，加载程序也会耗费时间。如果程序使用大量运行时DLL（动态链接的库或共享对象）、资源文件、配置文件、帮助文件和数据库，通常会发生这种情况。当程序启动时，操作系统可能不会加载一个大程序的所有模块。有些模块可能只在需要时加载，或者在RAM大小不足时将其交换到硬盘。

用户希望对简单的操作（如按键或鼠标移动）立即作出响应。如果因为它需要从磁盘加载模块或资源文件，这样的响应延迟了几秒钟，这对于用户来说是不可接受的。使用大量内存的应用程序会迫使操作系统将内存交换到磁盘。内存交换是鼠标移动或按键等简单操作的响应时间长得不可接受的常见原因。

避免大量的DLL，配置文件、资源文件、帮助文件等，分散再硬盘的不同地方。几个文件、最好和*.exe*文件在同一个路径下，这样是可以接受的。

3.6 动态链接和位置无关的代码

函数库可以是静态链接库（\.lib，*.a），*或动态链接库，也称为共享对象（*.dll，* . so*）。有几个因素可以使动态链接库比静态链接库慢。这些因素将在下面的149页（TODO）详细解释。

位置无关代码用于类*unix*系统中的共享对象。默认情况下，*Mac*系统在任何地方都使用与位置无关的代码。位置无关的代码效率很低，尤其是在32位模式下，原因如下面的149页所述（TODO）。

3.7 文件存取

读取或写入硬盘上的文件通常比处理文件中的数据花费更多的时间，特别是如果用户有一个病毒扫描程序，扫描所有访问的文件。

文件的顺序向前访问比随机访问快。读或写大块比一次读或写一小块文件更快。一次读写不要少于几千字节。你可以将整个文件复制到内存缓冲区中，并在一个操作中读写它，而不是以非顺序的方式读写几个位。通常，访问最近访问过的文件要比第一次访问快得多。这是因为文件已经复制到磁盘缓存。

远程或可移动媒体(如软盘和u盘)上的文件可能不会被缓存。这可能会产生非常戏剧性的后果。我曾经编写过一个Windows程序，该程序通过调用WritePrivateProfileString来创建一个文件，它每写一行就会打开和关闭一次文件。由于磁盘缓存，这在硬盘上工作得非常快，但是将文件写入软盘需要几分钟。

包含数字数据的大文件如果以二进制形式存储比以ASCII格式存储的文件更紧凑和高效。二进制数据存储的一个缺点是它不可读，并且不容易移植到具有大端存储的系统中。

在具有许多文件输入/输出操作的程序中，优化文件访问比优化CPU使用更重要。如果处理器在等待磁盘操作完成时可以执行其他工作，那么将文件访问放在一个单独的线程中可能会有好处。

3.8 系统数据库

在Windows中访问系统数据库可能需要几秒钟时间。与Windows系统中的大型注册数据库相比，将特定于应用程序的信息存储在单独的文件中更有效。注意，如果使用GetPrivateProfileString和WritePrivateProfileString等函数读写配置文件（*.ini文件），系统可能会将信息存储在数据库中。

3.9 其他数据库

许多软件应用程序使用数据库来存储用户数据。数据库会消耗大量的CPU时间、RAM和磁盘空间。在简单的情况下，可以用普通的旧数据文件替换数据库。数据库查询通常可以通过使用索引、使用集合而不是循环等方式进行优化。优化数据库查询超出了本手册的范围，但是你应该知道，优化数据库访问通常可以获得很多好处。

3.10 图形

图形用户界面可能使用大量的计算资源。通常会使用特定的图形框架。操作系统可以在其API中提供这样的框架。在某些情况下，在操作系统API和应用程序软件之间有一个额外的第三方图形框架层。这样一个额外的框架会消耗大量额外的资源。、

应用软件中的每个图形操作都通过调用图形库或API函数的函数调用实现的，然后这些函数调用设备驱动程序。对图形函数的调用非常耗时，因为它可能经过多个层，并且需要切换到受保护模式并再次返回。显然，对绘制整个多边形或位图的图形函数进行一次调用要比通过多次函数调用分别绘制每个像素或线条更有效率。

计算机游戏和动画中图形对象的计算当然也是很耗时的，特别是在没有图形处理单元的情况下。

各种图形函数库和驱动程序的性能差别很大。对于使用哪一种更好，我没有具体的建议。

3.11 其它系统资源

对打印机或其他设备的最好是一次写入大块内容，而不是每次一小块，因为对驱动程序的每次调用都涉及到切换到受保护模式并再次返回的开销。

访问系统设备和使用操作系统的高级工具可能会很耗时，因为它可能涉及到加载几个驱动程序、配置文件和系统模块。

3.12 访问网络

一些应用程序使用Internet或内部网络进行自动更新、远程帮助文件、数据库访问等。这里所存在的问题是访问时间无法控制。在简单的测试配置中，网络访问可能会很快，但在网络过载或用户远离服务器的使用情况下，网络访问可能很慢或完全无法访问。在决定是在本地还是远程存储帮助文件和其他资源时，这些问题应该被列入考量之中。如果需要频繁更新，那么最好在本地映射远程数据。访问远程数据库通常需要使用密码登录。对于许多辛勤工作的软件用户来说，登录过程是一个恼人的时间消耗器。在某些情况下，如果网络或数据库负载过重，登录过程可能需要一分钟以上。

3.13 访问内存

与对数据进行计算所需的时间相比，从RAM内存访问数据需要相当长的时间。这就是所有现代计算机都有内存缓存的原因。通常，一级数据缓存为8 - 64 Kb，二级缓存为256 Kb到2 Mb。计算机中也有可能还存在一个三级缓存。

如果程序中所有数据的总和大于二级缓存，并且分散在内存中或以非顺序方式访问，那么内存访问可能是程序中最耗时的地方。如果变量在内存缓存中，读写它只需要2-3个时钟周期，如果不缓存，则需要几百个时钟周期。参见第26页（TODO）关于数据存储，第89页（TODO）关于内存缓存。

3.14 上下文切换

上下文切换是多任务环境中不同任务之间的切换，多线程程序中不同线程的切换，或大型程序中的不同部分的切换。频繁的上下文切换会降低性能，因为数据缓存的内容，代码缓存、分支目标缓冲区、分支模式历史等可能需要更新。

如果分配给每个任务或线程的时间片很小，上下文切换将会更频繁。时间片的长度由操作系统决定，而不是由应用程序。

在具有多个CPU或具有多核CPU的计算机中，上下文切换的数量更少。

3.15 依赖链

现代微处理器可以乱序执行。这意味着如果软件指定A和B的计算，而A的计算速度较慢，则微处理器可以在计算完A之前开始计算B。显然，这只有在计算B不需要A的值时才有可能。

为了利用乱序执行的优势，必须避免长依赖链。依赖链是一系列的计算，其中每个计算依赖于前一个计算的结果。这可以妨碍CPU同时执行多个计算或者导致混乱。有关如何打破依赖关系链的示例，请参见第105页（TODO）。

3.16 执行单元的吞吐量

延迟和执行单元的吞吐量之间有一个重要的区别。例如，在现代CPU上执行浮点加法可能需要3-5个时钟周期。但每个时钟周期可以开始一个新的浮点加法。这意味着，如果每个加法依赖于前一个加法的结果，那么每三个时钟周期只有一个加法。但是，如果所有的加法都是独立的，那么每个时钟周期可以有一个加法。

在计算密集型的程序中，如果没有上面提到的耗时代码占主导地位，并且没有很长的依赖链，则可能获得最高的性能。在这种情况下，性能受到执行单元吞吐量的限制，而不是延迟或内存访问的限制。

现代微处理器的执行核心分为几个执行单元。通常，有两个或多个整数单元、一个或两个浮点加法单元和一个或两个浮点乘法单元。这意味着可以同时进行整数加法、浮点加法和浮点乘法。

因此，进行浮点运算的代码最好能够平衡加法和乘法。减法和加法使用相同的执行单位。除法需要更长的时间。在浮点操作之间执行整数操作而不降低性能是可能的，因为整数操作使用不同的执行单元。例如，执行浮点运算的循环通常使用整数运算来递增循环计数器、比较循环计数器与其极限等。在大多数情况下，可以假设这些整数操作不会增加总计算时间。

4 性能和可用性

更好的软件产品是可以为用户节省时间的产品。对于许多计算机用户来说，时间是一种宝贵的资源，许多时间浪费在速度慢、难于使用、不兼容或容易出错的软件上。所有这些问题都是可用性问题，我认为应该从更广泛的可用性角度来看待软件性能。

这不是一本关于可用性的手册，但是我认为有必要在这里引起软件编程人员注意一下影响软件使用效率的常见障碍。有关这个主题的更多信息，请参见我在Wikibooks上提供的免费电子书Usability for Nerds。

下面的列表指出了一些让软件用户感到沮丧和浪费时间的典型原因，以及软件开发人员应该注意的一些重要的可用性问题。

1. 大型的运行时框架。*.NET*框架和*Java*虚拟机框架通常比它们运行的程序占用更多的资源。这些框架是资源问题和兼容性问题的常见来源，它们在框架本身的安装过程中、在框架下运行的程序的安装过程中、在程序启动过程中以及在程序运行过程中都会浪费大量时间。使用这种运行时框架的主要原因是为了跨平台的可移植性。不幸的是，跨平台兼容性并不总是如预期的那么好。我相信通过更好地标准化编程语言、操作系统和API，可以更有效地实现可移植性。

2. 内存交换。软件开发人员通常拥有比最终用户拥有更多RAM的功能更强大的计算机。因此，开发人员可能看不到过多的内存交换和其他资源问题，对于最终用户，这些问题导致使用大量资源的应用程序表现很差。

3. 安装问题。程序的安装和卸载过程应该标准化，由操作系统而不是由单独的安装工具来完成。

4. 自动更新。如果网络不稳定或新版本有旧版本中不存在的问题，则软件的自动更新可能会导致问题。更新机制经常会弹出一些烦人的弹出消息，比如请安装这个重要的新更新，或者告诉用户在忙于重要工作时重启计算机。更新机制不应该中断用户，而应该只显示一个独立的图标，表示更新的可用性，或者在计算机重新启动时自动更新。软件分销商经常滥用更新机制来宣传其软件的新版本。这对用户来说很烦人。

5. 兼容性问题。所有软件都应该在不同的平台、不同的屏幕分辨率、不同的系统颜色设置和不同的用户访问权限上进行测试。软件应该使用标准的API调用，而不是自定义的hack（？）和直接的访问硬件。应该使用现成的协议和标准化的文件格式。Web系统应该在不同的浏览器、不同的平台、不同的屏幕分辨率等环境中进行测试。应遵守可访问性指南。

6. 复制保护。一些复制保护方案是基于违反或规避操作系统标准的黑客攻击。这种方案是兼容性问题和系统崩溃的常见根源。许多复制保护方案都是基于硬件识别的。当硬件更新时，这种方案会导致问题。大多数的复制保护方案都让用户感到厌烦，并且阻止合法备份复制而没有组织非法的复制。应该权衡复制保护方案的好处和在可用性和必要支持上付出的成本。

7. 硬件更新。更改硬盘或其他硬件通常要求重新安装所有软件，并且还会丢失用户设置。花上整个工作日或者更多时间重新安装的情况也很常见。许多软件应用程序需要更好的备份功能，当前的操作系统需要更好的硬盘复制支持。

8. 安全。具有网络访问权限的软件易受病毒攻击和其他滥用的弱点，可能使用户付出极其昂贵的代价。而防火墙、病毒扫描器和其他保护手段是造成兼容性问题和系统崩溃的最常见原因。此外，病毒扫描器比计算机上的其他任何东西都要花费更多的时间，这种情况并不少见。作为操作系统一部分的安全软件通常比第三方安全软件更可靠。

9. 后台服务。许多在后台运行的服务对用户来说是不必要的，是对资源的浪费。考虑只在用户激活时运行服务。

10. 过多的特性。由于市场原因，软件通常会向每个新版本添加新特性。这可能会导致软件速度变慢或需要更多的资源，即使用户从不使用过这些新特性。

11. 认真对待用户反馈。用户的抱怨应该被视为关于bug、兼容性问题、可用性问题和所需新特性的有价值的信息来源。系统地处理用户反馈，确保信息得到合理利用。用户应该得到关于问题调查和解决方案计划的回复。应该很容易从网站上获得补丁。

5 选择最优算法

要优化CPU密集型软件，首先要找到最佳算法。算法的选择对于排序、搜索和数学计算等任务非常重要。在这种情况下，选择最好的算法比优化想到的第一个算法，你可以得到更多提升。在某些情况下，您可能需要测试几种不同的算法，以便找到在一组典型测试数据上最有效的算法。

话虽如此，我必须提醒不要过度。如果一个简单的算法可以足够快地完成这项工作，就不要使用高级和复杂的算法。例如，一些程序员甚至使用哈希表来存储很小的数据列表。对于非常大的数据库，哈希表可以显著地提高搜索时间，但是对于使用二分搜索甚至线性搜索的都可以很快完成列表，这就没有理由使用它。哈希表增加了程序的大小和数据文件的大小。如果瓶颈是文件访问或缓存访问，而不是CPU时间，这实际上会降低速度。复杂算法的另一个缺点是，它使程序开发成本更高，而且更容易出错。

对于不同目的的不同算法的讨论超出了本手册的范围。您必须查阅关于标准任务（如排序和搜索）的算法和数据结构的一般文献，或针对更复杂的数学任务的特定文献。

在开始编写代码之前，您可以考虑其他人是否已经在您之前完成了这项工作。针对许多标准任务的优化函数库有许多种来源。例如，*Boost*包含许多常用的经过良好测试的库。"*Intel Math Kernek Library*"包含许多函数用于常见的数学计算包括线性代数和统计学，以及"Intel Performance Primitives"库包含许多含函数用于音频和视频处理，信号处理、数据压缩和密码学（www.intel.com）。如果你正在使用*Intel*函数库，参见133页，确保在非*Intel*处理器上运作良好。

在开始编程之前选择最优算法通常说起来容易做起来难。许多程序员已经发现，只有在他们将整个软件项目放在一起并对其进行测试之后，才有更明智的方法来处理问题。通过测试和分析程序性能以及研究瓶颈获得的见解可以更好地理解问题的整个结构。这种新的见解可以导致程序的完全重新设计，例如，当你发现有更好的方法来组织数据时。

对一个已经投入使用的的程序进行彻底的重新设计当然是一项相当大的工作，但这可能是一项相当好的投资。重新设计不仅可以提高性能，可以得到更易于维护的、结构良好的程序。实际上，你花在重新设计程序上的时间可能比您花在解决原来设计糟糕的程序的问题上的时间要少。

6 开发过程

关于使用哪种软件开发过程和软件工程原则存在着相当多的争论。我不打算推荐任何具体的模型。相反，我将对开发过程如何影响最终产品的性能发表一些评论。

在规划阶段，最好对数据结构、数据流和算法进行全面的分析，以预测哪些资源是最关键的。然而，在早期规划阶段可能有许多未知的因素，因此很难对问题进行详细的概述。在后一种情况下，您可以将软件开发工作视为一个学习过程，其中主要的反馈来自于测试。在这里，您应该为多次迭代的重新设计做好准备。

一些软件开发模型具有严格的形式，它要求在软件的逻辑架构中有几个抽象层。您应该知道，这种形式主义有其固有的性能成本。将软件分割成过多的抽象层是降低性能的常见原因。

由于大多数开发方法本质上都是增量的或迭代的，所以一定要有一种策略来保存每个中间版本的备份。对于单人项目，制作每个版本的zip文件就足够了。对于团队项目，建议使用版本控制工具。

7 不同C++结构的效率

大多数程序员对于如何将一段程序代码转换成机器码以及微处理器如何处理这些代码，了解很少或者根本不了解。例如，许多程序员不知道双精度计算和单精度计算一样快。谁会知道模板类比多态类更有效呢?本章旨在解释不同*C++*语言元素的相对效率，以帮助程序员选择最有效的替代方案。本系列手册的其他卷进一步解释了理论背景。

7.1 不同类型的变量存储

变量和对象存在内存的存储位置，取决于它们在C++程序中是如何声明的。这会影响数据缓存的效率(参见第89页，TODO)。如果数据在内存中随机分布，则数据缓存的效率很低。因此，理解变量是如何存储的非常重要。对于简单的变量、数组和对象，存储原则是相同的。

栈存储（Storage on the stack）

函数中声明的变量和对象存储在栈中，但以下描述的几种情况除外。

栈是内存的一部分，以先入后出的方式组织。它用于存储函数返回地址（即函数是从哪里调用的）、函数参数、局部变量，以及保存在函数返回之前必须恢复的寄存器。每次调用函数时，它都会为所有这些目的在栈上分配所需的空间。当函数返回时释放该内存空间。下一次调用函数时，新函数的参数可以使用相同的空间。

因为重复使用相同范围的地址，栈是用来存储数据的效率最高的内存空间了。如果没有很大的数组，那么这一部分数据基本是缓存在一级缓存中的，而这里的访问速度是非常快的。

我们从中可以学到的教训是，所有变量和对象最好在使用它们的函数中声明。

通过在`{}`中声明变量，可以使变量的作用域更小。然而，大多数编译器在函数返回之前不会释放变量所使用的内存，即使在退出声明变量的`{}`时可以释放内存。如果变量存储在寄存器中（见下文），那么它可能在函数返回之前被释放。

全局或静态存储（Global or static storage）

在任何函数之外声明的变量称为全局变量。可以在任何函数中访问它们。全局变量存储在内存的静态部分中。静态内存还用于使用`static`关键字声明的变量、浮点常量、字符串常量、数组初始化器列表、`switch`语句跳转表和虚拟函数表。

静态数据区域通常分为三部分：一部分用于程序从不修改的常量，一部分用于程序可能会修改的初始化变量，另一部分用于程序可能会修改的未初始化变量。

静态数据的优点是可以在程序启动之前将其初始化为所需的值。缺点是内存空间在整个程序执行过程中都被占用，即使变量只在程序的一小部分中使用。这会降低数据缓存的效率。

如果可以避免，就不要定义全局变量。不同线程之间的通信可能需要全局变量，但这是惟一不可避免的情况。如果一个变量被多个不同的函数访问，或者你希望避免将变量做为函数的参数的额外开销，那么让它成为全局变量可能是有用的。但是，将需要访问相同变量的函数作为同一个类的成员函数并在类中保存共享的变量可能是一个更好的方案。当然你喜欢用你一种方案是编程分割的问题。

通常最好将查找表声明为`static`，例如

```
// Example 7.1
float SomeFunction (int x) {
    static float list[] = {1.1, 0.3, -2.0, 4.4, 2.5};
    return list[x];
```

在这里使用`static`的优点是在调用函数时不需要初始化这个数组。当程序加载到内存中时，这些值机会被放在那里。如果将上面的示例中的`static`这个词去掉，那么每次调用函数时都必须重新将所有这五个值放入数组中。这是通过将整个列表从静态内存复制到堆栈内存来完成的。在大多数情况下，从静态内存中复制常数数据到栈是在浪费时间。但这在特殊情况下可能是最好的：在循环中多次使用数据，而一级缓存已经被很多数组占用了，这时你可能会想把数据保存在栈上（, but it may be optimal in special cases where the data are used many times in a loop where almost the entire level-1 cache is used in a number of arrays that you want to keep together on the stack）。

字符串常量和浮点常量在优化后的代码中，被存储在静态内存中。例如：

```
// Example 7.2
a = b * 3.5;
```

在这里，常量`3.5`将会被存储在静态内存中。大多数编译器将识别出这两个常量是相同的，因此只需要存储一个常量。将整个程序中所有相同的常量连接在一起，以最小化常量使用的缓存空间。

整数常量通常包含在指令代码中。你可以假设整数常量不存在缓存问题。

寄存器存储（Register storage）

有限数量的变量可以存储在寄存器中而不是主存中。寄存器是CPU中用于临时存储的一小块内存。存储在寄存器中的变量可以被快速访问。所有优化编译器都会自动选择一个函数中最常用的变量存储在寄存器中。同一个寄存器可以用于多个变量，只要它们的使用（生存周期）不重叠。

寄存器的数量非常有限。在32位操作系统中，大约有6个整数寄存器可用于一般用途，而在64位系统中，有14个整数寄存器。

浮点变量使用不同类型的寄存器。32位操作系统中有8个浮点寄存器，64位操作系统中有16个浮点寄存器。 一些编译器很难在32位模式下生成浮点寄存器变量，除非启用了SSE2指令集（或更高版本）。

易变变量（Volatile）

volatile关键字用于声明一个变量可变被其它线程改变。这阻止编译器依赖于变量始终具有代码中先前分配的值的假设而进行优化。例如:

```
// Example 7.3. Explain volatile
volatile int seconds; // incremented every second by another thread
void DelayFiveSeconds() {
    seconds = 0;
    while (seconds < 5) {
    // do nothing while seconds count to 5
    }
```

在本例中，DelayFiveSeconds函数将等待，直到另一个线程将秒数增加到5。如果seconds没有声明为volatile，那么具有优化功能的编译器将假设秒在while循环中保持为零，因为循环中没有任何东西可以更改该值。循环while(0 < 5){}，将是一个无限循环。

要注意到volatile并不意味着原子性（atomic）。它不阻止两个线程同时改变变量。如果上面示例中的代码试图在其他线程增加seconds的同时将seconds置为零，那么它可能会失败。更安全的实现时只读取seconds的值，并等待该值更改五次。

线程本地存储（Thread-local storage）

大多数编译器可以使用关键字__thread或__declspec(thread)来实现静态变量和全局变量的线程本地存储。这样的变量对于每个线程都有一个实例。线程本地存储是低效的，因为它是通过存储在线程环境块中的指针进行访问的。如果可能的话，应该避免线程本地存储，并将其替换为栈上的存储（参见上文）。存储在栈中的变量总是属于创建它们的线程。

Far

具有分段内存的系统，如*DOS*和16位*Windows*，允许使用关键字far（数组也可以用huge声明）将变量存储在其它数据段中。far存储、far指针和far过程是低效的。如果一个程序对于一个段（内存）有太多的数据，那么建议使用允许更大段的不同操作系统(32位或64位系统)。

动态内存分配

动态内存分配由new和delete操作符或malloc和free函数完成。这些操作符和函数消耗大量时间。内存中称为堆的一部分保留给动态分配。当以随机顺序分配和释放不同大小的对象时，堆很容易变得碎片化。堆管理器可以花费大量时间清理不再使用的空间并搜索空闲空间。这称为垃圾收集。按顺序分配的对象不一定按顺序存储在其中。当堆变的碎片化时，它们可能分散在不同的地方。这使得数据缓存效率低下。

动态内存分配还会使代码更复杂，更容易出错。程序必须保留指向所有已分配对象的指针，并跟踪它们何时不再使用。重要的是，在程序流的所有可能的情况下，也要释放所有分配的对象。不这样做是一个常见的导致错误的原因，称为内存泄漏。更糟糕的一种错误是在释放对象之后访问该对象。程序逻辑可能需要额外的开销来防止此类错误。

有关使用动态内存分配的优点和缺点的进一步讨论，请参见第92页（TODO）。

一些编程语言，如Java，对所有对象使用动态内存分配。这当然是低效的。

类中声明的变量

类中声明的变量按照它们在类声明中出现的顺序存储。存储类型由在哪里声明类的对象的决定。类、结构或联合的对象可以使用上面提到的任何存储方法。除了在最简单的情况下，对象不能存储在寄存器中，但是它的数据成员可以复制到寄存器中。

带有static修饰符的类成员变量将存储在静态内存中，并且只有一个实例。同一类的非静态成员将存储在该类的每个实例中。

将变量存储在类或结构中是一种很好的方法，可以确保在程序的相同部分中使用的变量也存储在彼此附近。使用类的优点和缺点见第52页(TODO)。

7.2 整数变量和运算符

整数大小

整数可以是不同的大小，可以有符号也可以无符号。下表总结了可用的不同整数类型。

delaration	size, bits	minimum value	maximum value	in stdint.h
char	8	-128	127	int8_t
shortint in 16-bit system: int	16	-32768	32767	int16_t
int in 16-bit system: long int	32	-2^{31}	$2^{31}-1$	int32_t
long long or int64_t MS compiler:__int64 64-bit Linux: long int	64	-2^{63}	$2^{63}-1$	int64_t
unsigned char	8	0	255	uint8_t
unsigned short int in 16-bit system: unsigned int	16	0	65535	uint16_t
unsigned int in 16-bit system: unsigned long	32	0	$2^{32}-1$	uint32_t
unsigned long long or uint64_t MS compiler: unsigned __int64 64-bit Linux: unsigned long int	64	0	$2^{64}-1$	uint64_t

Table 7.1 Sizes of different integer types

不幸的是，对于不同的平台，声明特定大小的整数的方式是不同的，如上表所示。如果标准头文件stdin .h或inttypes.h可用，则建议使用，以可移植的方式定义特定大小的整数类型。

无论大小如何，整数运算在大多数情况下都是很快的。但是，使用大于最大可用寄存器大小的整数大小是低效的。换句话说，在16位系统中使用32位整数或在32位系统中使用64位整数效率很低，特别是在代码涉及乘法或除法的情况下。

如你定义一个int类型，而不指定该类型的大小，编译器将始终选择最有效的整数大小。较小大小的整数（char，short int)的效率稍微低一些。在许多情况下，编译器在进行计算时将这些类型转换为默认大小的整数，然后只使用结果中较低的8或16位。您可以假设类型转换需要0或1个时钟周期。在64位系统中，只要不进行除法，32位整数和64位整数的效率之间只有极小的差别。

建议在大小无关且没有溢出风险的情况下使用默认整数大小，例如简单变量、循环计数器等。在大型数组中，为了更好地使用数据缓存，最好使用对于特定用途来说足够大的最小整数大小。大小不同于8、16、32和64位的位域（Bit-fields）效率较低。在64位系统中，如果应用程序可以使用额外的位，那么可以使用64位整数。

无符号整数类型size_t在32位系统中是32位，在64位系统中是64位。当您希望确保永远不会发生溢出时（即使是对于大于2 GB的数组），它可以被用于数组大小和数组索引。

在考虑特定整数大小是否足够大以满足特定用途时，必须考虑中间计算是否会导致溢出。例如，在表达式$a = (b*c)/d$中，即使a、b、c和d都低于最大值，也可能发生（$b*c$）溢出。这里没有对整数溢出的自动检查。

有符号整数 VS 无符号整数

在大多数情况下，使用有符号整数和无符号整数在速度上没有区别。但在一些情况下会有一些区别：

1. 除以常数：当你用一个整数除一个常数时，无符号要快于有符号（参见第141页:TODO）。这也适用于模运算符%。
2. 对于大多数指令集，有符号整数转换为浮点数要比无符号整数快(参见第145页)。
3. 有符号变量和无符号变量的溢出行为不同。无符号变量的溢出会产生较低的正数结果。带符号变量的溢出没有被正式定义。正常的行为下，正溢出将会变为负值，但是编译器可以基于不会发生溢出的假设，优化掉依赖于溢出的分支。

有符号整数和无符号整数之间的转换是无代价的。这仅仅是对相同的位进行不同的解释。负整数在转换为无符号时将被解释为一个非常大的正数。

```
// Example 7.4. Signed and unsigned integers
int a, b;
double c;
```

```
b = (unsigned int)a / 10; // Convert to unsigned for fast division
```

在示例7.4中，我们将a转换为无符号，以使除法更快。当然，这只在确定a永远不会为负的情况下有效。最后一行是将a隐式转换为double然后乘以常数2.5，也是double类型的。在这里我们希望a是有符号的。

确保不要在比较中混合有符号整数和无符号整数，例如<。比较有符号整数和无符号整数的结果是模糊的，可能会产生不希望的结果。

整数操作符

整数运算通常非常快。在大多数微处理器上，简单的整数操作（如加减、比较、位操作和移位操作）只需一个时钟周期。

乘法和除法需要更长的时间。在奔腾4处理器上，整数乘法需要11个时钟周期，在大多数其他微处理器，需要3-4个时钟周期。整数除法需要40 - 80个时钟周期，具体取决于微处理器。整数除法在AMD处理器上整数大小（size）越小速度越快，但在英特尔处理器上不会这样。关于指令延迟的详细信息列在手册4:"Instruction tables"中。关于如何加速乘法和除法的技巧，分别在第139页和第141页给出（TODO）。

自增和自减运算符

增量（前缀）运算符++i和增量（后缀）运算符i++和加法一样快。当仅用于递增整数变量时，使用递增前或递增后都没有区别。效果完全相同。例如,for (i=0; i<n; i++)和for (i=0; i<n; ++i)是一样的。：但是当使用表达式的结果时，效率可能会有所不同。例如，x = array[i++]比x = array[++i]更有效率，因为在后一种情况下，数组元素的地址的计算必须等待i的新值，这将使x的可用性延迟大约两个时钟周期。显然，如果将增量（前缀）更改为增量（后缀），则必须调整i的初始值。

还有一些情况下，增量（前缀）比增量（后缀）更有效。例如，在a = ++b的情况下；编译器会在这条语句之后识别出a和b的值是相同的，这样它就可以对两者使用相同的寄存器，而表达式a = b++，将使a和b的值不同，这样它们就不能使用相同的寄存器。

这里所说的关于递增运算符的所有内容也适用于整数变量上的递减运算符。

7.3 浮点变量和操作符

*x86*家族中的现代微处理器有两种不同类型的浮点寄存器，相应地也有两种不同类型的浮点指令。每种类型都有各有优缺点。

进行浮点运算的原始方法涉及到将8个浮点寄存器组成一个寄存器栈（register stack）。这些寄存器具有长双精度（80位）。使用寄存器栈的优点是:

1. 所有的计算都是长双精度的。
2. 不同精度之间的转换不需要额外的时间。
3. 对于数学函数，如对数函数和三角函数，有一些指令可用。
4. 码很紧凑，在代码缓存中占用的空间很小。

寄存器堆栈也有缺点：

1. 由于寄存器堆栈的组织方式，编译器很难生成寄存器变量。
2. 浮点数比较比较慢，除非使用奔腾-II或者更新的指令集。
3. 整数和浮点数之间的转换效率很低。
4. 当使用长双精度时，除法、平方根和数学函数需要更多的时间。

还有一种新的浮点运算方法涉及8个或16个向量寄存器（*XMM*或*YMM*），可用于多种用途。浮点运算以单精度或双精度进行，中间结果的计算精度始终与操作数相同。使用向量寄存器的优点是:

1. 浮点寄存器变量很容易实现
2. 矢量运算可用于对*XMM*寄存器中两个双精度或四个单精度变量的矢量进行并行计算(参见第107页,TODO)。如果*AVX*指令集是可用的，那么在YMM寄存器中每个向量可以容纳4个双精度或8个单精度变量。

缺点是：

1. 不支持长双精度
2. 在运算数具有混合精度的表达式的计算需要精确的转换指令，这可能非常耗时（参见第143页，TODO）。
3. 数学函数必须使用函数库，但这通常比硬件函数更快。

浮点堆栈寄存器在所有具有浮点功能的系统中都可用（64位*Windows*的设备驱动程序除外）。*XMM*向量寄存器可以在64位系统中使用，也可以在启用*SSE2*或更高的指令集时在32位系统中使用(单精度只需要*SSE*)。如果处理器和操作系统支持*AVX*指令集，则可以使用*YMM*寄存器。有关如何测试这些指令集的可用性，请参见第125页(TODO)。

当*XMM*寄存器可用时，大多数编译器都会使用它进行浮点计算，例如在64位系统中，或者启用*SSE2*指令集时。很少有编译器能够混合这两种类型的浮点运算，并为每种计算选择最优的类型。

在大多数情况下，双精度计算不会比单精度计算花费更多的时间。当使用浮点寄存器时，单精度和双精度的速度没有差别。长双精度只需要稍多一点的时间。单精度的除法、平方根和数学函数的计算速度都快于双精度。当使用*XMM*寄存器时，加、减、乘等操作的速度，无论精度如何，在大多数处理器上仍是相同的（没有使用向量操作）。

如果对应用程序有好处，可以使用双倍精度，而不必太担心成本。如果您有大数组，并且希望将尽可能多的数据放入数据缓存，则可以使用单精度。如果可以利用向量操作，单精度是最好的，如107页所述（TODO）。

根据微处理器的不同，浮点加法需要3 - 6个时钟周期。乘法需要4 - 8个时钟周期。除法需要14 - 45个时钟周期。当使用浮点堆栈寄存器时，浮点比较是低效的。在使用浮点堆栈寄存器时，浮点或双精度浮点到整数的转换需要很长时间。

当使用*XMM*寄存器时，不要混合使用单精度和双精度。见143页（TODO）。

如果可能的话，避免整数和浮点变量之间的转换。见144页（TODO）。

在*XMM*寄存器中长生浮点浮点数向下溢出的应用程序，可以从`flush-to- zero`模式而不是在向下溢出的情况下生成非规格化数（*subnormal number*）中获益:

```
// Example 7.5. Set flush-to-zero mode (SSE):
#include <xmmintrin.h>
```

强烈建议设置为`flush-to-zero`模式，除非有特殊原因需要使用非规格话数。此外，如果*SSE2*可用，你可以设置`denormars-are-zero`模式:

```
// Example 7.6. Set flush-to-zero and denormals-are-zero mode (SSE2):
#include <xmmintrin.h>
```

有关数学函数的更多信息，请参阅第149和122页(TODO)。

7.4 枚举

`enum`只是一个隐藏的整数。枚举的效率和整数一样。注意，枚举数（值名）将与具有相同名称的任何变量或函数冲突。因此，头文件中的枚举应该具有长且唯一的枚举数名称，或者将其放在命名空间中。

7.5 布尔值

布尔操作数的顺序

布尔运算符`&&`和`||`的操作数将会按照下面的顺序就行计算。如果`&&`的第一个操作数为`false`，那么就不会计算第二个操作数的值，因为表达式的结果无论第二个操作数的值是`true`还`false`，都为`false`。类似的，如果`||`的第一个操作数为`true`，那么也不会计算第二个操作数的值，因为无论如何结果都为`true`。

将通常为`true`的操作数放在`&&`表达式的最后，或者作为`||`表达式的第一个操作数中，这可能是有好处的。例如，假设`a`在50的情况下为真`b`在10的情况下为真。当`a`为真时，表达式`a && b`需要对`b`求值，即 50的情况。等价表达式`b && a`只需要在`b`为`true`时对`a`求值，只有10%的情况。如果`a`和`b`的计算时间相同，并且分支预测机制预测的可能性相同，这样的计算速度会更快。有关分支预测的解释，请参见第43页（TODO）。

如果一个操作数比另一个更可预测，那么将最可预测的操作数放在前面。

如果一个操作数的计算速度快于另一个操作数，则将计算速度最快的操作数放在首位。

但是，在交换布尔操作数的顺序时必须小心。如果操作数的求值有副作用，或者如果第一个操作数决定第二个操作数是否有效，则不能交换操作数。例如:

```
// Example 7.7
unsigned int i;
const int ARRAYSIZE = 100;
float list[ARRAYSIZE];
```

在这里，您不能交换操作数的顺序，因为当`i`不小于`ARRAYSIZE`时，表达式`list[i]`是无效的。另一个例子:

```
// Example 7.8
```

布尔变量被过度检查（overdetermined）

布尔变量存储在8位整数中，0代表false, 1代表true。

布尔变量是过度检查的，因为所有以布尔变量作为输入的运算符都要检查输入是否有除0或1之外的值，但是以布尔值作为输出的运算符只能产生0或1之外的值。这使得使用布尔变量作为输入的操作效率低于可能的效率。比如说：

```
// Example 7.9a
bool a, b, c, d;
c = a && b;
```

编译器通常是以以下方式实现：

```
bool a, b, c, d;
if (a != 0) {
    if (b != 0) {
        c = 1;
    }
    else {
        goto CFALSE;
    }
}
else {
CFALSE:
    c = 0;
}
if (a == 0) {
    if (b == 0) {
        d = 0;
    }
    else {
    goto DTRUE;
    }
}
else {
DTRUE:
    d = 1;
```

这当然远远不是最优的。为了防止错误的预测，这些分支可能耗费很长的时间（详见第43页，TODO）。如果知道操作数除了0就是1，布尔运算可以变得有效率的多。编译器之所以不这样假设，是因为如果变量是没有被初始化的或者是来自其它未知来源的。如果a和b被初始化为有效的值，或者他们来自输出 Boolean 值的操作符，那么上述代码是可以被优化的。优化后的代码类似下面这样：

```
// Example 7.9b
char a = 0, b = 0, c, d;
c = a & b;
```

在这里我使用char（或int）来代替bool,是为了使用位操作符（&和|）代替布尔操作符（&&和||）。位操作符是单条指令，只需一个时钟周期。即使a和b不是0或1，OR操作符（|)也可以正常工作。如果操作数有不是0或1的值，AND（&）操作符和EXCLUSIVE OR操作符（^）可能会产生不一致的结果。

请注意到这里有几个陷阱（pitfalls）。你不能使用~代替NOT。相反，如果已知变量的值为0或1，你可以对变量XOR上1，来对变量进行取反操作（*Boolean NOT*）。

```
// Example 7.10a
bool a, b;
```

可以被优化为：

```
// Example 7.10b
char a = 0, b;
```

如果打a为*false*时，表达式不应该被计算的话，你不能使用a&b代替a&&b。类似的，如果当a为*true*时，表达式不应该被计算的话，不能用a|b代替a||b。

使用位操作符的技巧在操作数时变量而不是比较表达式等其它情况时，更有优势。例如：

```
// Example 7.11
bool a; float x, y, z;
```

上述形式在大多数情况下通常是最优的。不要把&&改成&，除非你希望&&表达式产生很多错误的分支预测。

布尔向量操作：

一个整数可以被当作布尔向量使用。例如，如果a和b是32位整数，那么表达式y=a&b，将会在一个时钟周期中进行32个与操作。操作符&，|，^，~对于布尔向量操作都非常常用。

7.6 指针和引用

指针 VS 引用

指针和引用的效率是一样的，因为它们实际上做的事情是相同的。例如：

```
// Example 7.12
void FuncA (int * p) {
    *p = *p + 2;
}
void FuncB (int & r) {
    r = r + 2;
```

这两个函数做的是相同的事情，如果您查看编译器生成的代码，您会注意到这两个函数的代码完全相同，区别仅仅在于编程风格。使用指针的优点是:

1. 当你查看上面的函数体时，可以清楚地看到p是一个指针，但是不清楚r是一个引用还是一个简单的变量。使用指针使读者更清楚地了解正在发生的事情。
2. 可以用指针做引用不可能做的事情。你可以改变指针指向什么，你可以用指针做算术运算。

使用引用的优点是：

1. 使用引用时语法更简单。
2. 引用比指针更安全，因为在大多数情况下，它们肯定指向一个有效的地址。如果指针没有初始化，或者指针算术计算超出了有效地址的范围，亦或是指针被转换为错误的类型，指针可能无效并导致致命错误。
3. 对于复制构造函数和重载操作符来说，引用更加常用。
4. 被声明为常量引用的函数参数接受表达式作为参数，而指针和非常量引用则需要一个变量。

效率

通过指针或引用访问变量或对象可能与直接访问它一样快。拥有这种效率的原因在于微处理器的构造方式。函数中声明的所有非静态变量和对象都存储在堆栈中，并且实际上是相对于栈指针寻址的。同样，我们知道在*C++*中。类中声明的所有非静态变量和对象都可以通过已知的隐式指针“this”进行访问。因此，我们可以得出这样的结论：结构良好的*C++*程序中的大多数变量实际上都是通过指针这种方式访问的，这就是效率相同的原因。然而，使用指针和引用也有缺点。最重要的是，它需要一个额外的寄存器来保存指针或引用的值。寄存器是一种稀缺资源，尤其是在32位模式下。如果没有足够的寄存器，那么指针每次使用时都必须从内存中加载，这会使程序变慢。另一个缺点是指针的值需要几个时钟周期才能访问所指向的变量。

指针运算

指针实际上是一个保存内存地址的整数。因此，指针算术运算和整数算术运算一样快。当一个整数被添加到一个指针时，它的值乘以所指向的对象的大小。例如:

```
// Example 7.13
struct abc {int a; int b; int c;};
abc * p; int i;
```

在这里，p所增加的值不是i而是i*12，因为abc的大小是12个字节。p加上i的时间等于做乘法和加法的时间的和。如果abc的大小是2的幂，那么乘法可以被移位运算代替，移位运算要快得多。在上面的示例中，通过向结构体中添加一个整数，abc的大小可以增加到16字节。

递增或递减指针不需要乘法，只需要加法。比较两个指针只需要一个整数比较，这是很快的。计算两个指针之间的差值需要一个除法，这是很慢的，除非所指向的对象类型的大小是2的幂（关于除法，请参阅第141页，TODO）。

在计算指针的值之后，访问所指向的对象大约需要两个时钟周期。因此，建议在使用指针之前计算该指针的值。例如，x = *(p++)比x = *(++p)更高效，因为在后一种情况下，x的读取必须在指针p被递增后等待几个时钟周期，而在前一种情况下，x可以在p在递增之前被读取。有关递增和递减运算符的更多讨论，请参见7.2节关于递增和递减运算的讨论。

7.7 函数指针

如果可以预测目标地址，那么通过函数指针调用函数通常要比直接调用函数多花几个时钟周期。如果函数指针的值与上次执行语句时相同，则可以预测目标地址。如果函数指针的值发生了变化，那么目标地址很可能会被错误地预测，从而导致长时间的延迟。有关分支预测，请参见第43页（TODO）。如果函数指针的变化遵循一个简单的规则，奔腾M处理器可能能够预测目标地址，而奔腾4和AMD处理器一定会做出错误的预测，只要函数指针发生了变化。

7.8 成员指针（Member pointers）

在简单的情况下，数据成员指针只是存储数据成员相对于对象开头的偏移量，而成员函数指针只是成员函数的地址。但是在一些特殊的情况下，比如需要更复杂实现的多重继承。一定要避免这些复杂的情况。

如果编译器没有关于成员指针所引用的类的完整信息，那么它必须使用最复杂的成员指针实现。例如:

```
// Example 7.14
class c1;
```

在这里，声明MemberPointer时，编译器除了类c1的名称之外，没有其他关于类c1的信息。因此，它必须假设最坏的情况，并对成员指针进行复杂的实现。可以通过在声明MemberPointer之前完整地声明c1来避免这种情况。避免*多重继承*、*虚函数*和其他会降低成员指针效率的复杂情况。

大多数C++编译器都有不同的选项来控制成员指针的实现方式。如果可能的话，使用尽可能简单的实现选项，并确保对使用相同成员指针的所有模块使用相同的编译器选项。

7.9 智能指针

智能指针是一个行为像指针的对象。它有一个特殊的特性，当指针被删除时，它所指向的对象被删除。智能指针仅用于使用new存储在动态分配内存中的对象。使用智能指针的目的是确保对象被正确删除，以及在对象不再使用时释放内存。智能指针可以被认为是只包含单个元素的容器。

智能指针最常见的实现是auto_ptr和shared_ptr。auto_ptr的特性是，始终只有一个auto_ptr拥有所分配的对象，并且通过赋值将所有权从一个auto_ptr转移到另一个auto_ptr。shared_ptr则允许多个指针指向同一个对象。

通过智能指针访问对象没有额外的成本。无论p是简单指针还是智能指针，通过*p或p->member访问对象的速度都是一样快的。但是，每当创建、删除、复制或从一个函数转移到另一个函数时，都会产生额外的成本。shared_ptr的这些成本要高于auto_ptr。

当程序的逻辑结构要求一个对象必须由一个函数动态创建，然后由另一个函数删除，并且这两个函数互不相关（不是同一个类的成员）时，智能指针非常有用。如果相同的函数或类负责创建和删除对象，则不需要使用智能指针。

如果一个程序为每个智能指针使用许多动态分配的小对象，那么需要考虑一下这个解决方案的成本是否太高。将所有对象合用到一个容器中可能更有效，最好是使用连续内存。参见第95页（TODO）关于容器类的讨论。

7.10 数组

数组是通过在内存中连续存储元素来实现的。没有存储关于数组大小的信息。这使得在C和C++中使用数组比在其他编程语言中更快，但也更不安全。这个安全问题可以通过定义一个容器类来解决，该类的行为类似于一个带有边界检查的数组，如下例所示:

```
// Example 7.15a. Array with bounds checking
template <typename T, unsigned int N> class SafeArray {
    protected:
        T a[N]; // Array with N elements of type T39
    public:
        SafeArray() { // Constructor
            memset(a, 0, sizeof(a)); // Initialize to zero
        }
        int Size() { // Return the size of the array
            return N;
        }
        T & operator[] (unsigned int i) { // Safe [] array index operator
            if (i >= N) {
                // Index out of range. The next line provokes an error.
               // You may insert any other error reporting here:
                return *(T*)0; // Return a null reference to provoke error
            }
            // No error
            return a[i]; // Return reference to a[i]
        }
```

www.agner.org/optimize/cppexamples.zip中给出更多关于容器类的例子。

使用上述模板类的数组是通过将类型和大小指定为模板参数来声明的，如下面的示例7.15b所示。可以使用方括号索引访问它，就像普通数组一样。构造函数将所有元素设置为零。如果你不希望这个初始化，或者类型T是一个具有默认构造函数的类，它会执行必要的初始化，那么可以删除memset这一行。编译器可能会报告memset已被弃用。这是因为如果参数size错误，它会导致错误，但它仍然是将数组设置为0的最快方法。如果索引超出范围，[]操作符将检测到错误（参见第137页的边界检查，TODO）。在这里，通过返回空引用这一非常非常规的方式引发错误消息。如果数组元素被访问，这将在受保护的操作系统中引发错误消息，并且这个错误很容易通过调试器跟踪。您可以用任何其他形式的错误报告来替换这一行。例如，在*Windows*中，你可以这么写：`FatalAppExitA(0, "Array index out of range");`，或者更好的是，创建自己的错误消息函数。

下面的例子演示了如何使用SafeArray：

```
// Example 7.15b
SafeArray <float, 100> list; // Make array of 100 floats
for (int i = 0; i < list.Size(); i++) { // Loop through array
     cout << list[i] << endl; // Output array element
```

由列表初始化的数组最好是静态的，如7.1节，*全局或静态存储*中所述。数组可以使用memset初始化为0：

```
// Example 7.16
float list[100];
```

应该按顺序访问多维数组，保证最后一个索引变化最快（“A multidimensional array should be organized so that the last index changes fastest”）：

```
// Example 7.17
const int rows = 20, columns = 50;
float matrix[rows][columns];
int i, j; float x;
for (i = 0; i < rows; i++)
```

```
for (j = 0; j < columns; j++)
```

这确保元素是按顺序访问的。这两个循环的相反顺序将使访问是非顺序的，这将降低数据缓存的效率。如果为了使地址计算更高效，不按顺序索引，那么除了第一个维度外，所有维度的大小最好是2的幂：

```
// Example 7.18
int FuncRow(int); int FuncCol(int);
const int rows = 20, columns = 32;
float matrix[rows][columns];
int i; float x;
for (i = 0; i < 100; i++)
```

在这里，代码必须计算`(FuncRow(i)*columns + FuncCol(i)) * sizeof(float)`才能找到矩阵元素的地址。在这种情况下，当大小是2的幂时，乘上列数的速度更快。在前面的示例中，这不是问题，因为编译器优化可以看到这些行是连续访问的，并且可以通过将行长度添加到前一行的地址来计算每行的地址。

同样的建议也适用于结构或类对象的数组。如果以非顺序访问元素，则对象的大小(以字节为单位)最好为2的幂。

将列数设置为2的幂的建议并不总是适用于大于一级数据缓存且以非顺序访问的数组，因为这可能会导致缓存竞争。关于这个问题的讨论请参阅第89页（TODO）。

7.11 类型转换

C++语法有几种不同的类型转换方法：

```
// Example 7.19
int i; float f;
f = i; // Implicit type conversion
f = (float)i; // C-style type casting
f = float(i); // Constructor-style type casting
```

这些不同的方法有完全相同的效果。使用哪种方法取决于编程风格。下面讨论不同类型转换的时间消耗。

<u>signed/unsigned 转换</u>

```
// Example 7.20
int i;
```

有符号整数和无符号整数之间的转换只是使编译器换一种解释整数的不同位的方法。没有检查溢出，代码没有消耗额外的时间。这些转换可以随意使用，而不需要担心任何性能成本。

<u>整型大小的转换</u>

```
// Example 7.21
int i; short int s;
```

如果整数有符号，则通过扩展符号位将其转换为更大的大小；如果没有符号，则通过扩展0将其转换为更大的大小。如果代码是算术表达式，这通常需要一个时钟周期。如果是从内存中的变量中读取值，则大小转换通常不需要额外的时间，如示例7.22所示：

```
// Example 7.22
short int a[100]; int i, sum = 0;
```

将整数转换成较小的大小只需忽略较高的位即可。没有溢出检查。例如：

```
// Example 7.23
int i; short int s;
```

这种转换不需要额外的时间。它只存储32位整数中较低的16位。

<u>浮点精度转换</u>

当使用浮点寄存器栈时，浮点、双精度和长双精度之间的转换不需要额外的时间。使用*XMM*寄存器时，需要2到15个时钟周期（取决于处理器）。有关寄存器栈与*XMM*寄存器的区别，请参见*7.3 浮点变量和操作符*。例如：

```
// Example 7.24
float a; double b;
```

在本例中，如果使用*XMM*寄存器，那么转换的成本很高。为了避免这种情况，a和b应该是同一类型的。进一步讨论请参阅第143页(TODO)。

<u>整型转换为浮点型</u>

将有符号整数转换为浮点数或双精度浮点数需要4 - 16个时钟周期，这取决于处理器和使用的寄存器类型。无符号整数的转换需要更长的时间。如果没有溢出的风险，首先将无符号整数转换为有符号整数会更快：

```
// Example 7.25
unsigned int u; double d;
```

整型到浮点型的转换有时可以通过将整数替换为浮点型变量来避免。例如:

```
// Example 7.26a
float a[100]; int i;
```

在本例中，可以通过添加一个浮点变量来避免i转换为float:

```
// Example 7.26b
float a[100]; int i; float i2;
for (i = 0, i2 = 0; i < 100; i++, i2 += 2.0f)
```

<u>浮点型转换为整型</u>

如果不启用*SSE2*或者更新的指令集，浮点数到整数的转换将花费很长时间。通常，转换需要50 - 100个时钟周期。原因是*C/ C++*标准指定了截断，因此浮点四舍五入的模式必须更改为截断并再次返回。如果在代码的关键部分存在浮点数到整数的转换，那么对其采取一些措施是很重要的。可能的解决方案是：

1. 使用不同类型的变量避免转换。
2. 通过将中间结果存储为浮点数，将转换移出最内层循环。
3. 使用64位模式或启用*SSE2*指令集（需要一个支持该模式的微处理器）。
4. 使用四舍五入代替截断，并用汇编语言制作一个舍入函数。有关舍入的详细信息，请参见第144页（TODO）。

<u>指针类型转换</u>

指针可以转换为另一种类型的指针。同样，可以将指针转换为整数，也可以将整数转换为指针。重要的是整数有足够的位来保存指针。

这些转换不会产生任何额外的代码。这仅仅是用不同的方式解释相同的位或者绕过语法检查的问题。

当然，这些转换是不安全的。程序员有责任确保结果是有效的。

<u>重新解释对象</u>

通过类型转换它的地址，编译器可以将一个变量或对象当作另一个不同的类型来处理:

```
// Example 7.27
float x;
```

这里的语法可能看起来有点奇怪。将x的地址类型转换为指向整数的指针，然后对该指针取值，以便将x作为整数访问。编译器不会生成任何额外的代码来实际创建指针。指针被简单地优化掉了，结果是x被当作一个整数。但是,运算符强制编译器将x存储在内存中，而不是寄存器中。上面的示例使用|操作符设置x的符号位，只能应用于整数。这样操作比x = -abs(x)更快。

在类型转换指针时，有许多危险的地方需要注意：

1. 这个技巧违反了标准C的严格的别名规则，规定不同类型的两个指针不能指向相同的对象（char指针除外）。编译器优化可以将浮点数和整数表示形式存储在两个不同的寄存器中。你需要检查编译器是否按照您希望的方式运行。使用union会更安全，如146页（TODO）例14.23所示。
2. 如果将对象视为比实际更大的对象，这个技巧就会失效。如果int比float使用更多的位，上面的代码将会失败。（两者在x86系统中都使用32个位）。
3. 如果你只访问变量的一部分，例如64位双精度浮点数中的32位，那么代码将无法移植到使用大端存储的平台上。
4. 如果你以部分访问的方式访问变量，例如，如果您一次操作64位double类型的32位，那么由于CPU中的存储转发延迟，代码的执行速度可能会低于预期（参见手册3:“The microarchitecture of Intel, AMD and VIA CPUs”）。

<u>const_cast</u>

const_cast操作符用于解除const对指针的限制。它有一些语法检查，因此比C风格的类型转换更安全，er无需添加任何额外的代码。例如：

```
// Example 7.28
class c1 {
    const int x; // constant data
public:
    c1() : x(0) {}; // constructor initializes x to 0
    void xplus2() { // this function can modify x
        *const_cast<int*>(&x) += 2;} // add 2 to x
```

这里const_cast操作符的作用是消除x上的const限制，这是一种解除语法限制的方法，但它不会生成任何额外的代码，也不会花费任何额外的时间。这是确保一个函数可以修改x，而其他函数不能修改x的有用方法。

<u>static_cast</u>

static_cast操作符的作用与C风格的类型转换相同。例如，它用于将float转换为int。

<u>reinterpret_cast</u>

reinterpret_cast操作符用于指针转换。它的作用与C风格类型转换相同，只是多了一点语法检查。它不产生任何额外的代码。

<u>dynamic_cast</u>

dynamic_cast操作符用于将指向一个类的指针转换为指向另一个类的指针。它在运行时检查转换是否有效。例如，当基类的指针转换为派生类的指针时，它检查原始指针是否实际指向派生类的对象。这种检查使得dynamic_cast比简单类型的转换更耗时，但也更安全。它可能会捕获那些无法检测到的编程错误。

<u>转换类对象</u>

只有当程序员定义了构造函数、重载赋值运算符或重载类型转换运算符（指定如何进行转换）时，才有可能进行涉及类对象（而不是指向对象的指针）的转换。构造函数或重载操作符与成员函数的效率是一样的。

7.12 分支和switch语句

现代微处理器的高速运转是通过一个流水线（pipeline）来实现的，指令在执行之前，会在不同的几个阶段中被提取和解码。然而,流水线结构有一个大问题。当代码有分支时（例如if-else），微处理器事先不知道要给这两个分支中的哪个分支的数据送入流水线。如果将错误的分支送入管道，它通过读取、解码等方式所完成的工作，需要等到10 - 20个时钟周期以后才被检测到产生错误，在这段时间内投机性地执行指令是在浪费时间。结果是微处理器会将一个分支数据送入，只要稍后发现选择错了分支，那么就会浪费掉几个时钟周期。

微处理器设计者已经竭尽全力减少这个问题。其中最重要的方法是分支预测。现代微处理器使用先进的算法，根据该分支和附近其他分支的过去历史来预测分支的发展方向。对于不同类型的微处理器，用于分支预测的算法是不同的。这些算法在手册3"The microarchitecture of Intel, AMD and VIA CPUs"中有详细的描述。

在微处理器做出正确预测的情况下，分支指令通常需要0 - 2个时钟周期。根据处理器的不同，从分支错误预测中恢复所需的时间大约为12 - 25个时钟周期。这被称为分支错误预测的惩罚。

如果大多数时候分支是可预测的，那么它们消耗很少；但是如果预测错误，那么它们的消耗就大了。当然，总是沿着同一方向发展的分支是可预测的。一个分支在大多数情况下是单向的，很少是反向的，只有当它向另一个方向发展时，才会出现错误的预测。一个分支向一个方向走了很多次，然后又向另一个方向走了很多次，只有当它发生变化时才会被错误地预测。如果一个遵循简单周期模式的分支，在一个有很少或没有其它分支的循环中，那么也可以很好地进行预测。一个简单的周期模式可以是，例如，一条路走两遍，另一条路走三遍。同样的，两倍第一种方法，三倍另一种方法，等等。最坏的情况是一个分支随机地向一个方向或另一个方向移动，任意方向的概率都是50%。这样的分支有50%的几率会预测错误。

`for`循环或`while`循环也是一种分支。在每次迭代之后，它决定是重复还是退出循环。如果重复计数很小且始终相同，则通常可以很好地预测循环分支。根据处理器的不同，可以完美预测的最大循环数在9到64之间变化。嵌套循环只能在某些处理器上得到很好的预测。在许多处理器上，包含多个分支的循环并不能很好地被预测。

`switch`语句也是一种分支，它可以有两个以上的分支。如果`case`标签是遵循每个标签等于前一个标签加1的序列，那么`switch`语句是最高效的，因为它可以被实现为一个目标跳转表。如果switch语带有许多标签值，并且彼此相差很远，这将是低效的，因为编译器必须将其转换成一个分支树。

在较老的处理器上，会简单的认为带有顺序标签的`switch`语句的执行分支与上一次相同。因此，无论何时它走了不是上次走的分支，它肯定会被错误地预测。较新的处理器有时能够预测一个`switch`语句，如果它遵循一个简单的周期模式，或者它与前面的分支相关，并且不同目标的数量很少。

分支和`switch`语句的数量最好在程序的关键部分控制在较少的水平，特别是在分支的可预测性较差的情况下。那么展开循环可以消除分支，那这可能是有用的，这将在下一段中解释。

分支和函数调用的目标保存在称为分支目标缓冲区的特殊缓存中。如果一个程序有许多分支或函数调用，那么在分支目标缓冲区中就会产生竞争。这种竞争的结果是，即使分支有良好的可预测性，它们也可能被错误地预测。由于这个原因，甚至函数调用也可能被错误地预测。因此，在代码的关键部分具有许多分支和函数调用的程序可能会收到错误预测的影响。

在某些情况下，可以用表查找来替换难以预测的分支。例如：

```
// Example 7.29a
float a; bool b;
```

`?:`操作符在这里就是一个分支。如果它的可预测性很差，那么可以用一个表查找来代替它：

```
// Example 7.29b
float a; bool b = 0;
const float lookup[2] = {2.6f, 1.5f};
```

如果将`bool`变量用作数组索引，那么需要确保它被初始化或来自可靠的源，这非常重要，这样它除了0或1之外就不会有其他值。见*7.5 布尔值-布尔变量被过度检查*。

在某些情况下，编译器可以根据指定的指令集，自动使用条件转移替换分支。

第137页和139页（TODO）的例子展示了减少分支数量的各种方法。

手册3:"The microarchitecture of Intel, AMD and VIA CPUs"提供了不同微处理器中分支预测的更多细节。

7.13 循环

循环的效率取决于微处理器对循环制分支的预测能力。有关分支预测的说明，请参阅前文和手册3:"The microarchitecture of Intel, AMD and VIA CPUs"。一个具有一个较小并且固定的重复计数，没有分支的循环，可以完美地被预测。如上所述，可以预测的最大循环数取决于处理器。只有在某些具有特殊循环预测器的处理器上，嵌套循环才能很好地进行预测。在其他处理器上，只能很好地预测最里面的循环。只有在循环退出时，才会错误地预测具有高重复计数的循环。例如，如果一个循环重复1000次，那么循环控制分支在1000次中只会出现一次错误预测，因此错误预测惩罚对总执行时间的影响可以忽略不计。

<u>循环展开</u>

在某些情况下，展开循环可能很有益处。例如：

```
// Example 7.30a
int i;
for (i = 0; i < 20; i++) {
    if (i % 2 == 0) {
        FuncA(i);
    }
    else {
        FuncB(i);
    }
    FuncC(i);
```

这个循环重复20次，交替调用FuncA和FuncB，然后是FuncC。展开两个给出循环得到:

```
// Example 7.30b
int i;
for (i = 0; i < 20; i += 2) {
    FuncA(i);
    FuncC(i);
    FuncB(i+1);
    FuncC(i+1);
```

这么做有三个好处：

1. i<20循环控制分支执行10次而不是20次。
2. 重复计数已经从20减少到10，这意味着在奔腾4上可以完美地进行预测。
3. if分支被消除

展开循环同样也有缺点：

1. 展开循环在代码缓存或*micro-op*缓存中占用更多空间。
2. 非常小的循环（少于65字节的代码）在Core2处理器上执行得更好。
3. 如果重复计数为奇数，并将其展开为2，则必须在循环之外执行额外的迭代。通常，当重复计数不能被展开因子整除时，就会出现这种问题。

只有在能够取得特定好处的情况下，才应该使用循环展开。如果一个循环包含浮点运算，且循环计数器是整数，那么通常可以假设整个计算时间是由浮点代码决定的，而不是由循环控制分支决定的。在这种情况下，展开循环没有任何好处。

最好避免在有*micro-op*缓存的处理器上展开循环。因为节省*mircro-op*缓存的使用非常重要。

如果有利可图的话(见72页 TODO)， 编译器通常会自动展开一个循环。程序员不必手动展开循环，除非需要获得特定的优势，例如消除示例7.30b中的if分支。

<u>循环控制条件</u>

最高效的循环控制条件是一个简单的整数计数器。一个具有无序功能的微处理器（参见第105页 TODO）将能够在几个迭代之前评估循环控制语句。

如果循环控制分支依赖于循环内部的计算，则效率较低。下面的示例将以零结束的ASCII字符串转换为小写：

```
// Example 7.31a
char string[100], *p = string;
while (*p != 0)
```

如果字符串的长度是已知的，那么使用循环计数器会更有高效：

```
// Example 7.31b
char string[100], *p = string; int i, StringLength;
for (i = StringLength; i > 0; i--)
```

在数学迭代中，循环控制分支依赖于循环内部计算的是一种常见情况，如泰勒展开和牛顿-拉弗森迭代。在这里，需要重复迭代，直到残差小于一定的公差。计算残差绝对值并将其与公差进行比较所需的时间可能很长，因此确定最坏情况下的最大重复计数并始终使用此迭代次数会更高效。这种方法的优点是微处理器可以提前执行循环控制分支，并在循环内的浮点运算完成之前解决任何分支的错误预测。如果典型的重复计数接近最大重复计数，且每次迭代的残差计算对总计算时间有显著贡献，则该方法是有好处的。

循环计数器最好是整数。如果循环需要浮点计数器，那么创建一个额外的整数计数器。例如：

```
// Example 7.32a
double x, n, factorial = 1.0;
for (x = 2.0; x <= n; x++)
```

这可以通过添加一个整数计数器并在循环控制条件中使用整数来提升效率：

```
// Example 7.32b
double x, n, factorial = 1.0; int i;
for (i = (int)n - 2, x = 2.0; i >= 0; i--, x++)
```

注意带有多个计数器的循环中的逗号和分号之间的区别，如示例7.32b所示。for循环有三个子句：初始化、条件和增量。这三个子句用分号分隔，每个子句中的多个语句用逗号分隔。条件子句中应该只有一个语句。

将整数与零进行比较有时比将其与任何其他数字进行比较更高效。因此，将循环计数减少到0比将其增加到某个正值n要稍微快一些。但如果循环计数器用作数组索引，则不是这样。数据缓存是为向前而不是向后访问数组而优化的。

<u>复制或清除数组</u>

对于诸如复制数组或将数组设置为所有零这样的琐碎任务，使用循环可能不是最佳选择。例如：

```
// Example 7.33a
const int size = 1000; int i;
float a[size], b[size];
// set a to zero
for (i = 0; i < size; i++)
    a[i] = 0.0;
// copy a to b
for (i = 0; i < size; i++)
```

使用memset和memcpy函数通常会更快：

```
// Example 7.33b
const int size = 1000;
float a[size], b[size];
// set a to zero
memset(a, 0, sizeof(a));
// copy a to b
```

至少在简单的情况下，大多数编译器会自动使用memset和memcpy的替换这些循环。显式使用memset和memcpy是不安全的，因为如果参数 size 大于目标数组的大小，可能会发生严重错误。但是如果循环计数太大，同样的错误也会发生在循环中。

7.14 函数

函数调用可能会使程序慢下来，原因如下：

1. 函数调用使微处理器跳转到不同的代码地址，然后再返回。这可能需要4个时钟周期。在大多数情况下，微处理器能够将调用和返回操作与其他计算重叠以节省时间。
2. 如果代码分散在内存中，那么代码缓存的效率就会降低。
3. 在32位模式下，函数参数存储在堆栈中。将参数存储在堆栈上并再次读取它们需要额外的时间。如果参数是关键依赖链的一部分，则延迟是很明显的。
4. 需要额外的时间来设置栈帧（stack frame）、保存和恢复寄存器，可能还需要保存异常处理信息。
5. 每个函数调用语句需要在分支目标缓冲区（*BTB*）中占用空间。如果程序的关键部分有许多调用和分支，*BTB*中的竞争可能导致分支错误预测。

以下方法可用于减少在程序关键部分中，在函数调用上花费的时间。

<u>避免不必要的函数</u>

一些编程教科书建议，长度超过几行的每个函数都应该分成多个函数。我不同意这个规则。将一个函数分解成多个更小的函数只会降低程序的效率。仅仅因为一个函数很长就拆分它并不会使得程序更清晰，除非这个函数正在执行多个逻辑上不同的任务。如果可能的话，关键的最内层循环最好完全保留在一个函数中。

<u>使用内联函数</u>

内联函数会像宏一样展开，因此调用该函数的每个语句都会被函数体替换。如果使用了`inline`关键字，或者在类定义中定义了函数的主体，那么函数通常是内联的。如果函数很小，或者只在程序中的一个位置调用它，那么内联函数是有好处的。小函数通常由编译器自动内联。另一方面，在某些情况下，如果内联会导致技术问题或性能问题，编译器可能会忽略对函数内联的请求。

<u>避免在最内层循环中嵌套函数调用</u>

调用其他函数的函数称为帧函数（frame function），而不调用任何其他函数的函数称为叶函数（leaf function）。叶函数比框架函数更高效，原因见第63页（TODO）。如果程序关键的最内层循环包含对帧函数的调用，那么代码有可能通过内联帧函数或使帧函数调用的所有函数内联（把帧函数变为叶函数）来提升效率。

<u>使用宏代替函数</u>

用`#define`声明的宏肯定是内联的。但是要注意，宏的参数每次使用时都会被重新计算。例如：

```
// Example 7.34a. Use macro as inline function
#define MAX(a,b) (a > b ? a : b)
```

在这个例子中，`f(x)`或`g(x)`被计算两次，因为宏引用了两次。你可以通过使用内联函数而不是宏来避免这种情况。如果你想让函数可以使用任何类型的参数，那么可以使用模板：

```
// Example 7.34b. Replace macro by template
template <typename T>
static inline T max(T const & a, T const & b) {
    return a > b ? a : b;
```

宏的另一个问题是名称不能重载或限制作用区域。宏将干扰具有相同名称的任何函数或变量，而与作用域或命名空间无关。因此，对于宏来说，使用足够长且唯一的名称非常重要，尤其是在头文件中。

<u>使用<code>fastcall</code>函数</u>

在32位模式下，关键字`__fastcall`将会改变函数的调用方式，使用寄存器而不是栈来传递前两个整型参数（*CodeGear*编译器则是前三个）。这可以提升拥有整型参数的函数的速度。

浮点型参数则不会被`__fastcall`影响。成员函数中隐藏的`'this'`指针也被是为一个参数，所以可能只剩下一个空闲寄存器用于传输其他参数。因此，确保在使用`__fastcall`时，最关键的整数参数放在第一位。64位模式下的函数参数默认是使用寄存器传递的。因此，64位模式下无法识别`__fastcalll`关键字。

<u>使函数局部化</u>

在同一个模块中使用的函数（即当前.cpp文件）应该是本地的。这使得编译器使函数更容易美联，并对函数调用进行优化。有三种方法使一个函数局部化:

1. 将关键字`static`添加到函数声明中。这是最简单的方法，但它不适用于类成员函数，在类成员函数中，`static`有不同的含义。
2. 函数或类放入匿名命名空间（译者注：定义命名空间使，忽略命名空间的名称）。
3. *Gnu*编译器允许使用"**attribute**((visibility("hidden")))"

<u>使用全程序优化</u>

一些编译器可以选择对整个程序进行优化，也可以选择将多个.cpp文件组合成一个对象文件。这使得编译器能够在组成程序的所有.cpp模块之间优化寄存器分配和参数传递。对于作为目标文件或库文件分发的函数库，不能使用全程序优化。

<u>使用64位模式</u>

在64位模式下，参数传递比在32位模式下更高效，而64位*Linux*比64位*Windows*更快。在64位*Linux*中，前6个整数参数和前8个浮点参数使用寄存器传递，总计14个寄存器参数。而在64位*Windows*中，前四个参数在寄存器中传递，而不管它们是整数还是浮点数。因此，如果函数有四个以上的参数，64位*Linux*比64位*Windows*更快。32位*Linux*和32位*Windows*在这个层面上没有差别。

7.15 函数参数

在大多数情况下，函数参数是按值传递的。这意味着参数的值被复制到一个局部变量。对于`int`、`float`、`double`、`bool`、`enum`以及指针和引用等简单类型，这非常快。

数组总是使用指针传递，除非它们被打包在类或者结构体中。

如果参数是复合类型，例如结构体或类，那么情况会更复杂一些。复合类型的参数传递在符合一下几个条件的情况下是最高效的：

1. 这个对象很小，可以装入一个寄存器中。
2. 这个对象没有拷贝构造函数和析构函数。
3. 这个对象没有虚成员。
4. 这个对象没有使用运行时类型标识（*RTTI*）。

如果这些条件中，有任何一个不满足，那么使用指针或引用传递对象通常会更快。如果对象很大，那么显而易见，复制整个对象需要时间。当对象复制到参数时，必须调用复制构造函数，如果有析构函数的话，必须在函数返回之前调用析构函数。

将组合对象传递给函数的首选方法是使用`const`引用。`const`引用确保原始对象没有被修改。与指针或非`const`引用不同，`const`引用允许函数参数为表达式或匿名对象。如果函数是内联的，编译器可以很容易地优化掉const引用。

另一种解决方案是使函数成为对象的类或结构的成员，这同样有用。

在32位系统中，简单的函数参数在栈上传递，但在64位系统，使用寄存器中传递。后者效率更高。64位*Windows*允许在寄存器中传输最多4个参数。64位*Unix*系统允许在寄存器中传输最多14个参数（8个浮点数或双精度数加上6个整数、指针或引用参数）。成员函数中的`this`指针占用一个参数。手册5:"Calling conventions for different C++ compilers and operating systems"给出了更多的细节。

7.16 函数返回类型

函数的返回类型最好是简单类型、指针、引用或void。返回复合类型的对象更为复杂，而且常常效率低下。

复合类型的对象只能在最简单的情况下在寄存器中返回。有关何时可以在寄存器中返回对象的详细信息，请参见手册5:"Calling conventions for different C++ compilers and operating systems"。

除了最简单的情况外，复合对象的返回方式是通过一个隐藏指针将它们复制到调用方指定的位置。复制构造函数（如果有的话）通常在复制过程中被调用，而析构函数则在销毁原始构造函数时被调用。在简单的情况下，编译器可以通过在对象的最终目的地构造对象来避免对复制构造函数和析构函数的调用，但是不要依赖这一点。

你可以考虑以下替代方法，而不是返回复合对象：

1. 使函数成为对象的构造函数。
2. 让函数修改一个现有的对象，而不是创建一个新的对象。现有对象可以通过指针或引用提供给函数，或者函数可以是对象的类的成员。
3. 使函数返回一个指向函数内部定义的静态对象的指针或引用。这是有效的，但也有风险。返回的指针或引用仅在下一次调用函数并覆盖本地对象(可能在不同的线程中)之前有效。如果忘记将局部对象定义为静态的，那么一旦函数返回，它就会失效。
4. 使用`new`在函数中构造一个对象，并返回一个指向它的指针。由于动态内存分配的成本，这是低效的。如果忘记删除对象，此方法还涉及内存泄漏的风险。

7.17 函数尾调用

尾调用是优化函数调用的一种方法。如果函数的最后一条语句是对另一个函数的调用，那么编译器可以用跳转到第二个函数来替换该调用。编译器优化将自动完成此任务。第二个函数不会返回到第一个函数，而是直接返回第一个函数被调用的位置。这更有效效率，因为它消除了返回操作。例如：

```
// Example 7.35. Tail call
void function2(int x);
void function1(int y) {
    ...
    function2(y+1);
}
```

在这里，通过直接跳到`function2`来消除`function1`的返回。即使有返回值，也可以这样做：

```
// Example 7.36. Tail call with return value
int function2(int x);
int function1(int y) {
...
return function2(y+1);
```

尾调用优化只有在两个函数具有相同的返回类型时才有效。如果函数在栈上有参数（在32位模式下通常是这样），那么这两个函数必须为参数使用相同数量的栈空间。

7.18 递归函数

递归函数是一个调用自身的函数。函数递归调用对于处理递归数据结构非常有用。递归函数的代价是所有参数和局部变量在每次递归时都会有一个新实例，这将占用栈空间。深度递归还会降低返回地址的预测效率。这个问题通常出现在递归深度超过16的情况下（参见手册3"The microarchitecture of Intel, AMD and VIA CPUs"中对返回栈缓冲区的解释）。

递归函数调用仍然是处理分支数据树结构的最有效解决方案。较宽的树形结构比较深的树形结构，有更高的递归效率。无分支递归总是可以被循环替代，这样效率更高。递归函数的一个常见教科书例子是阶乘函数：

```
// Example 7.37. Factorial as recursive function
unsigned long int factorial(unsigned int n) {
if (n < 2) return 1;
    return n * factorial(n-1);
```

这种实现非常低效，因为`n`的所有实例和所有返回地址都会占用堆栈上的存储空间。使用循环更高效：

```
// Example 7.38. Factorial function as loop
unsigned long int factorial(unsigned int n) {
    unsigned long int product = 1;
    while (n > 1) {
        product *= n;
        n--;
        }
    return product;
```

递归尾调用（尾递归）比其他递归调用更高效，但仍然不如循环快。

初学者有时会调用`main`函数来重启程序。这不是一个好主意，因为每次递归调用`main`函数时，栈都会被所有本地变量的新实例填满。重新启动程序的正确方法是在`main`函数中使用循环。

7.19 结构体和类

现在，编程教科书推荐面向对象编程作为一种使软件更加清晰和模块化的方法。所谓的对象是结构和类的实例。面向对象的编程风格对程序性能既有积极的影响，也有消极的影响。积极的影响是：

1. 如果一起使用的变量是相同结构或类的成员，那么它们会被存储在一起。这使得数据缓存更有效率。
2. 不需要将类成员的变量作为参数传递给类成员函数。这些变量避免了参数传递的开销。

面向对象编程的负面影响有：

1. 非静态成员函数有一个 `this` 指针，该指针作为隐形参数传递给函数。`this` 的参数传输开销会在所有非静态成员函数上产生。
2. `this`指针占用一个寄存器。在32位系统中，寄存器是一种稀缺资源。
3. 虚成员函数的效率较低（参见第55页,TODO）。

关于面向对象编程的正面影响还是负面影响占主导地位，还没有一个通用的说法。至少，可以这样说，使用类和成员函数的代价并不大。如果面向对象的编程风格有利于程序的逻辑结构和清晰性，那么你可以使用这种风格，只要你避免在程序最关键的部分调用过多的函数。结构体的使用（没有成员函数的）对性能没有负面影响。

7.20 类的数据成员（变量实例）

类或结构体的数据成员是按创建类或结构实例时声明它们的顺序连续存储。将数据组织到类或结构中不存在性能损失。访问类或结构对象的数据成员所花费的时间不比访问简单变量多。

大多数编译器将数据成员对齐到整数地址以优化访问，如下表所示：

Type	size,bytes	alignments, bytes
`bool`	1	1
`char`, `signed` or `unsigned`	1	1
`short int`, `signed` or `unsigned`	2	2
`int`, `signed` or `unsigned`	4	4
64-bit integer, `signed` or `unsigned`	8	8
pointer or reference, 32-bit mode	4	4
pointer or reference, 64-bit mode	8	8
`float`	4	4
`double`	8	8
`long double`	8, 10, 12 or 16	8 or 16

Table 7.2. Alignment of data members

这种对齐会导致结构或类中的未使用的字节空洞，这些字节的成员大小不一。例如：

```
// Example 7.39a
struct S1 {
short int a;         // 2 bytes. first byte at 0, last byte at 1
                     // 6 unused bytes
double b;            // 8 bytes. first byte at 8, last byte at 15
int d;               // 4 bytes. first byte at 16, last byte at 19
                     // 4 unused bytes
};
```

这里，`a`和`b`之间有6个未使用的字节，因为`b`必须从一个能被8整除的地址开始。最后还有4个未使用的字节。这样做的原因是，数组中`S1`的下一个实例必须从一个可被8整除的地址开始，以便将其`b`成员与8对齐。通过将最小的成员放在最后，可以将未使用的字节数减少到2：

```
// Example 7.39b
struct S1 {
double b;     // 8 bytes. first byte at 0, last byte at 7
```

```
int d;        // 4 bytes. first byte at 8, last byte at 11
short int a;  // 2 bytes. first byte at 12, last byte at 13
              // 2 unused bytes
};
```

这种重新排序使结构减少了8字节，数组占用的空间减少了800字节。

通过对数据成员的重新排序，结构体和类对象通常可以变得更小。如果类至少有一个虚成员函数，则在第一个数据成员之前或最后一个成员之后有一个指向虚拟表的指针。这个指针在32位系统中是4字节，在64位系统中是8字节。如果你对一个结构体或其每个成员的大小有疑问，那么你可以使用sizeof操作符进行一些测试。sizeof操作符返回的值包括对象末尾未使用的字节。

如果成员相对于结构体或类的开头的偏移量小于128，则访问数据成员的代码会更紧凑，因为偏移量可以表示为8位有符号数字。如果相对于结构或类的开头的偏移量是128字节或更多，那么偏移量必须表示为32位数字（在8位到32位偏移量之间，指令集没有其它可选择的偏移量）。例如：

```
// Example 7.40
class S2 {
public:
int a[100]; // 400 bytes. first byte at 0, last byte at 399
int b; // 4 bytes. first byte at 400, last byte at 403
int ReadB() {return b;}
```

b的偏移量是400。任何通过指针或成员函数（如ReadB）访问b的代码都需要将偏移量编码为32位数字。如果交换了a和b，那么可以使用一个被编码为8位有符号数字的偏移量来访问它们，或者完全不使用偏移量。这使得代码更紧凑，从而更有效地使用代码缓存。因此，建议在结构或类声明中，大数组和其他大对象放在最后，最常用的数据成员放在前面。如果不可能在前128个字节中包含所有数据成员，则将最常用的成员放在前128个字节中。

7.21类的成员函数（方法）

每次声明或创建类的新对象时，它都会生成数据成员的新实例。但是每个成员函数只有一个实例。函数代码不会被复制，因为相同的代码可以应用于类的所有实例。

调用成员函数与调用简单函数使用结构体（类）的指针或引用一样快。例如：

```
// Example 7.41
class S3 {
public:
int a;
int b;
int Sum1() {return a + b;}
};
int Sum2(S3 * p) {return p->a + p->b;}
```

Sum1, Sum2和Sum3这三个函数做的是完全一样的事情，它们的效率是一样的。如果查看编译器生成的代码，你会注意到一些编译器将为这三个函数生成完全相同的代码。Sum1有一个隐式的 this 指针，它在Sum2和Sum3中p和r的作用相同。无论你是想让函数成为类的成员，还是给它一个指向类或结构的指针或引用，都只是编程风格的问题。一些编译器通过使用寄存器中而不是栈传输 this，使Sum1在32位*Windows*中比Sum2和Sum3稍微高效一些。

静态成员函数不能访问任何非静态数据成员或非静态成员函数。静态成员函数比非静态成员函数快，因为它不需要 this 指针。如果成员函数不需要访问任何非静态的东西，可以通过使它们变为静态来变得更快。

7.22 虚成员函数

虚函数用于实现多态类。一个多态类的每个实例都有一个指针指向一个指针表（虚函数表），其中的指针指向虚函数的不同版本。这个所谓的虚函数表用于在运行时查找虚函数的正确版本。多态性是面向对象程序比非面向对象程序效率低的主要原因之一。如果可以避免使用虚函数，那么你就可以获得面向对象编程的大多数优势，而无需付出性能成本。

如果函数调用语句总是调用虚函数的相同版本，那么调用虚成员函数所花费的时间要比调用非虚成员函数多几个时钟周期。如果版本发生了变化，你可能会得到10 - 20个时钟周期的错误预测惩罚。虚函数调用的预测和错误预测规则与switch语句相同，如第44页所述（TODO）；

在对已知类型的对象调用虚函数时，可以绕过分发机制，但是不能总是依赖编译器绕过分发机制，即使很明显可以这样做。见75页（TODO）。

只有在编译时无法知道调用了多态成员函数的哪个版本时，才会需要运行时多态。如果需要在程序的关键部分中使用虚函数，那么你可以考虑是否可以在不使用多态性或使用编译时多态性的情况下完成所需的功能。

有时可以使用模板而不是虚函数来获得所需的多态性效果。模板参数应该是包含具有多个版本的函数的类。这个方法更快，因为模板参数总是在编译时解析，而不是在运行时解析。第59页（TODO）的示例7.47展示了如何做到这一点。不幸的是，它的语法非常笨拙，可能不值得花这么多功夫。

7.23 运行时类型识别（RTTI）

运行时类型识别会向所有类对象添加额外的信息，而且效率不高。如果编译器有*RTTI*选项，那么关闭它并使用其他实现。

7.24 继承

派生类的对象与包含父类和子类成员的简单类的对象的实现方法相同。父类和子类的成员访问速度相同。一般来说，您可以假设使用继承几乎没有任何性能损失。

由于如下原因代码缓存的性能可能会有轻微的下降：

1. 父类数据成员的大小被添加到子类成员的偏移量中。访问总偏移量大于127字节的数据成员的代码稍微不那么紧凑。参见54页（TODO）。
2. 父类和子类的成员函数通常存储在不同的模块中。这可能会导致大量的跳转和低效的代码缓存。这个问题可以通过确保相互调用的函数存储在彼此附近来解决。详情见第90页（TODO）。

同一个类从多个父类继承会导致成员指针和虚函数，或者通过指向基类之一的指针访问派生类对象的复杂性太高。你可以通过在派生类中创建对象来避免多重继承：

```
// Example 7.42a. Multiple inheritance
class B1; class B2;
class D : public B1, public B2 {
public:
    int c;
```

替换为：

```
// Example 7.42b. Alternative to multiple inheritance
class B1; class B2;
class D : public B1 {
public:
    B2 b2;
    int c;
```

7.25 构造函数和析构函数

构造函数在内部被实现为一个成员函数，该成员函数返回对对象的引用。新对象的内存分配不一定由构造函数本身完成。因此构造函数和其他成员函数效率一样。这适用于默认构造函数、复制构造函数和任何其他构造函数。

类不需要构造函数。如果对象不需要初始化，则不需要默认构造函数。如果仅通过复制所有数据成员就可以复制对象，则不需要复制构造函数。可以将简单的构造函数定义为内联的来提高性能。

无论何时通过赋值复制对象、作为函数参数或作为函数返回值，都可以调用复制构造函数。如果复制构造函数涉及内存或其他资源的分配，则它可以相当耗时。有很多方法可以避免这种浪费的内存块的复制，例如：

1. 使用对象的引用或指针，而不是复制它。
2. 使用"移动构造函数"（move constructor）来转移内存块的所有权。这需要一个支持*C++ 0x*的编译器。
3. 创建一个成员函数或友元函数或操作符，将内存块的所有权从一个对象转移到另一个对象。失去内存块所有权的对象应该将其指针设置为NULL。当然，应该有一个析构函数来销毁对象所拥有的任何内存块。
4. 析构函数和成员函数效率一样。如果没有必要，不要创建析构函数。虚析构函数和虚成员函数效率一样。见55页(TODO)。

7.26 联合体

union是数据成员共享相同内存空间的结构。union可以通过允许从不同时使用的两个数据成员共享同一块内存来节省内存空间。参见第91页的示例（TODO）。

union还可以用于以不同的方式访问相同的数据。例如：

```
// Example 7.43
union {
    float f;
    int i;
} x;
x.f = 2.0f;
x.i |= 0x80000000; // set sign bit of f
```

在本例中，f的符号位是通过使用位或（|）运算符设置的，该运算符只能应用于整数。

7.27 位域

位域可能有助于使数据更加紧凑。访问位域成员不如访问结构的成员效率高。如果在大数组可以节省缓存空间或使文件更小，那么额外的时间是合理的。

使用<<和|组合操作来操作位域比单独操作成员要快。例如：

```
// Example 7.44a
struct Bitfield {
    int a:4;
    int b:2;
    int c:2;
};
Bitfield x;
int A, B, C;
x.a = A;
x.b = B;
```

假设A、B、C的值很小，不会导致溢出，可以通过以下方式对该代码进行改善：

```
// Example 7.44b
union Bitfield {
struct {
    int a:4;
    int b:2;
    int c:2;
};
char abc;
};
Bitfield x;
int A, B, C;
```

或者，如果需要防止溢出：

```
// Example 7.44c
```

7.28 重载函数

重载函数的不同版本被简单地视为不同的函数。使用重载函数没有性能损失。

7.29 重载操作符

重载的运算符相当于一个函数。使用重载操作符与使用具有相同功能的函数效率一样。

表达式具有多个重载操作符，将导致为中间结果创建临时对象，这可能是我们不希望看到的。例如：

```
// Example 7.45a
class vector { // 2-dimensional vector
public:
    float x, y; // x,y coordinates
    vector() {} // default constructor
    vector(float a, float b)
    {
        x = a;
        y = b;
    } // constructor
    vector operator + (vector const & a)
    { // sum operator
        return vector(x + a.x, y + a.y);
    } // add elements
};
vector a, b, c, d;
```

为中间结果（`b+c`）创建临时对象可以通过加入以下操作来避免：

```
// Example 7.45b
a.x = b.x + c.x + d.x;
```

幸运的是，大多数编译器会在简单的情况下自动进行优化。

7.30 模板

模板与宏的相似之处在于，模板参数在编译之前被它们的值所替换。下面的例子说明了函数参数和模板参数之间的区别：

```
// Example 7.46
int Multiply (int x, int m) {
    return x * m;}

template <int m>
int MultiplyBy (int x) {
    return x * m;}

int a, b;
a = Multiply(10,8);
```

*a*和*b*都得到$10 * 8 = 80$。区别在于`m`传递到函数的方式。在这个简单的函数中，`m`在运行时从调用者转移到被调用的函数。但是在模板函数中，`m`在编译时被它的值所代替，这样编译器看到的是常量8而不是变量`m`。使用模板参数而不是函数参数的优点是避免了参数传递的开销。缺点是编译器需要为每个不同的值创建模板函数的新实例。如果在本例中使用许多不同的系数作为模板参数来调用`MultiplyBy`，那么代码可能会变得非常大。

在上面的例子中，模板函数比简单函数快，因为编译器知道它可以通过以为操作来实现乘以2的幂。`x*8`被`x<<3`所代替，速度更快。在简单函数的情况下，编译器不知道`m`的值，因此不能进行优化，除非函数可以内联。（在上面的例子中，编译器能够内联和优化这两个函数，并简单地将`80`存入`a`和`b`中。但在更复杂的情况下，编译器可能无法做到这一点）。

模板参数也可以是类型。第38页（TODO）的示例展示了如何使用相同的模板创建不同类型的数组。

模板是高效的，因为模板参数总是在编译时被解析。模板使源代码更加复杂，而不是编译后的代码。一般来说，使用模板在执行速度方面没有任何成本。

如果模板参数完全相同，则将两个或多个模板实例合并为一个。如果模板参数不同，那么每一组模板参数都将获得一个实例。有许多实例的模板会使编译后的代码变大，并使用更多的缓存空间。

过度使用模板会使代码难以阅读。如果模板只有一个实例，那么你也可以使用`#define`、`const`或`typedef`来代替模板参数。

模板可以用于元编程，如第154页所述(TODO)；

<u>使用模板实现多态</u>

模板类可用于实现编译时多态性，这比使用虚拟成员函数获得的运行时多态性更加高效。下面的示例首先展示了运行时多态性：

```
// Example 7.47a. Runtime polymorphism with virtual functions
class CHello {
public:
    void NotPolymorphic(); // Non-polymorphic functions go here
    virtual void Disp(); // Virtual function
    void Hello() {
        cout << "Hello ";
        Disp(); // Call to virtual function
    }
};
class C1 : public CHello {
public:
    virtual void Disp() {
        cout << 1;
    }
};
class C2 : public CHello {
public:
    virtual void Disp() {
        cout << 2;
    }
};
void test () {
    C1 Object1; C2 Object2;
    CHello * p;
    p = &Object1;
    p->NotPolymorphic(); // Called directly
    p->Hello(); // Writes "Hello 1"
    p = &Object2;
    p->Hello(); // Writes "Hello 2"
```

如果编译器不知道对象`p`指向什么类（参见第75页，TODO），则会在运行时分发到`C1::Disp()`或`C2::Disp()`。当前的编译器不太擅长优化掉`p`，并内联`Object1.Hello()`的调用，不过将来的编译器可能能够做到这一点。

如果在编译时知道对象是属于类`C1`还是`C2`，那么我们就可以避免低效的虚函数分发过程。这可以通过在活动模板库（*ATL*）和Windows模板库（*WTL*）中使用的特殊技巧来实现：

```
// Example 7.47b. Compile-time polymorphism with templates
// Place non-polymorphic functions in the grandparent class:
class CGrandParent {
public:
    void NotPolymorphic();
};
// Any function that needs to call a polymorphic function goes in the
// parent class. The child class is given as a template parameter:
template <typename MyChild>
```

```
class CParent : public CGrandParent {
public:
void Hello() {
    cout << "Hello ";
    // call polymorphic child function:
    (static_cast<MyChild*>(this))->Disp();
    }
};
// The child classes implement the functions that have multiple
// versions:
class CChild1 : public CParent<CChild1> {
public:
    void Disp() {
        cout << 1;
    }
};
class CChild2 : public CParent<CChild2> {
public:
    void Disp() {
        cout << 2;
    }
};
void test () {
    CChild1 Object1; CChild2 Object2;
    CChild1 * p1;
    p1 = &Object1;
    p1->Hello(); // Writes "Hello 1"
    CChild2 * p2;
    p2 = &Object2;
    p2->Hello(); // Writes "Hello 2"
```

在这里CParent是一个模板类，它通过模板参数获取关于其子类的信息。它可以通过将它的 this 指针类型转换为指向它的子类的指针来调用它的子类的多态成员。只有将正确的子类名作为模板参数时，这才是安全的。换句话说，您必须确保子类的声明 class CChild1 : public CParent<CChild1> {和模板参数具有相同的名称。

现在继承的顺序如下。第一代类（CGrandParent）包含任何非多态成员函数。第二代类（CParent<>）包含任何需要调用多态函数的成员函数。第三代类包含多态函数的不同版本。第二代类可以通过模板参数获取关于第三代类的信息。the user might experience unacceptably long response times to keyboard and mouse inputs when the program is busy doing the spell checking.

如果对象的类名是已知的，那么在将运行时分派虚成员函数时不会浪费时间。这些信息包含在具有不同类型的p1和p2中。缺点是CParent::Hello()有多个实例占用缓存空间。

例7.47b中的语法显然是非常笨拙的。通过避免虚函数分发机制，我们节省出来的几个时钟周期，难以证明如此复杂的难以理解的，因此也难以维护的代码是合适的。如果编译器能够自动执行反虚化（参见第75页，TODO），那么依赖编译器优化肯定比使用这种复杂的模板方法更加方便。

7.31 线程

线程用于同时或看起来是同时地执行两个或多个作业。如果计算机只有一个*CPU*核心，那么不可能同时执行两个任务。对于前台任务，每个线程将获得通常为30 ms的时间片，对于后台任务，每个线程将获得10 ms的时间片。每个时间片之后的上下文切换非常耗时，因为所有缓存都必须适应新的上下文。可以通过设置更长的时间片来减少上下文切换的次数。这将使应用程序运行得更快，但用户输入的响应时间会更长。(在*Windows*中，你可以通过在高级系统性能选项下为后台服务选择优化性能，将时间片增加到120ms。我不知道这在*Linux*中是否可行)。

为不同的任务的不同线程分配不同的优先级是非常有用的。例如，在字处理软件中，用户希望按下一个按键或移动鼠标时能够立即得到响应，这项任务必须有很高的优先级。而其他任务，例如拼写检查和重新分页，在其他优先级较低的线程中运行。如果不同的任务没有被划分成具有不同优先级的线程，那么当程序忙于拼写检查时，可能需要花很长时间来响应键盘和鼠标输入，这是用户所不希望遇到的。

如果应用程序有图形用户界面，那么任何需要很长时间的任务，比如繁重的数学计算，都应该安排在单独的线程中。否则程序将无法快速响应键盘或鼠标输入。

在应用程序中执行类似线程的调度而不调用操作系统线程调度程序，以节省开销是可能的。这可以通过在图形用户界面（在*Windows MFC*中为OnIdle）的消息循环中调用的函数中逐块地进行大量的后台计算来实现。这种方法可能比在只有一个*CPU*内核的系统中创建单独的线程要

快，但是它要求后台作业可以被分割成合适持续时间的多个小块。

充分利用具有多个*CPU*内核的系统的最佳方法是将工作划分为多个线程。然后每个线程可以在自己的*CPU*内核上运行。

在优化多线程应用程序时，我们必须考虑多线程的四种成本：

1. 启动和停止线程的成本。如果与启动和停止线程所需的时间相比，任务的持续时间较短，则不要将其放入单独的线程中。
2. 任务切换的成本。如果具有相同优先级的线程数量不超过*CPU*内核的数量，则此成本达到最小值。
3. 线程间同步和通信的成本。信号量、互斥量等的开销相当大。如果两个线程经常为了访问同一资源而相互等待，那么最好将它们合并到一个线程中。多个线程之间共享的变量必须声明为`volatile`。这将阻止编译器对该变量进行优化。
4. 不同的线程需要单独的存储空间。多线程使用的函数或类都不应该依赖于静态变量或全局变量。(参见线程本地存储p.28，TODO）。线程有各自的堆栈。如果线程共享相同的缓存，这可能会导致缓存竞争。

多线程程序必须使用线程安全的函数。线程安全的函数永远不应该使用静态变量。

有关多线程技术的进一步讨论，请参见第10章第103页（TODO）。

7.32 异常和错误处理

运行时错误会导致异常，这些异常可以通过陷阱（traps）或软件中断的形式检测到。可以使用`try-catch`块捕捉这些异常。如果启用异常处理且没有try-catch块，则程序将崩溃，并显示错误消息。

异常处理旨在检测很少发生的错误，并以一种优雅的方式从错误条件中恢复。你可能认为只要没有发生错误，异常处理就不需要额外的时间，但不幸的是，这并不总是正确的。为了知道如何在异常事件中恢复，程序可能需要做大量的记录工作。这种记录的消耗在很大程度上取决于编译器。有些编译器具有高效的基于表的方法，开销很少或没有，而其他编译器则具有低效的基于代码的方法，或者需要运行时类型标识（*RTTI*），这会影响代码的其他部分。更详细的信息请参阅 ISO/IEC TR18015 Technical Report on C++ Performance 。

下面的例子解释了为什么需要记录工作：

```
// Example 7.48
class C1 {
public:
    ...
    ~C1();
};

void F1() {
    C1 x;
    ...
}
void F0() {
    try {
        F1();
    }
    catch (...) {
        ...
    }
```

函数`F1`在返回时应该调用对象`x`的析构函数。但是如果`F1`中的某个地方发生异常怎么办？然后我们跳出`F1`而不返回。`F1`的清理工作被阻止了，因为它被中断了。现在，异常处理程序负责调用`x`的析构函数，这只有在`F1`保存了要调用的析构函数的所有信息或可能需要的任何其他清理信息时才有可能。如果`F1`调用另一个函数进而调用另一个函数，等等，如果在最里面的函数产生了一个异常，然后异常处理程序需要关于函数调用链和需要遵循的函数调用的顺序等所有信息，来检查所有必要的清理工作。这叫做堆栈展开。

即使没有异常发生，所有函数仍必须为异常处理程序保存一些信息。这就是异常处理在某些编译器中代价高昂的原因。如果你的应用程序不需要异常处理，那么应该禁用它，以便使代码更小、更高效。你可以通过关闭编译器中的异常处理选项来禁用整个程序的异常处理。你也可以通过向函数原型中添加`throw()`来禁用单个函数的异常处理：

```
void F1() throw();
```

这允许编译器假设`F1`永远不会抛出任何异常，这样它就不必为函数`F1`保存恢复信息。但是，如果`F1`调用另一个可能抛出异常的函数`F2`，那么`F1`必须检查`F2`抛出的异常，并在`F2`实际抛出异常时调用`std::unexpected()`函数。因此，只有当F1调用的所有函数也有一个`throw()`声明时才可以对`F1`使用`throw()`声明。`throw()`声明对于库函数很有用。

编译器会区分叶函数和帧函数。帧函数是至少调用一个其他函数的函数。叶函数是一个不调用任何其他函数的函数。叶函数比帧函数简单，因为如果可以排除异常，或者在发生异常时没有什么需要清理的情况下，堆栈展开信息可以被忽略。帧函数可以通过内联它调用的所有函数来转换为叶函数。如果程序最内层的关键循环不包含对帧函数的调用，则可以得到最佳性能。

虽然`throw()`语句在某些情况下可以提升、优化程序性能，但是没有理由添加诸如`throw(A,B,C)`这样的语句来显式地告诉函数可以抛出什么样的异常。实际上，编译器可能会添加额外的代码来检查抛出的异常是否属于指定的类型（参见Sutter的文章：A Pragmatic Look at Exception Specifications, Dr Dobbs Journal, 2002）。

在某些情况下，即使在程序最关键的部分使用异常处理也是最优的。如果替代实现的效率较低，并且您希望能够从错误中恢复，那么就会出现这种情况。下面的示例演示了这种情况：

```
// Example 7.49
// Portability note: This example is specific to Microsoft compilers.
// It will look different in other compilers.
#include <excpt.h>
#include <float.h>
#include <math.h>
#define EXCEPTION_FLT_OVERFLOW 0xC0000091L

void MathLoop()
{
    const int arraysize = 1000; unsigned int dummy;
    double a[arraysize], b[arraysize], c[arraysize];
    // Enable exception for floating point overflow:
    _controlfp_s(&dummy, 0, _EM_OVERFLOW);
    //_controlfp(0, _EM_OVERFLOW); // if above line doesn't work
    int i = 0; // Initialize loop counter outside both loops
    // The purpose of the while loop is to resume after exceptions:
    while (i < arraysize)
    {
        // Catch exceptions in this block:
        __try
        {
            // Main loop for calculations:
            for ( ; i < arraysize; i++)
            {
                // Overflow may occur in multiplication here:
                a[i] = log (b[i] * c[i]);
            }
        }
        // Catch floating point overflow but no other exceptions:
        __except (GetExceptionCode() == EXCEPTION_FLT_OVERFLOW
        ? EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH)
        {
            // Floating point overflow has occurred.
            // Reset floating point status:
            _fpreset();
            _controlfp_s(&dummy, 0, _EM_OVERFLOW);
            // _controlfp(0, _EM_OVERFLOW); // if above doesn't work
            // Re-do the calculation in a way that avoids overflow:
            a[i] = log(b[i]) + log(c[i]);
            // Increment loop counter and go back into the for-loop:
            i++;
        }
    }
```

假设`b[i]`和`c[i]`中的数字非常大，以至于在乘法`b[i]*c[i]`中可以发生溢出，尽管这种情况很少发生。上面的代码将捕获溢出时的异常，并以一种花费更多时间但避免溢出的方式重新执行计算。对每个因子取对数，而不是对乘积取对数，可以确保不会发生溢出，但是计算时间增加了一倍。

支持异常处理所需的时间可以忽略不计，因为在关键的最内层循环中没有try块或函数调用（日志除外）。log是一个库函数，我们假设它是经过优化的。无论如何，我们都不能更改其可能的异常处理支持。异常发生时代价很高，但这不是问题，因为我们假设这种情况很少发生。

在这里，测试循环内部的溢出条件不需要任何成本，因为我们依赖微处理器硬件在发生溢出时引发异常。异常被操作系统捕获，如果有try块，操作系统会将其重定向到程序中的异常处理程序。

捕获硬件异常存在可移植性问题。这种机制依赖于编译器、操作系统和CPU硬件中的非标准化细节。将这样的应用程序移植到不同的平台可能需要修改代码。

让我们在这个例子中看看异常处理的可能替代方法。在相乘之前，我们可以检查b[i]和c[i]是否太大，从而检查溢出。这将需要两个浮点数比较，这是比较耗时的，因为它们必须在最内层循环中。另一种可能是始终使用安全的公式a[i] = log(b[i]) + log(c[i])，这将使log的调用次数增加一倍，而对数需要很长时间来计算。如果有一种方法可以在不检查所有数组元素的情况下检查循环之外的溢出，那么这可能是一种更好的解决方案。如果所有因子都是由相同的几个参数生成的，那么在循环之前进行这样的检查是可能的。或者，如果结果由某些公式组合成单个结果，那么可以在循环之后进行检查。

<u>异常和向量代码</u>

向量指令对于并行执行多个计算是有用的。下文第12章对此进行了描述。异常处理不适用于向量代码，因为向量中的单个元素可能会导致异常，而其他向量元素可能不会。由于分支在向量代码中实现的方式，你甚至可以在未采用的分支中得到异常。如果代码可以从向量指令中获益，那么最好禁用异常捕获，转而依赖NAN和INF的传递。见下文第7.34章。关于这一点进一步讨论参见www.agner.org/optimize/nan_propagation.pdf。

<u>避免异常处理的成本</u>

当不需要尝试从错误中恢复时，不需要异常处理。如果你只是希望程序发出错误消息并在出现错误时停止程序，那么就没有理由使用try、catch和throw。更好的方法是定义自己的错误处理函数，该函数只打印适当的错误消息，然后调用exit。

如果有已分配的资源需要被清理的话， 调用exit可能并不安全，解释如下。还有其他不使用异常处理错误的可能方法。检测错误的函数可以返回一个错误代码，调用函数可以使用该代码进行恢复或发出错误消息。

建议使用系统的、经过深思熟虑的方法来处理错误。你必须区分可恢复错误和不可恢复错误；确保分配的资源在发生错误时得到清理；并向用户发送适当的错误消息。

<u>编写异常安全代码</u>

假设一个函数以独占模式打开一个文件，并且在文件关闭之前有一个错误条件终止了程序。程序被终止之后，该文件将保持锁定后，用户将无法访问该文件，直到计算机重新启动。为了防止这类问题，你必须使您的程序异常安全。换句话说，程序必须在异常或其他错误情况下清理所有东西。可能需要清理的东西包括：

1. 使用new或者malloc分配的内存。
2. 窗口、图形画刷等的句柄。
3. 锁定的互斥量。
4. 打开的数据库连接。
5. 打开的文件和网络连接。
6. 需要被删除的临时文件。
7. 需要保存的用户工作。
8. 任何其它已分配的资源。

C++处理清理工作的方法是创建一个析构函数。可以将读取或写入文件的函数包装到具有确保文件关闭的析构函数的类中。相同的方法可以用于任何其他资源，例如动态分配的内存、窗口、互斥量、数据库连接等等。

C++异常处理系统确保调用本地对象的所有析构函数。如果包装器类有析构函数来处理分配资源的所有清理工作，则程序是异常安全的。如果析构函数引发另一个异常，则系统可能会出现问题。

如果你使用自己的错误处理系统而不是使用异常处理，那么你无法确保调用了所有析构函数并清理了资源。如果错误处理程序调用exit()、abort()、_endthread()等，则不能保证所有析构函数被调用。在不使用异常的情况下处理不可恢复错误的安全方法是从函数返回。如果可能，函数可能返回错误代码，或者错误代码可能存储在全局对象中。然后调用函数必须检查错误代码。如果后者也需要清理，那么它必须返回给自己的调用者，依此类推。

7.33 堆栈展开的其它情况

前面一节描述了一种称为堆栈展开的机制，异常处理程序使用这种机制清理和调用任何必要的析构函数，这些析构函数在出现异常时跳出函数，而不使用正常的返回路径。这种机制也适用于其他两种情况：

当线程终止时，可以使用堆栈展开机制。目的是检测线程中声明的任何对象是否具有需要调用的析构函数。建议在结束线程之前从需要清理的函数返回。你不能确保对_endthread()的调用会清除堆栈。这种行为依赖于具体的实现。

当使用longjmp函数从函数中跳出时，也使用堆栈展开机制。如果可能，避免使用longjmp。在时间相当重要的代码中不要依赖longjmp。

7.34 <code>NAN</code>和<code>INF</code>的传递

在大多数情况下，浮点错误会传播到一系列计算的最终结果。这是异常和错误捕获的一种非常有效的替代方法。

浮点溢出和除以0得到无穷大。如果你把无穷大和某数相加或相乘，结果就是无穷大。INF代码可以以这种方式传播到最终结果。然而，并不是所有使用INF输入的操作都会得到INF。如果用一个正常的数字除以INF，会得到0。特殊情况INF-INF和INF/INF得到NAN （not-a-number）。当你用0除以0以及函数的输入超出范围时，比如sqrt(-1)和log(-1)，也会出现特殊的代码NAN。

使用NAN作为输入的大多数操作将输出NAN，因此NAN将传播到最终结果。这是一种简单有效的浮点错误检测方法。几乎所有以INF或NAN形式出现的浮点错误都将传播到它们最终结果。如果打印结果，你将看到INF或NAN，而不是数字。跟踪错误不需要额外的代码，INF和NAN的传播也不需要额外的成本。

NAN可以包含带有额外信息的负载（payload）。函数库可以在出现错误时将错误代码放入此负载中，此负载将传播到最终的结果。

当参数为INF或NAN时，函数finite()将返回false，如果它是一个普通的浮点数，则返回true。这可用于在浮点数转换为整数之前检测错误，以及在其他需要检查错误的情况下。

INF和NAN传播的详细信息请参阅NAN propagation versus fault trapping in floating point code。该手册还讨论了INF和NAN的传递失败的情况，以及影响这些代码传递的编译器优化选项。

7.35 预处理命令

就程序性能而言，预处理指令（以#开头的所有指令）的性能成本很少，因为它们在程序编译之前就已经解析了。

#if指令对于支持多个平台或使用相同源代码的多个配置是很有用的。#if比if更高效，因为#if是在编译时解析的，而if是在运行时解析的。

当用于定义常量时，#define指令等价于const定义。例如，#define ABC 123 和 const int ABC = 123的效率相同的，因为在大多数情况下，编译器优化可以用它的值替换整数常量。然而，const int声明在某些情况下可能占用内存空间，而#define指令从来不占用内存空间。浮点常量总是占用内存空间，即使没有给它命名。

当作为宏使用时，#define指令有时比函数更高效。参见第48页（TODO）的讨论。

7.36 命名空间

使用名称空间，对执行速度没有影响。

8 编译器中的优化

8.1 编译器时如何优化的

现代编译器为了提高性能，会对代码进行大量修改。程序员知道编译器能做什么和不能做什么是很有用的。下面几节描述了一些编译器的优化，这些优化是程序员需要了解的。

<u>函数内联</u>

编译器可以用被调用函数的主体替换函数调用。例如：

```
// Example 8.1a
float square (float a)
{
    return a * a;
}

float parabola (float x)
{
    return square(x) + 1.0f;
```

编译器可以将对square的调用替换为square内部的代码：

```
// Example 8.1b
float parabola (float x)
{
    return x * x + 1.0f;
```

函数内联的优点是：

1. 节约了调用、返回和参数传递的开销
2. 因为代码变得连续了，代码缓存的效率会更高，
3. 如果只调用一次内联函数，那么代码就会变得更小。
4. 函数内联可以使其他优化的成为可能，如下所述。

函数内联的缺点是：如果对内联函数有多个调用且函数体很大，则代码会变得更大。 如果函数很小，或者只从一个或几个地方调用它，那么编译器更可能使函数内联。

<u>常量折叠和常数传播</u>

只包含常量的表达式或子表达式将被计算结果替换。例如：

```
// Example 8.2a
double a, b;
```

编译器将会替换成下面的代码：

```
// Example 8.2b
```

这其实很方便，使用2.0/3.0要比计算值并使用许多小数要来的容易。建议为这样的子表达式加上括号，以确保编译器将其识别为子表达式。例如，b*2.0/3.0将识别为(b*2.0)/3.0，而不是b*(2.0/3.0)，除非为常量子表达加上括号。

常量可以通过一系列的计算来传播：

```
// Example 8.3a
float parabola (float x)
{
    return x * x + 1.0f;
}
float a, b;
a = parabola (2.0f);
```

有可能被编译器替换成：

```
// Example 8.3b
a = 5.0f;
```

当表达式包含不能被内联的函数或者不能再编译时期计算的时候，常量折叠和常量传播就不可能起作用。例如：

```
// Example 8.4
```

sin函数是在一个单独的函数库中定义的，不能期望编译器能够内联这个函数并在编译时计算它。一些编译器能够在编译时计算最常见的数学函数，如sqrt和pow，但不能计算更复杂的函数，比如sin。

<u>消除指针</u>

如果指向的目标已知，则可以消除指针或引用。例如：

```
// Example 8.5a
void Plus2 (int * p)
{
    *p = *p + 2;
}
int a;
```

可能被编译器替换成：

```
// Example 8.5b
```

<u>消除公共子表达式</u>

如果相同的子表达式出现多次，那么编译器可能只计算一次。例如：

```
// Example 8.6a
int a, b, c;
b = (a+1) * (a+1);
```

可能被编译器替换成：

```
// Example 8.6b
int a, b, c, temp;
temp = a+1;
b = temp * temp;
```

<u>寄存器变量</u>

最常用的变量存储被在寄存器中（参见第27页，TODO）。

再32为系统中，整数寄存器变量的最大数量大约是6个，在64位系统中大约是14个。

在32位系统中，浮点寄存器变量的最大数量为8个，在64位系统中为16个。一些编译器很难在32位系统中生成浮点寄存器变量，除非启用了*SSE2*（或更高版本）指令集。

编译器将为寄存器变量选择最常用的变量。这包括指针和引用，它们可以存储在整数寄存器中。寄存器变量的典型候选对象是临时中间变量、循环计数器、函数参数、指针、引用、`this`指针、公共子表达式和归纳变量（见下文）。

如果一个变量的地址被取走，也就是说，如果有指向它的指针或引用，那么这个变量就不能存储在寄存器中。因此，对于可能受益于寄存器存储的变量，你应该避免使任何指针或引用。

<u>活动范围分析</u>

变量的活动范围是指变量被使用的代码范围。对于活动范围不重叠的变量，编译器优化可以使用相同的寄存器。这在可用寄存器数量有限的时候是非常有用的。例如：

```
// Example 8.7
int SomeFunction (int a, int x[])
{
    int b, c;
    x[0] = a;
    b = a + 1;
    x[1] = b;
```

```
    c = b + 1;
    return c;
```

在本例中，a、b和c可以共享同一个寄存器，因为它们的活动范围不重叠。如果c = b + 1更改为c = a + 2，那么a和b就不能使用相同的寄存器，因为它们的活动范围现在发生重叠了。

编译器通常不会将此原则用于存储在内存中的对象。它不会为不同的对象使用相同的内存区域，即使它们的活动范围不重叠。有关如何使不同的对象共享相同的内存区域的示例，请参见第91页（TODO）。

<u>合并相同的分支</u>

通过合并相同的代码片段，可以使代码更加紧凑。例如：

```
// Example 8.8a
double x, y, z; bool b;
if (b)
{
    y = sin(x);
    z = y + 1.;
}
else
{
    y = cos(x);
    z = y + 1.;
```

可能被编译器替换为：

```
// Example 8.8b
double x, y; bool b;
if (b)
{
    y = sin(x);
}
else
{
    y = cos(x);
}
```

<u>消除跳转</u>

可以通过复制它跳转到的代码来避免跳转。例如：

```
int SomeFunction (int a, bool b)
{
    if (b)
    {
        a = a * 2;
    }
    else
    {
        a = a * 3;
    }
    return a + 1;
```

这段代码从a=a*2跳转到return a+1;。编译器可以通过复制return语句来消除这个跳转：

```
```

```
// Example 8.9b
int SomeFunction (int a, bool b)
{
    if (b)
    {
        a = a * 2;
        return a + 1;
    }
    else
    {
        a = a * 3;
        return a + 1;
    }
```

如果条件可以被简化为永远为真或永远为假，则可以消除分支：

```
// Example 8.10a
if (true)
{
    a = b;
}
else
{
    a = c;
```

可以被简化为：

```
// Example 8.10b
```

如果可以从前一个分支知道某个分支的情况，那么也可以删除该分支。例如：

```
// Example 8.11a
int SomeFunction (int a, bool b)
{
    if (b)
    {
        a = a * 2;
    }
    else
    {
        a = a * 3;
    }
    if (b)
    {
        return a + 1;
    }
    else
    {
        return a - 1;
    }
```

编译器可能会把这个简化成：

```
// Example 8.11b
int SomeFunction (int a, bool b)
```

```
{
    if (b)
    {
        a = a * 2;
        return a + 1;
    }
    else
    {
        a = a * 3;
        return a - 1;72
    }
```

<u>循环展开</u>

如果需要高度优化，一些编译器将展开循环。见45页(TODO)。如果循环体非常小，或者它为进一步优化提供了可能性，那么这可能是有利的。重复计数非常小的循环可以完全展开，以避免循环开销。例如：

```
// Example 8.12a
int i, a[2];
for (i = 0; i < 2; i++)
```

编译器可能会把这个简化成：

```
// Example 8.12b
int a[2];
a[0] = 1;
```

不幸的是，一些编译器展开太多。过多的循环展开不是最优的，因为它会占用太多的代码缓存空间，并且会填满某些微处理器的循环缓冲区。在某些情况下，关闭编译器中的循环展开选项是有用的。

<u>移动循环中的不变代码</u>

如果计算独立于循环计数器，则可以将其移出循环。例如：

```
// Example 8.13a
int i, a[100], b;
for (i = 0; i < 100; i++)
{
    a[i] = b * b + 1;
```

可能会被编译器改成这样：

```
// Example 8.13b
int i, a[100], b, temp;
temp = b * b + 1;
for (i = 0; i < 100; i++)
{
    a[i] = temp;
```

<u>归纳变量（Induction variables）</u>

循环计数器的线性函数表达式可以通过在前一个值上添加一个常数来计算。例如：

```
// Example 8.14a
```

```
int i, a[100];
for (i = 0; i < 100; i++)
{
    a[i] = i * 9 + 3;
```

编译器可能会将其改成下面的形式以避免乘法：

```
// Example 8.14b
int i, a[100], temp;
temp = 3;
for (i = 0; i < 100; i++)
{
    a[i] = temp;
    temp += 9;
```

归纳变量常用于计算数组元素的地址。例如：

```
// Example 8.15a
struct S1 {double a; double b;};
S1 list[100]; int i;
for (i = 0; i < 100; i++)
{
    list[i].a = 1.0;
    list[i].b = 2.0;
```

为了访问`list`的元素，编译器必须计算它的地址。`list[i]`的地址等于`list`的起始地址加上`i*sizeof(S1)`。这是一个关于`i`的线性函数，这是可以通过归纳变量计算的。编译器可以使用相同的归纳变量来访问`list[i].a`和`list[i].b`。当可以提前计算归纳变量的最终值时，也可以消去`i`，用归纳变量作为循环计数器。这可以将代码简化为：

```
// Example 8.15b
struct S1 {double a; double b;};
S1 list[100], *temp;
for (temp = &list[0]; temp < &list[100]; temp++)
{
    temp->a = 1.0;
    temp->b = 2.0;
```

因子`sizeof(S1) = 16`实际上隐藏在示例8.15b中的C++语法后面。`& 8list[100]`的整数表示形式为`(int)(&list[100]) = (int)(&list[0]) + 100*16`，而`temp++`实际上是在`temp`的整数值上加上16。

编译器不需要归纳变量来计算简单类型的数组元素的地址，因为CPU有硬件支持计算一个数组元素的地址，如果地址可以表示为一个基地址加上一个常数加上索引乘以一个系数1，2，4或8,但不是任何其他因数。如果在例 8.15a中的`a`和`b`是`float`而不是`double`，那么`sizeof(S1)`的值将是8，那么就不需要归纳变量了，因为CPU有硬件支持 index乘上8。

我研究的编译器不为浮点表达式或更复杂的整数表达式生成归纳变量。有关如何使用归纳变量计算多项式的示例，请参见第81页（TODO）。

<u>调度</u>

编译器可以为了并行执行对指令重新排序。例如：

```
// Example 8.16
float a, b, c, d, e, f, x, y;
x = a + b + c;
y = d + e + f;
```

在这个例子中，编译器可以交错这两个公式,先算a + b，然后是d + e，然后将c加到第一个和中，那么f被加到第二个和中，第一个结果是存储在x中，最后第二个结果存储在y中。这样做的目的是帮助*CPU*实现多个并行计算。现代*CPU*实际上可以在没有编译器帮助的情况下对指令进行重新排序（参见第105页，TODO），但是编译器可以使*CPU*更容易地对指令进行重新排序。

<u>代数约简</u>

多数编译器可以使用代数的基本定律来约简简单的代数表达式。例如，编译器可以将表达式-(-a)更改为a。

我不认为程序员会经常写出像-(-a)这样的表达式，但是这种表达式可能是其他优化（如函数内联）的结果。可约表达式也经常作为宏展开的结果出现。

然而，程序员经常编写可以约简的表达式。这可能是因为未约简的表达式更好地解释了程序背后的逻辑，或者因为程序员没有考虑代数约简的可能性。例如，程序员可能更喜欢使用if(!a && !b)而不是同等的if(!(a || b))即使后者少一个运算符。幸运的是，在这种情况下，所有编译器都能够进行约简。

你不能指望编译器约简复杂的代数表达式。例如，在我测试的编译器中，只有一个编译器能够将(a*b*c)+(c*b*a)约简为a*b*c*2。在编译器中实现很多代数规则是相当困难的。一些编译器可以约简某些类型的表达式，而另一些编译器可以约简其他类型的表达式，但我所见过的编译器不能都约简它们。在布尔代数中，可以实现一种通用算法（例如，Quine-McCluskey或者Espresso）来约简任何表达式，但我测试过的编译器似乎都没有这样做。

编译器在约简整数表达式上比浮点表达式做得更好，尽管这两种情况下的代数规则是相同的。这是因为浮点表达式的代数操作可能会产生不希望的效果。这种效果可以用下面的例子来说明：

```
// Example 8.17
char a = -100, b = 100, c = 100, y;
```

这里y的值是$-100 + 100 + 100 = 100$。现在，根据代数规则，我们可以这样写：

```
y = c + b + a;
```

如果子表达式c+b可以在其他地方重用，那么这可能很有用。在这个例子中，我们使用的是8位整数，范围从-128到+127。整数溢出将使值反转（wrap around）。127加1等于-128，减1等于-128。计算c+b会产生溢出，结果是-56而不是200。接下来，我们将-100加到-56中，这将产生一个下溢，得到100，而不是-156。令人惊讶的是，我们得到了正确的结果，因为上溢和下溢相互抵消了。这就是为什么对整数表达式使用代数操作是安全的（除了<、<=、>和>=操作符）。

同样的讨论不适用于浮点表达式。浮点变量在上溢和下溢时，不会反转。浮点变量的范围非常大，除了在特殊的数学应用中，我们不必太担心上溢和下溢。但是我们必须担心精度的损失。让我们用浮点数重复上面的例子：

```
// Example 8.18
float a = -1.0E8, b = 1.0E8, c = 1.23456, y;
```

这里的计算结果先得出a+b=0，然后0+1.23456 = 1.23456。但是如果我们改变操作数的顺序，先加上b和c，就不会得到相同的结果。b + c = 100000001.23456。浮点类型的精度大约为7位有效数字，因此b+c的值四舍五入为100000000。把a加到这个数上得到0，而不是1.23456。

这里讨论的结果是，改变浮点操作数的顺序，就有丢失精度的风险。除非你指定一个允许不需要精确的浮点运算的选项，否则编译器不会这么做。即使打开了所有相关的优化选项，编译器也不会执行诸如0/a = 0这样的明显简化，因为如果a为0、无穷大或NAN（不是一个数字），这将是无效的。不同的编译器的行为不同，因为对于哪些不精确应该允许，哪些不允许，存在不同的观点。

不能依赖编译器对浮点代码执行任何代数消减，只能依赖于对整数代码进行最简单的缩减。手动消减会更安全。我测试了在7个不同的编译器，简化各种代数表达式的能力。结果如下表8.1所示。

<u>实体化（Devirtualization）</u>

如果知道所需要虚函数的个版本，编译器优化可以绕过虚拟表查找，直接调用虚拟函数。例如：

```
// Example 8.19. Devirtualization
class C0
{
public:
```

```
    virtual void f();
};
class C1 : public C0
{
public:
    virtual void f();
};
void g()
{
    C1 obj1;
    C0 * p = & obj1;
    p->f(); // Virtual call to C1::f
```

如果不进行优化，编译器需要在虚拟表中查找`p->f()`调用是否转到`C0::f`或`C1::f`。但是编译器优化将看到`p`总是指向类`C1`的对象，因此它可以直接调用`C1::f`，而不使用虚拟表。不幸的是，很少有编译器能够进行这种优化。

8.2 不同编译器的对比

我在9种不同的C++编译器上做了一系列的实验，看看它们是否能够进行不同种类的优化。结果见表8.1。该表显示了不同的编译器是否成功地在我的测试示例中应用了各种优化方法和代数约简。

该表可以提供一些指导，说明你可以期望特定的编译器获得哪些优化，以及必须手动进行哪些优化。

必须强调的是，编译器在不同的测试示例上可能有不同的行为。你不能期望编译器总是根据表格的结果来运行。

Optimization method	Microsoft	Borland	Intel	Gnu	PathScale	PGI	Digital Mars	Watcom	Codeplay
Function inlining	X	-	X	X	X	X	-	-	X
Constant folding	X	X	X	X	X	X	X	X	X
Constant propagation	X	-	X	X	X	X	-	-	X
Pointer elimination	X	X	X	X	X	X	X	X	X
Common subexpression elimin, integer	X	(X)	X	X	X	X	X	X	X
Common subexpression elimin, float	X	-	X	X	X	X	-	X	X
Register variables, integer	X	X	X	X	X	X	X	X	X
Register variables, float	X	-	X	X	X	X	-	X	X
Live range analysis	X	X	X	X	X	X	X	X	X
Join identical branches	X	-	-	X	-	-	-	X	-
Eliminate jumps	X	X	X	X	X	X	-	X	X
Eliminate branches	X	-	X	X	X	X	-	-	-
Remove branch that is always true/false	X	-	X	X	X	X	X	X	X
Loop unrolling	X	-	X	X	X	X	-	-	X
Loop invariant code motion	X	-	X	X	X	X	X	X	X
Induction variables for array elements	X	X	X	X	X	X	X	X	X
Induction variables for other integer expressions	X	-	X	X	X	-	X	X	X

Optimization method	Microsoft	Borland	Intel	Gnu	PathScale	PGI	Digital Mars	Watcom	Codeplay
Induction variables for float expressions	-	-	-	-	-	-	-	-	-
Automatic vectorization	-	-	X	X	X	X	-	-	X
Devirtualization	-	-	-	X	-	-	-	-	-
Profile-guided optimization	X	-	X	X	X	X	-	-	-
Whole program optimization	X	-	X	X	X	-	-	-	-
Integer algebra reductions:									
`a+b = b+a`	X	(X)	X	X	X	X	-	X	X
`a*b = b*a`	X	(X)	X	X	X	X	-	X	X
`a+b+c = a+(b+c)`	X	-	X	X	-	-	X	X	-
`a+b+c = c+a+b`	X	-	-	X	-	-	-	-	-
`a+b+c+d = (a+b)+(c+d)`	-	-	X	X	-	-	-	-	-
`a*b+a*c = a*(b+c)`	X	-	X	X	X	-	-	-	X
`a*x*x*x+b*x*x+c*x+d =  ((a*x+b)*x+c)*x+d</br>`	X	-	X	X	X	-	X	X	X
`X*X*X*X*X*X=((X<sup>2</sup>)<sup>2</sup>)<sup>2</sup>`	-	-	-	X	-	-	-	-	-
`a+a+a+a=a*4`	X	-	X	X	-	-	-	-	X
`-(-a)=a`	X	-	X	X	X	X	X	X	-
`a-(-b)=a+b`	X	-	X	X	X	X	-	X	-
`a-a = 0`	X	-	X	X	X	X	X	X	X
`a+0 = a`	X	X	X	X	X	X	X	X	X
`a*0 = 0`	X	X	X	X	X	X	X	-	X
`a*1 = a`	X	X	X	X	X	X	X	X	X
`(-a)*(-b) = a*b`	X	-	X	X	X	-	-	-	-
`a/a = 1`	-	-	-	-	X	-	-	-	X
`a/1 = a`	X	X	X	X	X	X	X	X	X
`0/a = 0`	-	-	-	X	X	-	-	X	X
`(-a == -b) = (a == b)`	-	-	-	X	X	-	-	-	-
`(a-c == b+c) = (a == b)`	-	-	-	-	X	-	-	-	-
`!(a < b) = (a >= b)`	X	X	X	X	X	X	X	X	X
`(a<b && b<c && a<c) = (a<b && b<c)`	-	-	-	-	-	-	-	-	-
Multiply by constant = shift and add	X	X	X	X	-	X	X	X	-
Divide by constant = multiply and shift	X	-	X	X	X	(-)	X	-	-

Optimization method	Microsoft	Borland	Intel	Gnu	PathScale	PGI	Digital Mars	Watcom	Codeplay
Floating point algebra reductions:									
`a+b = b+a`	X	-	X	X	X	X	-	-	X
`a*b = b*a`	X	-	X	X	X	X	-	-	X
`a+b+c = a+(b+c)`	X	-	X	X	-	-	-	-	-
`(a+b)+c = a+(b+c)`	-	-	X	X	-	-	-	-	-
`a*b*c = a*(b*c)`	X	-	X	-	-	-	-	-	-
`a+b+c+d = (a+b)+(c+d)`	-	-	-	X	-	-	-	-	-
`a*b+a*c = a*(b+c)`	X	-	-	-	X	-	-	-	-
`a*x*x*x+b*x*x+c*x+d =  ((a*x+b)*x+c)*x+d</br>`	X	-	X	X	X	-	-	-	-
`X*X*X*X*X*X=((X<sup>2</sup>)<sup>2</sup>)<sup>2</sup>`	-	-	-	X	-	-	-	-	-
`a+a+a+a=a*4`	X	-	X	X	-	-	-	-	-
`-(-a)=a`	-	-	X	X	X	X	X	X	-
`a-(-b)=a+b`	-	-	-	X	X	X	-	X	-
`a+0 = a`	X	-	X	X	X	X	X	X	-
`a*0 = 0`	-	-	X	X	X	X	-	X	X
`a*1 = a`	X	-	X	X	X	X	X	-	X
`(-a)*(-b) = a*b`	-	-	-	X	X	X	-	-	-
`a/a = 1`	-	-	-	-	X	-	-	-	X
`a/1 = a`	X	-	X	X	X	-	X	-	-
`0/a = 0`	-	-	-	X	X	-	-	X	X
`(-a == -b) = (a == b)`	-	-	-	X	X	-	-	-	-
`(-a > -b) = (a < b)`	-	-	X	X	-	-	-	-	X
Divide by constant = multiply by reciprocal	X	X	-	X	X	-	-	X	-
Boolean algebra reductions:									
`!(!a) = a`	X	-	X	X	X	X	X	X	X
`(a&&b)||(a&&c) = a&&(b||c)`	X	-	X	X	X	-	-	-	-
`!a && !b = !(a||b)`	X	X	X	X	X	X	X	X	X
`a && !a = false,a || !b = true`	X	-	X	X	X	X	-	-	-
`a && true = a,a || false = a`	X	X	X	X	X	X	X	X	-
`a && false = false,a || true = true`	X	-	X	X	X	X	X	X	-

Optimization method	Microsoft	Borland	Intel	Gnu	PathScale	PGI	Digital Mars	Watcom	Codeplay
`a && a = a`	X	-	X	X	X	X	-	-	-
`(a&&b) || (a&&!b) = a`	X	-	-	X	X	-	-	-	-
`(a&&b) || (!a&&c) = a ? b : c`	X	-	X	X	-	-	-	-	-
`(a&&b) || (!a&&c) || (b&&c) = a ? b : c`	X	-	-	X	-	-	-	-	-
`(a&&b) || (a&&b&&c) = a&&b`	X	-	-	X	X	-	-	-	-
`(a&&b) || (a&&c) || (a&&b&&c) = a&&(b||c)`	X	-	-	X	X	-	-	-	-
`(a&&!b) || (!a&&b) = a XOR b`	-	-	-	-	-	-	-	-	-
Bit vector algebra reductions:									
`~(~a) = a`	X	-	X	X	X	X	X	-	-
`(a&b)|(a&c) = a&(b|c)`	X	-	X	X	X	X	-	-	X
`(a|b)&(a|c) = a|(b&c)`	X	-	X	X	X	X	-	-	X
`~a & ~b = ~(a | b)`	-	-	X	X	X	X	-	-	-
`a & a = a`	X	-	-	X	X	X	-	-	X
`a & ~a = 0`	-	-	-	X	X	X	-	-	-
`a & -1 = a,a | 0 = a`	X	-	X	X	X	X	X	X	X
`a & 0 = 0,a | -1 = -1`	X	-	X	X	X	X	X	X	X
`(a&b) | (~a&c) | (b&c) = (a&b) | (~a&c)`	-	-	-	-	-	-	-	-	-
`a&b&c&d = (a&b)&(c&d)`	-	-	-	X	-	-	-	-	-
`a ^ 0 = a`	X	X	X	X	X	-	X	X	X
`a ^ -1 = ~a`	X	-	X	X	X	-	X	X	-
`a ^ a = 0`	X	-	X	X	X	X	-	X	X
`a ^ ~a = -1`	-	-	-	X	X	X	-	-	-
`(a&~b) | (~a&b) = a ^ b`	-	-	-	-	-	-	-	-	-
`~a ^ ~b = a ^ b`	-	-	-	X	X	-	-	-	-
`a<<b<<c = a<<(b+c)`	X	-	X	X	X	-	-	X	X
Integer XMM (vector) reductions:									
Common subexpression elimination	X	n.a.	X	X	X	-	n.a.	n.a.	X
Constant folding	-	n.a.	-	X	-	-	n.a.	n.a.	-
`a+b = b+a`, `a*b = b*a`	-	n.a.	-	X	-	-	n.a.	n.a.	X

Optimization method	Microsoft	Borland	Intel	Gnu	PathScale	PGI	Digital Mars	Watcom	Codeplay
`(a+b)+c = a+(b+c)`	-	n.a.	-	-	-	-	n.a.	n.a.	-
`a*b+a*c = a*(b+c)`	-	n.a.	-	-	-	-	n.a.	n.a.	-
`X*X*X*X*X*X=((X<sup>2</sup>)<sup>2</sup>)<sup>2</sup>`	-	n.a.	-	-	-	-	n.a.	n.a.	-
`a+a+a+a = a*4`	-	n.a.	-	-	-	-	n.a.	n.a.	-
`-(-a) = a`	-	n.a.	-	-	-	-	n.a.	n.a.	-
`a-a = 0`	-	n.a.	X	-	-	-	n.a.	n.a.	-
`a+0 = a`	-	n.a.	-	-	-	-	n.a.	n.a.	-
`a*0 = 0`	-	n.a.	-	X	-	-	n.a.	n.a.	-
`a*1 = a`	-	n.a.	-	X	-	-	n.a.	n.a.	-
`(-a)*(-b) = a*b`	-	n.a.	-	-	-	-	n.a.	n.a.	-
`!(a < b) = (a >= b)`	-	n.a.	-	-	-	-	n.a.	n.a.	-
Floating point XMM (vector) reductions:									
`a+b = b+a`, `a*b = b*a`	X	n.a.	-	X	-	-	n.a.	n.a.	X
`(a+b)+c = a+(b+c)`	-	n.a.	-	-	-	-	n.a.	n.a.	-
`a*b+a*c = a*(b+c)`	-	n.a.	-	-	-	-	n.a.	n.a.	-
`-(-a) = a`	-	n.a.	-	-	-	-	n.a.	n.a.	-
`a-a = 0`	-	n.a.	-	X	-	-	n.a.	n.a.	-
`a+0 = a`	-	n.a.	X	-	-	-	n.a.	n.a.	-
`a*0 = 0`	-	n.a.	X	-	-	-	n.a.	n.a.	-
`a*1 = a`	-	n.a.	-	X	-	-	n.a.	n.a.	-
`a/1 = a`	-	n.a.	-	X	-	-	n.a.	n.a.	-
`0/a = 0`	-	n.a.	X	X	-	-	n.a.	n.a.	-
Divide by constant = multiply by reciprocal	-	n.a.	-	-	-	-	n.a.	n.a.	-
Boolean XMM (vector) reductions:									
`~(~a) = a`	-	n.a.	-	-	-	-	n.a.	n.a.	-
`(a&b)|(a&c) = a&(b|c)`	-	n.a.	-	-	-	-	n.a.	n.a.	-
`a & a = a`, `a | a = a`	-	n.a.	X	X	-	-	n.a.	n.a.	-
`a & ~a = 0`	-	n.a.	-	X	-	-	n.a.	n.a.	-
`a & -1 = a`, `a | 0 = a`	-	n.a.	-	-	-	-	n.a.	n.a.	-
`a & 0 = 0`	-	n.a.	-	X	-	-	n.a.	n.a.	-
`a | -1 = -1`	-	n.a.	-	-	-	-	n.a.	n.a.	-
`a ^ a = 0`	-	n.a.	X	X	-	-	n.a.	n.a.	-

Optimization method	Microsoft	Borland	Intel	Gnu	PathScale	PGI	Digital Mars	Watcom	Codeplay
`andnot(a,a) = 0`	-	n.a.	-	X	-	-	n.a.	n.a.	-
`a<<b<<c = a<<(b+c)`	-	n.a.	-	-	-	-	n.a.	n.a.	-

Tabel 8.1. Comparison of optimizations in different C++ compilers

测试中所有相关的优化选项都被打开，包括放宽浮点精度。被测试的编译器版本如下：

1. Microsoft C++ Compiler v. 14.00 for 80x86 / x64 (Visual Studio 2005).
2. Borland C++ 5.82 (Embarcadero/CodeGear/Borland C++ Builder 5, 2009).
3. Intel C++ Compiler v. 11.1 for IA-32/Intel64, 2009.
4. Gnu C++ v. 4.1.0, 2006 (Red Hat).
5. PathScale C++ v. 3.1, 2007.
6. PGI C++ v. 7.1-4, 2008.
7. Digital Mars Compiler v. 8.42n, 2004.
8. Open Watcom C/C++ v. 1.4, 2005.
9. Codeplay VectorC v. 2.1.7, 2004.

没有发现*Microsoft*、*Intel*、*Gnu*和*PathScale*编译器的32位和64位代码的优化功能有任何差异。

8.3 编译器优化的障碍

有几个因素妨碍编译器执行我们希望它完成的优化。对于程序员来说，了解这些障碍并知道如何避免它们是很重要的。下面将讨论优化的一些重要障碍。

<u>无法跨模块优化</u>

编译器除了正在编译的模块外，没有关于其他模块中的函数的信息。这阻止了它对函数调用进行优化。例如：

```
// Example 8.20
<u>module1.cpp</u>
int Func1(int x)
{
    return x*x + 1;
}

<u>module2.cpp</u>
int Func2()
{
    int a = Func1(2);
    ...
}
```

假如`Func1`和`Func2`在同一个模块中，那么编译器将能够进行函数内联和常量传播，并将`a`约简为常量5。但是在编译`module2.cpp`时，编译器没有关于`Func1`的必要信息。

解决这个问题最简单的方法是使用`#include`指令将多个`.cpp`模块组合成一个模块。这个方法适用于所有编译器。有些编译器有一个称为"全程序优化"的特性，它将支持跨模块的优化（参见第83页）。

<u>指针别名（pointer aliasing）</u>

当通过指针或引用访问变量时，编译器可能无法完全排除所指向的变量与代码中的其他变量相同的可能性。例如：

```
// Example 8.21
void Func1 (int a[], int * p)
{
    int i;
    for (i = 0; i < 100; i++)
```

```
    {
        a[i] = *p + 2;
    }
}

void Func2()
{
    int list[100];
    Func1(list, &list[8]);
```

在这里，需要重新加载 *p 并计算 *p+2 100次，因为p所指向的值与循环过程中会发生变化的a[]中的一个元素相同。不允许假定*p+2是可以移出循环的循环不变代码。例8.21确实是一个非常做作的例子，但关键是编译器不能排除这种做作例子存在的理论可能性。因此，编译器不能假设*p+2是一个可以移动到循环外部的循环不变表达式。

大多数编译器都有一个假设没有指针别名（*/Oa*）的选项。克服可能的指针别名障碍的最简单方法是打开这个选项。这要求你仔细分析代码中的所有指针和引用，以确保在代码的同一部分中没有以一种以上的方式访问任何变量或对象。如果编译器支持的，还可以通过使用关键字__restrict或__restrict__告诉编译器某个特定指针不是任何变量的别名。

我们永远不能确定编译器是否接受关于没有指针别名的提示。确保代码得到优化的唯一方法是显式地进行优化。在示例8.21中，如果你确信指针不是数组中的任何元素的别名，那么可以先计算*p+2并将其存储在循环外部的临时变量中。这种方法要求你能够预先知道优化的障碍在哪里。

<u>动态内存分配</u>

动态分配（使用new或malloc）的任何数组或对象都必须通过指针进行访问。对于程序员来说，指向不同动态分配的对象的指针没有重叠或混淆是很明显的，但是编译器通常看不到这一点。它还阻止了编译器以最佳方式对齐数据，或者阻止编译器知道对象是对齐的。最好在需要的函数中声明对象和固定大小的数组。

<u>纯函数（Pure functions）</u>

纯函数是一个没有副作用（side-effects）的函数，它的返回值只取决于参数的值。这与"函数"的数学概念密切相关。

多次调用具有相同参数的纯函数肯定会得到相同的结果。编译器可以消除包含纯函数调用的常见子表达式，并且可以移出包含纯函数调用的循环不变代码。不幸的是，如果函数是在不同的模块或函数库中定义的，编译器则无法知道函数是否是纯函数。

因此，当涉及纯函数调用时，有必要手动进行优化，如公共子表达式消除、常量传播和移动循环不变代码。

*Gnu*编译器和用于*Linux*的*Intel*编译器都有一个属性，该属性可以用于声明函数原型，以告诉编译器这是一个纯函数。例如：

```
// Example 8.22
#ifdef __GNUC__
#define pure_function __attribute__((const))
#else
#define pure_function
#endif

double Func1(double) pure_function ;
double Func2(double x)
{
    return Func1(x) * Func1(x) + 1.;
```

在这里，*Gnu*编译器将只调用Func1一次，而其他编译器将会调用两次。

其他一些编译器（*Microsoft*，*Intel*)知道像sqrt、pow和log这样的标准库函数是纯函数，但不幸的是，无法告诉这些编译器用户定义的函数是纯函数。

<u>虚函数和函数指针</u>

编译器几乎不可能确定地预测将调用虚拟函数的哪个版本，或者函数指针指向什么。因此，它不能内联函数，也不能对函数调用进行优化。

<u>代数约简</u>

大多数编译器可以做简单的代数约简，比如`-(a) = a`，但是它们不能做更复杂的约简。代数约简是一个复杂的过程，很难在编译器中实现。

由于数学的纯粹性（mathematical purity），许多代数约简是不被允许的。在许多情况下，可以构造一些晦涩的例子，其中约简会导致溢出或精度损失，特别是在浮点表达式中（参见第74页TODO）。编译器不能排除特定情况下某个特定约简无效的可能性，但是程序员可以。因此，在许多情况下有必要显式地进行代数约简。

整数表达式不太容易出现溢出和精度损失的问题，原因见第74页（TODO）。因此，编译器可以对整数表达式进行比浮点数表达式更多的约简。大多数涉及整数加法、减法和乘法的约简在所有情况下都是被允许的，而许多涉及除法和关系运算符（如"`>`"）的约简，由于数学的纯粹性是不被允许的。例如，由于存在隐藏的溢出的可能性，编译器不能将整数表达式`-a > -b`约简到`a < b`。

表8.1显示了编译器在某些情况下，能够进行哪些约简，以及不能进行哪些约简。编译器无法完成的所有约简都必须由程序员手动完成。

<u>浮点归纳变量</u>

编译器不能生成浮点归纳变量的原因与它们不能对浮点表达式进行代数约简的原因相同。因此，有必要手动完成这项工作。通过上一次的值计算循环计数器的函数比使用循环计数器直接计算更高效，这个规则就很有用。循环计数器的n次多项式的任何表达式都可以通过n次加法来计算，而不需要使用乘法。下面的例子展示了使用加法二阶多项式的原理：

```
// Example 8.23a. Loop to make table of polynomial
const double A = 1.1, B = 2.2, C = 3.3;  // Polynomial coefficients
double Table[100];                       // Table
int x; // Loop counter
for (x = 0; x < 100; x++)
{
    Table[x] = A*x*x + B*x + C;          // Calculate polynomial
```

这个多项式的计算通过两个归纳变量，只需要两个加法就可以完成：

```
// Example 8.23b. Calculate polynomial with induction variables
const double A = 1.1, B = 2.2, C = 3.3; // Polynomial coefficients
double Table[100];                      // Table
int x;                                  // Loop counter
const double A2 = A + A;                // = 2*A
double Y = C;                           // = A*x*x + B*x + C
double Z = A + B;                       // = Delta Y
for (x = 0; x < 100; x++)
{
    Table[x] = Y;                       // Store result
    Y += Z;                             // Update induction variable Y
    Z += A2;                            // Update induction variable Z
}
```

例8.23b中的循环中有两个循环依赖链（*loop-carried dependency chain*），即两个归纳变量`Y`和`Z`。每个依赖链都有一个延迟，这个延迟与浮点加法的延迟相同。这延迟足够小，证明该方法是合适的。一个较长的循环依赖链会使归纳变量方法变得不利，除非该值是从一个两次或多次迭代的值计算出来的。

归纳变量的方法也可以向量化，如果你考虑到每个值都是从序列中位于`r`位置的值计算出来的，其中`r`是一个向量中的元素数或循环展开因子。要在每种情况下找到正确的公式，需要一点数学知识。

<u>内联函数的非内联副本</u>

函数内联的复杂性在于，同一个函数可能从另一个模块调用。为了在另一个模块中调用该函数，编译器必须生成一个内联函数的非内联的副本，。如果没有其他模块调用这个函数，那么这个非内联副本就是无用代码。这种代码片段降低了缓存的效率。

有很多方法可以解决这个问题。如果一个函数没有被任何其他模块引用，那么将关键字`static`添加到函数定义中。这将告诉编译器不能从任何其他模块调用该函数。静态声明使编译器更容易评估使函数内联是否是最优的，并防止编译器生成未使用的内联函数副本。`static`关键字还使各种其他优化成为可能，因为编译器不必遵守任何特定的函数调用约定，而这些函数在其他模块中是无法访问的。可以用`static`声明所有本地非成员函数。

不幸的是，这个方法并不适用于类成员函数，因为`static`关键字对于成员函数有不同的含义。可以通过在类定义中声明函数体来强制使成员函数内联。这将防止编译器生成函数的非内联副本，但它的缺点是，即使在不适合内联的情况下（例如，如果成员函数很大，并且从许多不同的地方调用），函数也总是内联的。

一些编译器有一个选项(*Windows*：`/Gy`,* Linux*：`- fffunction -sections`)，允许链接器删除未引用的函数。建议打开此选项。

8.4 CPU优化的障碍

现代*CPU*可以通过无序执行指令来进行大量优化。如第22页（TODO）所述，代码中的长依赖链妨碍*CPU*的无序执行。避免长依赖链，特别是具有长延迟的循环依赖链。

8.5 编译器优化选项

所有*C++*编译器都有各种各样的优化选项，你可以打开或关闭它们。研究正在使用的编译器的可用选项并打开所有相关选项是非常重要的。

许多优化选项与调试不兼容。调试器可以一次一行地执行代码，并显示所有变量的值。显然，当部分代码被重新排序、内联或优化时，这是不可能的。生成两个版本的可执行文件是很常见的：一个带有完整调试支持的调试版本（在程序开发期间使用）和一个带有所有相关优化选项的发布版本。大多数*IDE*（集成开发环境）都有用于生成目标文件和可执行文件的调试版本和发布版本的工具。确保能够区分这两个版本，并在可执行文件的优化版本中关闭调试和性能分析支持。

大多数编译器都提供了大小优化和速度优化的选择。当代码非常快时，你希望可执行文件尽可能小；或者当代码缓存非常关键时，优化大小是非常重要的。当*CPU*访问和内存访问是消耗巨大时，速度优化是与之相关的。选择可用的最大优化选项。

一些编译器提供配置分析引导的优化。其工作方式如下。首先，编译程序并使之支持分析。然后使用分析器进行测试运行，分析器确定程序流以及每个函数和分支执行的次数。然后，编译器可以使用这些信息来优化代码，并将不同的函数按最佳顺序排列。

一些编译器支持全程序优化。这可以通过两个步骤进行编译。所有源文件首先被编译成中间文件格式，而不是通常的目标文件格式。然后在第二步中将中间文件链接在一起后完成编译。寄存器分配和函数内联是在第二步中完成的。中间文件格式没有标准化。它甚至不兼容同一编译器的不同版本。因此，不可能以这种格式分发函数库。

其他编译器提供了将多个`.cpp`文件编译为单个对象文件的可能性。这使编译器能够在启用过程间优化时进行跨模块优化。一种更原始但更有效的方法是通过`#include`指令将所有源文件连接到一个文件中，并声明所有函数为静态或内联的。这将使编译器能够对整个程序进行过程间优化。

在*CPU*发展的历史中，每一代*CPU*都增加了可用的指令集，更新的指令集使得编译器可以生成更高效的代码，但这使得代码与旧的*CPU*不兼容。*奔腾Pro*指令集使浮点数比较更高效。所有现代*CPU*都支持这个指令集。*SSE2*指令集非常有意思，因为它使浮点代码在某些情况下更高效，并使使用向量指令成为可能（参见第107页，TODO）。然而，使用*SSE2*指令集并不总是最优的。在某些情况下，*SSE2*指令集使浮点代码变的更慢，特别是在代码混合浮点和双精度浮点时（参见第143页，TODO）。目前大多数*CPU*和操作系统都支持*SSE2*指令集。

当不需要兼容旧*CPU*时，可以选择较新的指令集。更好的方法是，你可以使代码中最关键部分的有多个版本支持不同的*CPU*。这个方法在第125页（TODO）有解释。

当没有异常处理时，代码会变得更高效。建议关闭对异常处理的支持，除非代码依赖于结构化异常处理，并且你希望代码能够从异常中恢复。见62页（TODO）。

建议关闭对运行时类型标识（**RTTI**）的支持。参见第55页（TODO）。

建议启用快速浮点运算或关闭对严格浮点运算的要求，除非要求严格。参见第74页的讨论（TODO）。

如果选项"函数级链接（*function level linking*）"可用，可以打开该选项。有关此选项的解释，请参见第82页（TODO）。

如果你确定代码没有指针别名，请使用"假设没有指针混叠（*assume no pointer aliasing*）"选项。有关解释，请参阅第79页（TODO）。（*Microsoft编译器*仅在*专业版*和*企业版*中支持此选项）。

不要修正"`FDIV bug`"。`FDIV bug`是最老版本中的一个小错误。*奔腾CPU*，在一些罕见的浮点除法情况下可能会导致轻微的不精确。修正了*FDIV bug*将导致浮点除法变慢。

许多编译器都有"标准堆栈帧（*standard stack frame*）"或"帧指针（*frame pointer*）"选项。标准堆栈帧用于调试和异常处理。省略标准堆栈帧使函数调用更快，并节省出一个额外的寄存器用于其他目的。这是有利的，因为寄存器是一种稀缺资源。除非程序依赖异常处理，否则不要使用堆栈帧。

8.6 优化指令

一些编译器有许多关键字和指令，用于在代码中的特定位置给出特定的优化指令。其中许多指令是特定于编译器的。你不能期望*Windows编译器*的指令在*Linux编译器*上工作，反之亦然。但是大多数*Microsoft指令*可以在*Intel编译器*和*Gnu编译器*的*Windows*版本上工作，而大多数*Gnu*指令也可用于*PathScale*和*Intel*编译器的*Linux*版本。

<u>适用于所有C++编译器的关键字</u>

可以将register关键字添加到变量声明中，告诉编译器希望它是一个寄存器变量。register关键字只是一个提示，编译器可能不会接受提示，但是在编译器无法预测哪些变量将被最多使用的情况下，它会非常有用。

与register相反的是volatile。volatile关键字确保变量永远不会存储在寄存器中，即使是临时的。这适用于多个线程之间共享的变量，但也可以在用于测试目的时，关闭的变量的所有优化。

const关键字表示变量永远不会改变。这将允许编译器在许多情况下优化掉变量。例如：

```
// Example 8.24. Integer constant
const int ArraySize = 1000;
int List[ArraySize];
...
for (int i = 0; i < ArraySize; i++)
```

在这里，编译器可以将所有出现的ArraySize替换为1000。如果循环计数ArraySize是常量，编译器在编译时能知道它的值，则可以以更高效的方式实现示*例8.24*中的循环。将不会为整数常量分配内存，除非它的地址（&ArraySize）被占用（什么意思？）。

const指针或const引用不能更改它所指向的内容。const成员函数不能修改数据成员。建议在适当的情况下使用const关键字来为编译器提供关于变量、指针或成员函数的额外信息，因为这可能会提高优化的可能性。例如，编译器可以安全地假设类数据成员的值在调用同一类的const函数时保持不变。

根据上下文，static关键字有多种含义。当关键字static应用于非成员函数时，意味着该函数不被任何其他模块访问，这使得内联更加高效，并支持过程间优化， 见82页(TODO)；当应用于全局变量时，意味着它不被任何其他模块访问，这将支持过程间优化；当应用于函数内部的局部变量时，意味着该变量将在函数返回时保留，并在下一次调用该函数时保持不变，这可能是低效的，因为一些编译器会插入额外的代码来防止多个线程同时访问该变量。即使变量被声明为const，也可能会这样。

然而，可能有一个原因使局部变量静态，并确保它只在第一次调用函数时初始化。例如：

```
// Example 8.25
void Func ()
{
    static const double log2 = log(2.0);
    ...
```

在这里，log(2.0)只会在第一次执行Func被计算。如果没有static，将会在每次执行Func时重新计算。这样做的缺点是，函数必须检查以前是否调用过它。这比再次计算对数要快，但是将log2作为全局const变量或将其替换为计算后的值会更快。

当static关键字用于类的成员函数时表示该函数不能访问任何非静态的成员变量和成员函数。因为不需要this指针，调用静态成员函数会比非静态成员函数更快。建议在任何合适的时候将成员函数声明为静态的。

<u>特定编译器的关键字</u>

快速函数调用。__fastcall或者__attribute__((fastcall))。fastcall修饰符可以使函数调用在32位模式下更快。前两个整型参数将会在寄存器而不是栈中传递（对于CodeGear编译器则是前三个参数）。快速调用函数在编译器之间不兼容。在64位模式下不需要快速调用，因为参数已经在寄存器中传递的。

纯函数。__attribute__((const))（Linux only），制定函数位纯函数。这将允许消除公共子表达式和移动循环不变代码。参见第80页（TODO）；

假设没有指针别名。__declspec(noalias)或__restrict或#pragma optimize("a", on)，假设没有指针别名。见第79页的解释。注意到这些指令并不是总是有用的。

数据对齐。__declspec(align(16))或__atrribute__((aligned(16)))，指定数组和结构体的对齐。这对向量操作非常有用，参见第107页。

8.7 检查编译器做了什么

研究编译器生成的代码，看看它如何优化代码，这是非常有用的。有时编译器会做一些非常巧妙的事情来提高代码的效率，而有时它会做一些非常愚蠢的事情。查看编译器输出通常能够发现可以通过修改源代码来改进的内容，如下面的示例所示。

检查编译器生成的代码的最佳方法是将编译器选项设置位输出汇编语言。在大多数编译器上，可以从命令行调用编译器来，通过使用所有相关优化选项和选项*-S*或*/Fa*以输出汇编代码来实现这一点。一些系统上的*IDE*也提供输出汇编代码的选项。如果编译器有输出汇编语言的选项，则可以使用目标文件反汇编器。

请注意 *Intel 编译器* 在输出汇编代码时，有源代码注释选项：*FAs* 或 *-fsource-asm*。这个选项使汇编输出的可读性更高，但不幸的是这会妨碍某些特定的优化。如果你想要看到拥有全部优化的结果，请不要使用源代码注释选项。

还可以在调试器的反汇编窗口中看到编译器生成的代码。但是，你在调试器中看到的代码不是优化后的版本，因为调试选项阻止了优化。调试器无法在完全优化的代码中设置断点，因为它没有行号信息。通常可以使用的内联汇编指令在代码中插入一个固定的断点：*interrupt 3*。代码是`__asm int 3`，或`__asm("int 3")`，或`__debugbreak ()`。如果你在调试器中运行优化的代码（release 版本），那么它将在 *interrupt 3* 断点处中断，并显示反汇编后的代码，可能没有关于函数名和变量名的信息。记住删除断点*interrupt 3*。

下面的示例显示编译器的汇编代码输出是什么样子的，以及如何使用它来改进代码。

```
// Example 8.26a
void Func(int a[], int & r)
{
    int i;
    for (i = 0; i < 100; i++)
    {
        a[i] = r + i/2;
    }
```

对于**示例8.26a**（32位模式）生成以下汇编代码：

```
; Example 8.26a compiled to assembly:
ALIGN 4 ; align by 4
PUBLIC ?Func@@YAXQAHAAH@Z ; mangled function name
?Func@@YAXQAHAAH@Z PROC NEAR ; start of Func
; parameter 1: 8 + esp ; a
; parameter 2: 12 + esp ; r
$B1$1:                                  ; unused label
        push ebx                        ; save ebx on stack
        mov ecx, DWORD PTR [esp+8]      ; ecx = a
        xor eax, eax                    ; eax = i = 0
        mov edx, DWORD PTR [esp+12]     ; edx = r
$B1$2:                                  ; top of loop
        mov ebx, eax                    ; compute i/2 in ebx
        shr ebx, 31                     ; shift down sign bit of i
        add ebx, eax                    ; i + sign(i)
        sar ebx, 1                      ; shift right = divide by 2
        add ebx, DWORD PTR [edx]        ; add what r points to
        mov DWORD PTR[ecx+eax*4],ebx    ; store result in array
        add eax, 1                      ; i++
        cmp eax, 100                    ; check if i < 100
        jl $B1$2                        ; repeat loop if true
$B1$3:                                  ; unused label
        pop ebx                         ; restore ebx from stack
        ret                             ; return
        ALIGN 4                         ; align
```

编译器生成的大多数注释已经被我的注释（灰色）所取代。阅读和理解编译器生成的汇编代码需要一定的经验。让我详细解释一下上面的代码。看着有点怪异的名字`?Func@@YAXQAHAAH@Z`是`Func`的名称，其中添加了许多关于函数类型及其参数的信息。这叫做名称重整（*name mangling*）。汇编的名称允许使用 “?” 、“@”和“$”。有关名称重整的详细信息在手册5:“Calling conventions for different C++ compilers and operating systems”中解释。参数`a`和`r`在地址为`esp+8`和`esp+12`的堆栈上传递，并分别加载到`ecx`和`edx`中（在64位模式下，参数将在寄存器中传递，而不是在堆栈中）。`ecx`现在包含数组`a`的第一个元素的地址，edx包含`r`指向的变量的地址。引用和指针在汇编代码中是一样的。寄存

器ebx在使用之前入栈，在函数返回之前出栈。这是因为寄存器使用约定不允许函数更改ebx的值。只有寄存器eax、ecx和edx可以自由更改。循环计数器i作为寄存器变量存储在eax中。循环初始化条件i=0，已翻译成指令xor eax,eax。这是一种将寄存器设置为0的常见方法，比mov eax, 0更快。循环体从标签$B1$2开始。这只是编译器为标签选择的任意名称。它使用ebx作为计算i/2+r的临时寄存器。指令mov ebx,eax / shr ebx,31将i的符号位复制到ebx的最小有效位。接下来的两条指令是 add ebx, eax / sar ebx,1把这个加到i上然后向右移动一个位置以便将i除以2。指令add ebx, DWORD PTR [edx]加到ebx上的不是edx，而是地址位为edx中值的变量。方括号表示使用edx中的值作为内存指针。这是r所指向的变量。现在ebx包含i/2+r。下一条指令mov DWORD PTR [ecx+eax*4], ebx将这个结果存储在a[i]中。注意数组地址的计算是很高效的。ecx包含数组开头的地址。eax保存了索引i，这个索引必须乘以每个数组元素的大小（以字节为单位）才能计算出第i个元素的地址，int的大小是4。所以数组元素a[i]的地址是ecx+eax*4。结果ebx存储在地址[ecx+eax*4]。这都是在一条指令中完成的。*CPU*支持这种指令来快速访问数组元素。指令add eax,1是循环增量i++。cmp eax, 100/ jl $B1$2是循环条件i < 100。它将eax与100进行比较，如果i < 100，则跳回回$B1$2标签。pop ebx恢复在开始时保存的ebx值。ret从函数返回。

汇编代码清单显示了三件可以进一步优化的事情。我们注意到的第一件事是它对i的符号做了一些怪异的处理，以便将i除以2。编译器没有注意到i不能是负的，所以我们不需要关心符号位。我们可以通过将i声明为无符号整型数或在除以2之前将i的类型转换为无符号整型数，来告诉编译器这一点（参见第141页，TODO）。

我们注意到的第二件事是，r所指向的值会从内存中重新加载100次。这是因为我们忘记告诉编译器假设没有指针别名（见79页，TODO）。添加编译器选项"assume no pointer aliasing"(如果有效)有可能改善代码。

第三个可以改进的是r+i/2可以通过归纳变量来计算因为它是循环索引的阶梯函数（staircase function）。整数除法将会防止编译器生成归纳变量，除非循环由2展开。(见页72)。

结论是，我们可以帮助编译器优化*示例8.26a*，方法是将循环由2展开，并创建一个显式归纳变量。(这消除了对前两个改进建议的需要)。

```
// Example 8.26b
void Func(int a[], int & r)
{
    int i;
    int Induction = r;
    for (i = 0; i < 100; i += 2)
    {
        a[i] = Induction;
        a[i+1] = Induction;
        Induction++;
    }
```

从**示例8.26b**编译器将生成以下汇编代码：

```
; Example 8.26b compiled to assembly:
ALIGN 4                                 ; align by 4
PUBLIC ?Func@@YAXQAHAAH@Z               ; mangled function name
?Func@@YAXQAHAAH@Z PROC NEAR            ; start of Func
; parameter 1: 4 + esp ; a
; parameter 2: 8 + esp ; r
$B1$1: ; unused label
    mov eax, DWORD PTR [esp+4]          ; eax = address of a
    mov edx, DWORD PTR [esp+8]          ; edx = address in r
    mov ecx, DWORD PTR [edx]            ; ecx = Induction
    lea edx, DWORD PTR [eax+400]        ; edx = point to end of a
$B2$2:                                  ; top of loop
    mov DWORD PTR [eax], ecx            ; a[i] = Induction;
    mov DWORD PTR [eax+4], ecx          ; a[i+1] = Induction;
    add ecx, 1                          ; Induction++;
    add eax, 8                          ; point to a[i+2]
    cmp edx, eax                        ; compare with end of array
    ja $B2$2                            ; jump to top of loop
$B2$3:                                  ; unused label
    ret                                 ; return from Func
    ALIGN 4
; mark_end;
```

这个解决方案显然更好。尽管它在一个循环中执行了两次迭代，但循环体现在只包含6条指令，而不是9条，。编译器用包含当前数组元素地址的第二个归纳变量（eax）来代替i。它没有将i与循环条件中的100进行比较，而是将数组指针eax与数组末端的地址进行比较，后者是它预先计算并存储在edx中。此外，这个解决方案减少了一个寄存器的使用，这样它就不必压入和弹出ebx。

9 优化内存访问

9.1 代码和数据缓存

高速缓存是计算机主存的代理。代理更小，更接近*CPU*，比主内存快，因此访问速度更快。可能有两级或者三级缓存，以便尽可能快地访问使用最多的数据。

*CPU*的速度比*RAM*内存的速度增长得快。因此，高效的缓存变得越来越重要。

9.2 缓存结构

如果你正在编写需要非顺序访问大数据结构的程序，并且希望防止缓存竞争，那么如果了解缓存是如何组织的，将会很有有用。如果你喜欢更多的启发式指导原则，可以跳过本节。

大多数缓存被组织成行和组。让我用一个例子来解释一下。我的示例是一个大小为8 kb、行大小为64字节的缓存。每行包含64个连续字节的内存。1K 字节是1024字节，所以我们可以计算出行数是`8*1024/64 = 128`。这些行按32组*4行的方式组织在一起。这意味着不能将特定的内存地址加载到任意的缓存行中。只能使用32组缓存中一个，但是在同一个组中的4行可以随意使用。我们可以通过公式：`(set) = (memory address) / (line size) % (number of sets)`，来计算特定内存地址使用缓存中的哪个组。在这里，`/`表示带截断的整数除法，`%`表示模。例如，如果我们想从内存地址`a = 10000`中读取数据，那么我们有`(set) =(10000 / 64) % 32 = 28$`。这意味着必须将a读入第28组中的4个缓存线之一。如果我们使用十六进制数，计算就会变得更简单，因为所有的数都是2的幂。使用十六进制数，我们得到`a = 0x2710`和`(set) = (0x2710 / 0x40) % 0x20 = 0x1C$`。从地址`0x2710`读取或写入变量将导致缓存将地址`0x2700`到`0x273F`的全部`64`或`0x40`字节加载到集合`0x1C`的4个缓存行之一。如果程序随后读取或写入该范围内的任何其他地址，那么该值已经在缓存中，因此我们不必等待另一个内存访问。

假设一个程序从地址`0x2710`读取数据，之后从地址`0x2F00`、`0x3700`、`0x3F00`和`0x4700`读取数据。这些地址都属于组`0x1C`。每组中只有4条缓存线。如果缓存总是选择最近使用最少的缓存线，那么当我们从`0x4700`读取数据时，将会覆盖地址范围从`0x2700`到`0x273F`的内容。再次读取地址`0x2710`将导致缓存不命中。但是，如果程序从具有不同设置值的不同地址读取，那么包含地址范围从`0x2700`到`0x273F`的行仍然在缓存中。问题只出现在地址之间的间隔是`0x800`的倍数的情况下。我把这段距离称为关键步。内存中的距离是关键步的倍数的变量将争夺相同的缓存线。关键步长可以这样计算：`(critical stride) = (number of sets) * (line size) = (total cache size) / (number of ways)`。

如果一个程序包含许多分散在内存中的变量和对象，那么就存在这样一种风险，即多个变量恰好被多个关键步长隔开，从而在数据缓存中引起竞争。如果程序内存中有许多分散的函数，那么在代码缓存中也会发生同样的情况。如果在程序的同一部分中使用的几个函数恰好被多个关键步间隔，那么这可能会在代码缓存中引起竞争。接下来的章节将描述各种避免这些问题的方法。

关于缓存如何工作的更多细节可以在`Wikipedia`的词条：[**CPU缓存**]((en.wikipedia.org/wiki/L2_cache)中找到。

手册3:“The microarchitecture of Intel, AMD and VIA CPUs”涵盖了不同处理器的缓存组织细节。

9.3 一起使用的函数应该被放在一起

如果在代码内存中使用的函数彼此接近，那么代码缓存的工作效率最高。函数通常按照它们在源代码中出现的顺序存储。因此，最好将代码中最关键部分中使用的函数集中在同一个源文件中，这些函数彼此相邻。将经常使用的函数与很少使用的函数分开，并将很少使用的分支（如错误处理）放在函数的末尾或单独的函数中。

有时，为了模块化，函数被保存在不同的源文件中。例如，在一个源文件中有父类的成员函数，在另一个源文件中有派生类的成员函数，这样做可能比较方便。如果父类和派生类的成员函数是在程序的相同关键部分被调用的，那么在程序内存中保持这两个模块的连续是有利的。这可以通过控制模块链接的顺序来实现。链接顺序通常是模块在项目窗口或`makefile`中出现的顺序。你可以通过向链接器请求映射文件来检查内存中函数的顺序。映射文件告诉每个函数相对于程序开始的地址。映射文件包含从静态库链接（*.lib*或*.a*）的库函数的地址，但不是动态库（*.dll*或*.so*）。没有一种简单的方法可以控制动态链接库函数的地址。

9.4 一起使用的变量应该被放在一起

缓存不命中的代码是非常高昂的。从缓存中加载一个变量只需要几个时钟周期，但是如果变量不在缓存中，那么将需要耗费超过100个时钟周期的是从*RAM*加载它。

如果一起使用的数据片段在内存中彼此靠近存储，则缓存的工作效率最高。变量和对象最好在使用它们的函数中声明。这些变量和对象将存储在栈中，栈很可能在一级缓存中。第26页（TODO）解释了不同类型的变量存储。如果可能，避免全局变量和静态变量，并避免动态内存分配（`new`和`delete`）。

面向对象编程是一种将数据存储在一起的有效方法。类的数据成员（也称为属性）总是一起存储在类的对象中。父类和派生类的数据成员一起存储在派生类的对象中（参见第52页）。

如果代码中有大数据结构，那么存储数据的顺序可能非常重要。例如，如果一个程序有两个数组，a和b，并且元素的访问顺序是a[0]，b[0]，a[1]，b[1]，...，然后，你可以通过将数据组织为结构数组来提高性能：

```
// Example 9.1a
int Func(int);
const int size = 1024;
int a[size], b[size], i;
...
for (i = 0; i < size; i++)
{
    b[i] = Func(a[i]);
```

如果按如下方法组织数据，那么这个例子中的数据可以在内存中被按顺序访问：

```
// Example 9.1b
int Func(int);
const int size = 1024;
struct Sab {int a; int b;};
Sab ab[size];
int i;
...
for (i = 0; i < size; i++)
{
    ab[i].b = Func(ab[i].a);
```

将数据结构编程**例9.1b**中那样，那么程序代码中将不会有额外的开销。相反的，代码变的更加简单，因为只需要计算一个数组的地址，而不是两个。

一些编译器将为不同的数组使用不同的内存空间，即使它们从未同时被使用过。例如：

```
// Example 9.2a
void F1(int x[]);
void F2(float x[]);
void F3(bool y)
{
    if (y)
    {
        int a[1000];
        F1(a);
    }
    else
    {
        float b[1000];
        F2(b);
    }
```

在这里，可以为a和b使用相同的内存区域，因为它们的活动范围不重叠。通过将a和b放入union中，可以节省大量缓存空间：

```
// Example 9.2b
void F3(bool y)
{
    union
    {
        int a[1000];
```

```
        float b[1000];
    };
    if (y)
    {
        F1(a);
    }
    else
    {
        F2(b);
    }
```

当然，使用union不是一种安全的编程实践，因为如果a和b的使用重叠，编译器不会发出警告。您应该只对占用大量缓存空间的大型对象使用此方法。将简单变量放入union中不是最佳选择，因为它妨碍使用寄存器变量。

9.5 数据对齐

如果将变量存储在可被变量大小整除的内存地址中，则访问该变量的效率最高。例如，double占用8字节的存储空间。因此，最好将其存储在可被8整除的地址中。大小应该总是2的幂。大于16字节的对象应该存储在可被16整除的地址中。您通常可以假设编译器会自动处理这种对齐。

结构和类成员的对齐可能会造成缓存空间的浪费，如**示例7.39**（TODO）中所述。

您可以选择按缓存线大小(通常是)对齐大型对象和数组64字节。这可以确保对象或数组的开头与缓存线的开头一致。一些编译器会自动对齐大的静态数组，但你也可以通过以下方式显示指定：

```
__declspec(align(64)) int BigArray[1024]; // Windows syntax
```

或

```
int BigArray[1024] __attribute__((aligned(64))); // Linux syntax
```

关于动态分配内存对齐的讨论参见第97页和第123页（TODO）。

9.6 动态分配内存

对象和数组可以通过new和delete或malloc和free动态分配。当编译时不知道所需的内存大小时，这可能非常有用。下面是动态内存分配的四种典型用法：

1. 可以在编译时不知道数组大小的情况下动态分配大数组。
2. 当编译时不知道对象总数时，可以动态分配可变数量的对象。
3. 可以动态分配文本字符串和类似大小可变对象。
4. 对于栈来说太大的数组可以动态分配。

动态分配内存的优点有：

1. 在某些情况下提供了更清晰的程序结构。
2. 不会分配超过所需的空间。这使得数据缓存比当一个固定大小的数组非常大, 以覆盖最坏的情况下最大可能的内存要求的情况时更高效。
3. 当不能预先给出所需内存空间的合理上限时，这是非常有用的。

动态分配内存的缺点有：

1. 动态分配和释放内存的过程比其他类型的存储需要更多的时间。见26页（TODO）。
2. 当以随机顺序分配和释放不同大小的对象时，堆空间就会变得碎片化。这使得数据缓存效率低下。
3. 如果已分配的数组已满，则可能需要调整其大小。这可能需要分配一个新的更大的内存块，并将整个内容复制到新块中。指向旧块中的数据的任何指针都将失效。
4. 当堆空间变得太碎片化时，堆管理器将启动垃圾收集。此垃圾收集可能在不可预测的时间开始，并在用户等待响应的不方便的时间导致程序执行的延迟。
5. 程序员有责任确保已分配的所有内容也被释放。如果不这样做，将导致堆被填满。这是一种常见的编程错误，称为内存泄漏。
6. 序员有责任确保在释放对象之后没有对象被访问。没有这么做也是一个常见的编程错误。
7. 所分配的内存可能不是最佳对齐的。有关如何对齐动态分配的内存，请参见第123页（TODO)。
8. 编译器很难优化使用指针的代码，因为它不能排除别名（参见第79页，TODO）。
9. 当行长度在编译时未知时，矩阵或多维数组的效率较低，因为在每次访问时需要额外的工作来计算行地址。编译器可能无法使用归纳变量对其进行优化。

在决定是否使用动态内存分配时，权衡利弊是很重要的。当数组的大小或对象的数量在编译时已知或可以知道合理的上限时，没有理由使用动态内存分配。

当分配的数量有限时，动态内存分配的成本可以忽略不计。因此，当一个程序有一个或几个可变大小的数组时，动态内存分配是有利的。另一种解决方案是将数组设置得非常大，以覆盖最坏的情况，这是在浪费缓存空间。如果一个程序有几个大数组，并且每个数组的大小是关键步长（参见上面第89页，TODO）的倍数，那么很可能会在数据缓存中引起竞争。

如果一个数组中的元素数量在程序执行期间增长，那么最好从一开始就分配最终的数组大小，而不是一步一步地分配更多的空间。在大多数系统中，您无法增加已经分配的内存块的大小。如果最终大小无法预测，或者预测结果太小，那么就需要分配一个新的更大内存块，并将旧内存块的内容复制到新的更大内存块的开头。当然，这是低效的，并且会导致堆空间变得碎片化。另一种方法是保留多个内存块，要么以链表的形式，要么以内存块的索引的形式。具有多个内存块的方法使得对单个数组元素的访问更加复杂和耗时。

一个可变数量的对象集合通常被实现为一个链表。链表中的每个元素都有自己的内存块和指向下一个块的指针。链表的效率不如线性数组，原因如下：

1. 每个对象都是单独分配的。分配、释放和垃圾收集需要大量的时间。
2. 对象没有连续地存储在内存中。这会降低数据缓存的效率。
3. 额外的内存空间用于链接指针和堆管理器为每个分配的块存储的信息。
4. 遍历链表比遍历线性数组要花费更多的时间。在加载前一个元素指针之前，不能加载任何指针。这就形成了一个关键的依赖链，这会阻止无序执行。

为所有对象分配一个大内存块（内存池）通常比为每个对象分配一个小内存块效率更高。

使用new和delete分配可变大小的数组的一个鲜为人知的替代方法是使用alloca分配来代替。这是一个在栈上而不是堆上分配内存的函数。内存空间在当从调用alloca的函数返回时会被自动释放。在使用alloca时，不需要显式地释放空间。与new和delete或malloc和free相比，alloca的优势有：

1. 分配过程的开销很小，因为微处理器有硬件支持对栈的操作。
2. 由于堆栈的先入后出特性，内存空间不会变得支离破碎。
3. 重新分配没有成本，因为它在函数返回时将自动执行。不需要垃圾收集。
4. 所分配的内存与栈上的其他对象是连续的，这使得数据缓存非常高效。

下面的例子将展示如何适应alloca分配可变大小的数组：

```
#include <malloc.h>
void SomeFunction (int n)
{
    if (n > 0)
    {
    // Make dynamic array of n floats:
        float * DynamicArray = (float *)alloca(n * sizeof(float));
    // (Some compilers use the name _alloca)
        for (int i = 0; i < n; i++)
        {
            DynamicArray[i] = WhateverFunction(i);
            // ...
        }
    }
```

显然，函数不应该返回任何使用alloca分配的指针或引用，因为它在函数返回时被释放。alloca可能与结构化异常处理不兼容。有关使用alloca的限制，请参阅编译器手册。

C99扩展支持可变大小的数组。这个特性是有争议的，并且只在**C**中可用，而不能在**C++**中使用。你可以使用alloca而不是可变大小的数组，因为它提供了相同的功能。

9.7 容器类

每当使用动态内存分配时，建议将分配的内存包装到容器类中。容器类必须具有析构函数，以确保所分配的所有内容也被释放。这是防止内存泄漏和与动态内存分配相关的其他常见编程错误的最佳方法。

容器类还可以方便地向数组中添加边界检查，以及使用先进的数据结构，如先进先出（或先进后出)访问、排序和搜索工具、二叉树、哈希映射等。

通常以模板的形式创建容器类，其中包含的对象类型作为模板参数提供。使用模板没有性能成本。

现成的容器类模板可用于许多不同的用途。最常用的容器集是标准模板库（**STL**），它时大多数现代**C++编译器**自带的。使用现成容器的优点是你不必重新发明轮子。**STL**中的容器是通用的、灵活的、经过良好测试的，对于许多不同的用途都非常有用。

然而，**STL**是以通用性和灵活性为准则而设计，而执行速度、内存经济性、缓存效率和代码大小的优先级较低。特别是在**STL**中，内存分配存在不必要的浪费。一些**STL**模板，如`list`、`set`和`map`，甚至可能分配比容器中对象更大的内存块。**STL** `deque`（双向链表）为每四个对象分配一个内存块。**STL** `vector`将所有的对象都存储在同一个内存块中，当这快内存被填满时会重新分配，这种情况经常发生，因为块大小每次只增长50%或更少。在一个实验中，10个元素被一个接一个地插入**STL** `vector`中，结果导致内存一共重新分配了7次，大小分别是 1、2、3、4、5、6和13（*MS Visual Studio 2008 version*）。可以在将第一个对象添加到`vector`之前，可以通过调用`vector::reserve`预先分配预测或估计的最终大小的内存来防止这种浪费的行为。其他**STL**容器没有预先分配内存的功能。

频繁地使用`new`和`delete`（或`malloc`和`free`）分配和释放内存会导致内存碎片化和缓存效率低下。如上所述，这会导致内存管理和垃圾收集的开销很大。

STL的通用性还会影响代码的大小。实际上**STL**因代码膨胀和复杂性而饱受批评（en.wikipedia.org/wiki/Standard_Template_Library）。存储在**STL**容器中的对象允许具有构造函数和析构函数。每次移动对象时都会调用每个对象的复制构造函数和析构函数，而这种情况经常发生。如果存储的对象本身就是容器，那么这是必要的。但是在**STL**使用一个或多个`vector`中实现一个矩阵，就像我们经常看到的那样，这肯定是一个非常低效的解决方案。

许多容器使用链表。链表是使容器可扩展的一种简便方法，但效率非常低。在大多数情况下，线性数组比链表快。

STL中用于访问容器元素的所谓的迭代器对于许多程序员来说很麻烦，如果你可以使用带有简单索引的线性列表，那么它们就不是必需的。一个好的编译器可以在某些情况下优化掉迭代器的额外开销（但不是全部）。

幸运的是，在执行速度、节约内存和代码大小比代码通用性具有更高优先级的情况下，可以使用更有效的替代方案。最重要的补救措施是内存池。将多个对象存储在一个大内存块中比将每个对象存储在它自己分配的内存块中更有效。如果没有复制构造函数和析构函数要调用，可以通过对`memcpy`的一次调用复制或移动包含许多对象的大块，而不是单独移动每个对象。

我实现了一组示例容器类，它们使用这些方法来提高效率。这些文件可以在www.agner.org/optimize/cppexamples.zip上作为本手册的附录获得，其中包含用于不同用途的容器类和模板。所有这些示例都针对执行速度和最小化内存碎片进行了优化。为了安全起见，包含了边界检查，但是如果出于性能原因需要，可以在调试之后删除。在**STL**的性能不令人满意的情况下，使用这些示例容器。

在为特定用途选择容器时，应考虑以下因素：

1. 包含一个还是多个元素？如果容器包含一个元素，那么可以使用智能指针（见第38页）。
2. 编译时是否知道大小？如果在编译时已知元素的数量，或者可以设置不太大的上限，那么最优解决方案是一个固定大小的数组或容器，而不需要动态内存分配。但是，如果数组或容器对于栈来说太大的时候，则可能需要动态内存分配。
3. 在存储第一个元素之前，大小是否已知？如果在存储第一个元素之前可以知道元素的总数（或者有一个合理的估计），那么最好使用允许预先分配（`reserve`）内存的容器,而不是分段分配内存或当内存块太小的时候重新分配。
4. 对象是连续编号的么？如果对象是由连续的索引或有限范围内的键标识的，那么简单的数组是高效的解决方案。
5. 对象是以先进先出的方式访问的么？如果在先进先出（**FIFO**）的基础上访问对象，则使用队列。将队列作为循环缓冲区而不是链表使用更高效。
6. 对象是以先进后出的方式访问的么？如果对象是在先入后出（**FILO**）的基础上访问的，那么使用带有栈顶部索引的线性数组（how？）。
7. 对象是由键标识的么？如果键值被限制在一个狭窄的范围内，那么可以使用一个简单的数组。如果对象的数量很多，那么最高效的解决方案可能是二叉树或哈希图。
8. 对象有顺序吗？如果你需要做这样的搜素："离元素 x 最近的是哪个？"或者 "在 x 和 y之间有多少个元素？"，那么你可以使用有序列表或者二叉树。
9. 添加所有对象之后是否需要搜索？如果需要搜索工具，但必须在容器中存储了所有对象之后，那么线性数组将是一个高效的解决方案。在添加所有元素之后对数组进行排序，然后使用二分搜索来查找元素。哈希表也可能是一种高效的解决方案。
10. 添加所有对象之前是否需要搜索？如需要搜索工具，并且可以随时添加新对象，那么解决方案就更复杂了。如果元素的总数很小，那么有序列表是最高效的解决方案，因为它的简单。但是如果列表很大，有序列表会非常低效，因为在列表中插入一个新元素会导致所有后续元素都需要移动。在这种情况下我们需要二叉树或者哈希表。如果元素是有序的，并且在一定间隔后就会有搜素请求，那么可以使用二叉树。哈希表则可以在元素没有特定顺序但又唯一的键标识时使用。
11. 对象是否具有混合类型或大小？可以在同一个内存池中存储不同类型的对象或不同长度的字符串。见 www.agner.org/optimize/cppexamples.zip。如果在编译时知道元素的数量和类型，那么就不需要使用容器或内存池。
12. 是否要对齐？一些应用程序要求数据按可以被整除的地址对齐。特别是使用向量指令时，需要对齐的地址可以被16整出。在某些情况下，将数据结构对齐到可被缓存线大小整除的地址（通常为64）可以提高性能。
13. 是否使用多线程？如果多个线程可以同时添加、删除或修改对象，那么容器类通常不是线程安全的。在多线程应用程序中，为每个线程设置单独的容器要比临时锁定一个容器以供每个线程独占访问高效的多。
14. 有指向包含的对象的指针么？将指针指向包含的对象可能是不安全的，因为容器可能在需要重新分配内存时移动对象。容器内的对象应该通过其在容器中的索引或键来标识，而不是通过指针或引用。但是，如果没有其他线程访问容器，则可以将指向此类对象的指针或引用传递给不添加或删除任何对象的函数。
15. 容器可以被回收么？创建和删除容器的消耗很大。如果程序的逻辑允许，复用一个容器可能比删除它再重新创建一个更高效。

我在www.agner.org/optimize/cppexamples.zip中提供了几个合适的容器类模板示例。如果不需要标准模板库容器的通用性和灵活性，那么可以使用它们作为标准模板库（**STL**）的替代品。你可以编写自己的容器类，或者修改可用的容器类来满足特定的需求。

9.8 字符串

文本字符串通常具有在编译时不知道的可变长度。文本字符串在string、wstring或CString等类中的存储使用new和delete来在每次创建或修改字符串时分配一个新的内存块。如果一个程序创建或修改了很多字符串，这可能是非常低效的。

在大多数情况下，处理字符串最快的方法是使用老式**C**风格的字符数组。字符串可以通过**C**函数如strcpy、strcat、strlen、sprintf等进行操作。但是要注意，这些函数没有检查数组是否溢出。数组溢出会导致在程序的其他地方出现难以预测的错误，这些错误很难诊断。程序员有责任确保数组足够大，能够处理包括终止零在内的字符串，并在必要时进行溢出检查。在www.agner.org/optimize/asmlib.zip的asmlib库中提供了常用字符串函数的快速版本以及用于字符串搜索和解析的高效函数。

如果希望在不影响安全性的情况下提高速度，可以将所有字符串存储在内存池中，如上所述，在本手册的附录www.agner.org/optimize/cppexamples.zip中提供相关示例。

9.9 按顺序访问数据

当按顺序访问数据时，缓存的工作效率最高。当逆序访问数据时，它的工作效率略低，而当以随机方式访问数据时，它的工作效率则低得多。这对读取和写入数据都是适用的。

多维数组应该以在最内层循环中变化最后一个索引的方式进行访问。这反映了元素在内存中存储的顺序。例如：

```
// Example 9.4
const int NUMROWS = 100, NUMCOLUMNS = 100;
int matrix[NUMROWS][NUMCOLUMNS];
int row, column;
for (row = 0; row < NUMROWS; row++)
    for (column = 0; column < NUMCOLUMNS; column++)
```

不要交换这两个循环的顺序（除非在Fortran中，具有相反的存储顺序）。

9.10 在大数据结构中的缓存竞争

按顺序访问多维数组并不总是可能的。一些应用程序（例如，在线性代数中）需要其他访问模式。如果一个大矩阵中的行之间的距离恰好等于关键步长，就会导致严重的延迟，如第89页（TODO）所述。如果矩阵行的大小（以字节为单位）是2的高次幂，就会发生这种情况。

下面的例子说明了这一点。我的例子是一个对二次矩阵进行转置的函数，即每个元素矩阵[r][c]与元素矩阵[c][r]交换。

```
// Example 9.5a
const int SIZE = 64;// number of rows/columns in matrix
void transpose(double a[SIZE][SIZE])// function to transpose matrix
{
    // define a macro to swap two array elements:
    #define swapd(x,y) {temp=x; x=y; y=temp;}
    int r, c; double temp;
    for (r = 1; r < SIZE; r++)
    { // loop through rows
        for (c = 0; c < r; c++)
        {
            // loop columns below diagonal
            swapd(a[r][c], a[c][r]); // swap elements
        }
    }
}

void test ()
{
    __declspec(__align(64))     // align by cache line size
    double matrix[SIZE][SIZE]; // define matrix
```

```
    transpose(matrix);          // call transpose function
```

矩阵的转置和以对角线为轴做镜像是一样的。对角线以下的每个元素矩阵[r][c]在对角线以上的镜像位置与元素矩阵[c][r]交换。**示例9.5a**中的循环 **c** 从最左边的列到对角线。对角线上的元素保持不变。

这段代码的问题是，如果对角线以下的元素矩阵[r][c]是逐行访问的，那么对角线以上的镜像元素矩阵[c][r]是逐列访问的。

假设现在我们再奔腾4电脑上运行这段代码，矩阵的大小是64。电脑的一级缓存为 `8 kb = 8192 bytes`，4 路，行大小为64。每个缓存行可以保存8个double变量，每个变量的大小为8个字节。关键步长为 $8192/4 = 2048bytes = 4rows$。

让我们看看循环内部发生了什么，例如当`r = 28`时。我们从对角线以下的第28行取出元素，并将这些元素与对角线以上的第28列交换。第28行中的前8个元素共享同一缓存线。因为缓存线按行而不是按列缓存，这8个元素将在第28列中进入8个不同的缓存行中。每四个高速缓存线路属于同一组高速缓存。当我们操作到第28列中的16号元素时，缓存将收回该列中0号使用的缓存线。17号元素将覆盖1号元素使用的缓存线，18号元素将覆盖2号元素使用的缓存线，依此类推。这意味着当我们将第29列与第29行交换时，对角线以上使用的所有缓存线都被覆盖了。因为在我们需要下一个元素之前，它会被删除，每个缓存线必须重新加载8次。我已经通过使用不同矩阵大小的奔腾4上的**示例9.5a**来测量转置矩阵所需的时间来证实这一点。我的实验结果如下，时间单位是每个数组元素所需要的时钟周期。

Matrix Size	Total kilobytes	Time per element
63*63	31	11.6
64*64	32	16.4
65*65	33	11.8
127*127	126	12.2
128*128	128	17.4
129*129	130	14.4
511*511	2040	38.7
512*512	2048	230.7
513*513	2056	38.1

Table 9.1. Time for transposition of different size matrices, clock cycles per element.

从表中可以看出，当矩阵的大小是一级缓存大小的倍数时，转置矩阵要多花40%的时间。这是因为关键步长是矩阵行的倍数。由于无序执行机制可以预先加载数据，延迟比一级缓存从二级缓存中重新加载数据的时间少。

当竞争发生在二级缓存中时，这种效果更为显著。二级缓存$512kb$，8路。二级缓存的关键步长是$512kb/8 = 64kb$。这对应于$512*512$矩阵中的16行数据。我在*表9.1**中的实验结果表明，在二级缓存中发生竞争时，转置矩阵所需的时间是不发生竞争时的6倍。这种效果在二级缓存竞争中比在一级缓存竞争中强得多的原因是二级缓存一次不能预加载多行。

解决这个问题的一个简单方法是使矩阵中的行比需要的长，以避免关键步长是矩阵行大小的倍数。我试着让矩阵的大小为$512 * 520$，包含不使用最后8列。这消除了竞争，时间消耗减少到36个时钟周期。

在某些情况下，不可能向矩阵中添加未使用的列。例如，一个数学函数库应该能够有效地处理所有大小的矩阵。在这种情况下，一个有效的解决方案是将矩阵分成更小的正方形，一次处理一个正方形。这被称为**square blocking **或**tiling。示例9.5b**演示了这种技术：

```
// Example 9.5b
void transpose(double a[SIZE][SIZE])
{
    // Define macro to swap two elements:
    #define swapd(x,y) {temp=x; x=y; y=temp;}
    // Check if level-2 cache contentions will occur:
    if (SIZE > 256 && SIZE % 128 == 0)
    {
        // Cache contentions expected. Use square blocking:
        int r1, r2, c1, c2; double temp;
        // Define size of squares:
        const int TILESIZE = 8; // SIZE must be divisible by TILESIZE
        // Loop r1 and c1 for all squares:
```

```
        for (r1 = 0; r1 < SIZE; r1 += TILESIZE)
        {
            for (c1 = 0; c1 < r1; c1 += TILESIZE)
            {
            // Loop r2 and c2 for elements inside sqaure:
                for (r2 = r1; r2 < r1+TILESIZE; r2++)
                {
                    for (c2 = c1; c2 < c1+TILESIZE; c2++)
                    {
                        swapd(a[r2][c2],a[c2][r2]);
                    }
                }
            }
           // At the diagonal there is only half a square.
           // This triangle is handled separately:
            for (r2 = r1+1; r2 < r1+TILESIZE; r2++)
            {
                for (c2 = r1; c2 < r2; c2++)
                {
                    swapd(a[r2][c2],a[c2][r2]);
                }
            }
        }
    }
    else
    {
        // No cache contentions. Use simple method.
        // This is the code from example 9.5a:
        int r, c; double temp;
        for (r = 1; r < SIZE; r++)
        { // loop through rows
            for (c = 0; c < r; c++)
            {
                // loop columns below diagonal
                swapd(a[r][c], a[c][r]); // swap elements
            }
        }
    }
```

在我的实验中，使用这段代码，对于512*512的矩阵来说，每个元素消耗50个时钟周期。

二级缓存中竞争的代价是如此的昂贵，因此对它们采取措施非常重要。因此，你应该了解矩阵中列数为2的高次幂的情况。一级缓存中的竞争消耗较少。使用复杂的技术，可能不值得在一级缓存中使用像**square blocking**这么复杂的技术。

Squre blocking以及类似的技术在 S. Goedecker 和 A. Hoisie 2001年出版的 “Performance Optimization of Numerically Intensive Codes”一书中有更详细的描述。

9.11 显示缓存控制

具有**SSE**和**SSE2**指令集的微处理器具有某些指令，允许你操作数据缓存。这些指令可以从支持内部函数（如**Microsoft**、**Intel**和**Gnu**）的编译器中访问。其他编译器需要通过汇编代码来访问这些指令。

Function	Assembly name	Intrinsic function name	Instruction set
Prefetch	PREFETCH	_mm_prefetch	*SSE*
Store 4 bytes without cache	MOVNTI	_mm_stream_si32	*SSE2*
Store 8 bytes without cache	MOVNTQ	_mm_stream_pi	*SSE*
Store 16 bytes without cache	MOVNTPS	_mm_stream_ps	*SSE*

Function	Assembly name	Intrinsic function name	Instruction set
Store 16 bytes without cache	MOVNTPD	_mm_stream_pd	*SSE2*
Store 16 bytes without cache	MOVNTDQ	_mm_stream_si128	*SSE2*

Table 9.2. Cache control instructions.

除了**表9.2**中提到的指令外，还有其他缓存控制指令，如*fluse*和*fence*指令，但这些指令与优化几乎没有关系。

<u>预加载数据</u>

预取指令可用于预先加载我们希望稍后在程序流中使用的缓存线。然而，在我测试的所有示例中，这并没有提高执行速度。原因是由于无序执行和先进的预测机制，现代处理器能自动预取数据。现代微处理器能够在包含多个具有不同步长的数据流的有规律访问模式下自动预取数据。因此，如果按照固定步长有规律地访问书库，则不必显式地预取数据。

<u>未缓存内存存储（uncached memory store）</u>

未缓存写入（uncached write）比未缓存读取（uncached read）开销更大，因为写入会需要先读取。然后再写回整个缓存行。

所谓的非时序写指令（**MOVNT**）就是为了解决这个问题而设计的。这些指令直接写入内存，而无需加载缓存线。如果我们正在写入未缓存的内存中，并且我们不希望在缓存线被清除之前再次从相同的或附近的地址读取数据，那么这是非常有利的。不要在同一个内存区域同时使用非时序的写操作和普通的写操作或读操作。

非时序写指令不适合**示例9.5**，因为我们在相同的地址读和写，所以无论如何都会加载缓存线。如果我们修改**示例9.5**，使它只写，那么非时序写指令的效果就会很明显。下面的示例对一个矩阵进行置换，并将结果存储在不同的数组中。

```
// Example 9.6a
const int SIZE = 512; // number of rows and columns in matrix
// function to transpose and copy matrix
void TransposeCopy(double a[SIZE][SIZE], double b[SIZE][SIZE])
{
    int r, c;
    for (r = 0; r < SIZE; r++)
    {
        for (c = 0; c < SIZE; c++)
        {
            a[c][r] = b[r][c];
        }
    }
```

这个函数逐列写入矩阵a，而由于关键步长导致所有写入都在一级缓存和二级缓存中都需要加载新的缓存线。使用非时序写指令可以防止二级缓存为矩阵a的加载任何缓存线：

```
// Example 9.6b.
#include "xmmintrin.h" // header for intrinsic functions
// This function stores a double without loading a cache line:
static inline void StoreNTD(double * dest, double const & source)
{
    _mm_stream_pi((__m64*)dest, *(__m64*)&source); // MOVNTQ
    _mm_empty(); // EMMS
}
const int SIZE = 512; // number of rows and columns in matrix
// function to transpose and copy matrix
void TransposeCopy(double a[SIZE][SIZE], double b[SIZE][SIZE])
{
    int r, c;
    for (r = 0; r < SIZE; r++)
    {
        for (c = 0; c < SIZE; c++)
```

```
        {
            StoreNTD(&a[c][r], b[r][c]);
        }
    }
}
```

在奔腾4计算机上测量了不同矩阵大小下每个矩阵单元的执行时间。测量结果如下：

Matrix size	Time per element Example 9.6a	Time per element Example 9.6b
64*64	14.0	80.8
65*65	13.6	80.9
512*512	378.7	168.5
513*513	58.7	168.3

Table 9.3. Time for transposing and copying different size matrices, clock cycles per element.

如**表9.3**所示，当且仅当可能出现二级缓存命中失败时，不经过缓存存储数据的方法才是有利的。$64*64$的矩阵大小会导致一级缓存中出现命中失败，但这对总执行时间几乎没有任何影响，因为一个存储操作上的缓存命中失败不会延迟后续指令。*512* 512$d的矩阵大小导致二级缓存中出现命中失败。这对执行时间有非常显著的影响，因为内存总线已经饱和。这是可以通过使用非时序写来改善。如果缓存竞争可以通过其他方式防止，如**第9.10节**所述，那么非时序写指令就不是最优的。

使用**表9.2**中所列的指令有一定的限制。所有这些指令都要求微处理器具有表中所列的**SSE**或**SSE2**指令集。16字节指令MOVNTPS、MOVNTPD和MOVNTDQ要求操作系统支持**XMM**寄存器；参见125页（TODO）。

当使用`#pragma vector nontemporal`时，**Intel编译器**可以在向量化代码中自动插入非时态写操作。然而，在**例9.6b**中这并不适用。

在任何浮点指令之前，MOVNTQ指令必须后跟EMMS指令。代码为`_mm_empty()`，如**示例9.6b**所示。MOVNTQ指令不能在64位*Windows*设备驱动程序中使用。

10 多线程

CPU的时钟频率受到物理因素的限制。在时钟频率有限的情况下，提高**CPU**密集型程序的吞吐量的方法是同时做多个事情。有三种方法可以并行地执行任务：

1. 使用多个**CPU**或多核**CPU**，如本章所述。
2. 使用现代**CPU**的乱序执行能力，如第11章所述。
3. 使用现代**CPU**的向量操作，如第12章所述。

多数现代**CPU**都拥有两个或更多个核心，可以预期的是，在未来核心的数量还会继续增加。为了使用多个**CPU**或者多个**CPU**核心，我们需要将任务划分到不同的线程。这里有两个主要的方法：功能分解和数据分解。功能分解意味着不同的线程做不同的工作。例如，一个线程处理用户界面，另一个线程处理和远程数据库的通信，第三个线程处理数学计算。将用户界面和耗时任务放在不同的线程中是很重要的，否则响应时间会变的长且规则，这是很令人讨厌的。将耗时的任务放在低优先级的单独线程中通常是很有帮助的。

然而，在许多情况下，一个任务就消耗了大部分资源。在这种情况下，我们需要将数据分割成多个块，以便利用多个处理器内核。然后每个线程应该处理自己的数据块。这就是数据分解。

在决定并行处理是否有利时，区分粗粒度并行和细粒度并行非常重要。粗粒度并行是指长序列的操作可以独立于并行运行的其他任务的情况。细粒度并行是指任务被划分为许多小的子任务，但是在与其他子任务进行必要的协调之前，不可能在特定的子任务上工作很长时间。

由于不同内核之间的通信和同步比较慢，因此粗粒度并行比使用细粒度并行效率更高。如果粒度太细，那么将任务拆分为多个线程是没有优势的。无序执行（**第11章**）和向量操作（**第12章**）是利用细粒度并行的更有用的方法。

使用多个**CPU**内核的方法是将工作划分为多个线程。第61页（TODO）讨论了线程的使用。在数据分解的情况下，我们最好不要有比系统中可用的内核或逻辑处理器数量更多的具有相同优先级的线程。可用逻辑处理器的数量可以通过系统函数获得（例如**Windows**中的`GetProcessAffinityMask`）。

有几种方法可以在多个**CPU**内核之间划分工作负载：

1. 定义多个线程，并在每个线程中投入等量的工作。此方法适用于所有编译器。

2. 使用自动并行化。**Gnu**、**Intel**和**PathScale**编译器可以自动检测代码中的并行化机会，并将其划分为多个线程，但编译器可能无法找到数据的最佳分解方案。
3. 使用**OpenMP**指令。**OpenMP**是**C++**和**Fortran**中定义并行处理的标准。这些指令被**Microsoft**、**Intel**、**PathScale**和**Gnu**编译器所支持。有关详细信息，请参见www.openmp.org和编译器手册。
4. 使用具有内部使用多线程的函数库，例如 *Intel Math Kernel Library*。

多个**CPU**内核或逻辑处理器通常共享相同的缓存，至少在最后一级缓存中是这样，在某些情况下甚至共享相同的一级缓存。共享相同缓存的优点是线程之间的通信变得更快，并且线程可以共享相同的代码和只读数据。缺点是，如果线程使用不同的内存区域，缓存就会被填满，如果线程写入相同的内存区域，就会发生缓存竞争。

只读的数据可以在多个线程之间共享，而可以修改的数据应该被每个线程单独存储。让两个或多个线程写入同一缓存行是没有任何好处的，因为线程会使彼此的缓存无效，并造成较大的延迟。每个线程都有自己的堆栈。使数据特定于线程的最简单方法是在线程函数中声明它，使其为线程本地的，以便将其存储在堆栈中。或者，你可以为包含特定于线程的数据定义结构或类，并为每个线程创建一个实例。此结构或类应至少按缓存线大小进行对齐，以避免多个线程写入同一缓存线。在现代处理器上，缓存线大小通常为64个字节。在未来的处理器上，缓存线大小可能会更大（28或256字节）。

线程之间有很多通信和同步方法，如信号量、互斥量和消息系统。所有这些方法都很耗时。因此，应该对数据和资源进行组织，以便尽可能减少线程之间的必要通信。例如，如果多个线程共享相同的队列、列表、数据库或者其他数据结构，那么你可以考虑能否给每个线程分配自己的数据，当所有线程完成耗时的数据处理后，最后再合并多个数据。

如果在一个只有一个逻辑处理器的系统上运行多个线程，而这些线程会争夺相同的资源，那么这不是一种优势。但是将耗时计算放在一个优先级低于用户界面的单独线程中可能是一个好主意。将文件访问和网络访问放在不同的线程中也很有用，这样一个线程可以在另一个线程等待硬盘或网络响应时进行计算。

Intel提供了各种支持多线程软件的开发工具。参见**《Intel 技术期刊》**2007年第11卷第4期。

10.1 同步多线程技术

许多微处理器能够在每个内核中运行两个线程。例如，一个有4个内核的处理器可以同时运行8个线程。这个处理器有四个物理处理器，但有八个逻辑处理器。

“**超线程**”是 **Intel** 对同步多线程的称呼。在同一个内核中运行的两个线程总是会争夺相同的资源，比如缓存和执行单元。如果有任何的共享资源是制约性能的因素，那么使用同步多线程没有任何优势。相反，由于缓存回收和其他资源冲突，每个线程的运行速度可能不到一半。但是，如果大部分时间存在缓存命中失败、分支错误预测或长依赖链，那么每个线程的运行速度将超过单线程速度的一半。在这种情况下，使用同步多线程有一点优势，但性能不会提高一倍。与另一个线程共享内核资源的线程总是比在内核中单独运行的线程运行得慢。

为了确定在特定的应用程序中使用同步多线程是否有利，常常需要进行测试。

如果并发多线程没有优势，那么有必要查询某些操作系统函数（例如**Windows**中的`GetLogicalProcessorInformation`），以确定处理器是否具有并发多线程。如果有，那么你可以通过只使用序号为偶数的逻辑处理器（0、2、4等）来避免同步多线程。旧的操作系统缺乏区分物理处理器数量和逻辑处理器数量的必要功能。

没有办法告诉处理器给一个线程比另一个线程更高的优先级。因此，低优先级线程经常会从运行在同一内核中的高优先级线程窃取资源。操作系统的责任是避免在同一个处理器内核中运行优先级相差很大的两个线程。不幸的是，当代的操作系统并不能很好地解决这个问题。

Intel编译器能够生成两个线程，其中一个线程用于为另一个线程预取数据。然而，在大多数情况下，自动硬件预加载比软件预加载效率更高。

乱序执行

除了一些小的低功耗**CPU** （如**Intel Atom**）之外，所有现代**x86 CPU**都可以无序地执行指令或同时执行多个操作。下面的示例展示了如何利用这种功能：

```
// Example 11.1a
float a, b, c, d, y;
```

这个表达式计算为`((a+b)+c)+d`。这是一个依赖链，每个加法都必须等待前一个的结果。你可以这样写来提高效率：

```
// Example 11.1b
float a, b, c, d, y;
```

两个括号可以独立的计算。在计算完(a+b)之前，**CPU**将开始计算(c+d)。这可以节省几个时钟周期。你不能假定优化编译器会自动将**示例11.1a**中的代码更改为**示例11.1b**，尽管这似乎是一件显而易见的事情。编译器不对浮点表达式进行这种优化的原因是，它可能会导致精度的损失，如第74页（TODO）所述。你必须手动添加。

当依赖链较长时，其影响更强。在循环中通常是这样。考虑下面的例子，它计算100个数字的和：

```
// Example 11.2a
const int size = 100;
float list[size], sum = 0; int i;
for (i = 0; i < size; i++)
```

这里有一个很长的依赖链。如果浮点加法需要5个时钟周期，那么这个循环大约需要500个时钟周期。通过展开循环并将依赖链一分为二，可以显著提高性能：

```
// Example 11.2b
const int size = 100;
float list[size], sum1 = 0, sum2 = 0; int i;
for (i = 0; i < size; i += 2)
{
    sum1 += list[i];
    sum2 += list[i+1];
}
```

如果微处理器从时间 T 到 T+5 对sum1做加法，那么它可以从时间 T+1 到 T+6 对 sum2 做加法，整个循环只需要256个时钟周期。

在循环中，每个迭代都需要前一个迭代的结果，这种循环中的计算称为循环依赖链。这样的依赖链可能非常长，并且非常耗时。如果这样的依赖链能够被打破，将会有很多收益。sum1和sum2这两个求和变量称为累加器。当前的**CPU**只有一个浮点加法单元，但是如上所述，这个单元是流水线操作的，因此它可以在前一个加法完成之前开始一个新的加法。

浮点加法和乘法的累加器的最佳数量可能是3个或4个，这取决于**CPU**。

如果循环的次数不能被展开因子整除，那么展开循环就会变得稍微复杂一些。例如，如果**示例11.2b**中 list 中的元素数量是奇数，那么我们必须在循环之外计算最后一个元素，或者向list中添加一个额外的伪元素，使这个额外的元素为零。

如果没有循环依赖链，则不需要展开循环并使用多个累加器。具有无序功能的微处理器可以重叠迭代，并在前一个迭代完成之前开始下一个迭代的计算。例如：

```
// Example 11.3
const int size = 100; int i;
float a[size], b[size], c[size];
float register temp;
for (i = 0; i < size; i++)
{
    temp = a[i] + b[i];
    c[i] = temp * temp;
```

具有无序功能的微处理器非常智能。他们可以检测到**示例11.3**中循环的一次迭代中的寄存器临时值独立于前一次迭代中的值。这允许它在计算完前一个值完成之前开始计算一个新的临时值。它通过为temp分配一个新的物理寄存器来实现这一点，即使在机器码中出现的逻辑寄存器是相同的。这叫做寄存器重命名。**CPU**可以保留同一逻辑寄存器的许多重命名实例。

这种优势是自动产生的。没有理由展开循环并使用temp1和temp2。现代**CPU**能够在满足某些条件的情况下重命名寄存器和并行执行多个计算。使**CPU**能够重叠循环迭代计算的条件为：

1. 没有循环依赖链。一次迭代的计算不应该依赖于前一次迭代的结果（循环计数器除外，当它是整数时，它的计算速度很快）。
2. 所有的中间结果都应该保存在寄存器中，而不是内存中。重命名机制只对寄存器有效，而对内存或缓存中的变量无效。在**示例11.3**中，即使没有register关键字，大多数编译器也会使temp成为寄存器变量。**CodeGear编译器**不能生成浮点寄存器变量，但会在内存中保存临时变量。这会阻止CPU的重叠计算。
3. 循环分支被应该可以预测。如果重复计数很大或恒定，则不存在此问题。如果循环计数很小且不断变化，那么CPU可能偶尔会预测循环分支已经退出了，而实际上它没有，因此无法开始下一个计算。然而，无序机制允许**CPU**提前增加循环计数器，这样它就可以在判断错误之前及

时发现。因此，你不必太担心这种情况。

通常，无序执行机制是自动工作的。但是，程序员可以做一些事情来最大限度地利用无序执行。最重要的是避免过长的依赖链。你可以做的另一件事是混合不同类型的操作，以便在**CPU**中的不同执行单元之间均匀地分配工作。只要不需要在整数和浮点数之间进行转换，就可以混合使用整数和浮点数计算。将浮点加法与浮点乘法混合使用、将简单整数与向量整数操作混合使用、将数学计算与内存访问混合使用也有很大的好处。

过长的依赖链会给**CPU**的无序资源带来了压力，即使它们没有进入循环的下一个迭代。一个现代的**CPU**通常可以处理100多个待定操作（参见手册3:"The microarchitecture of Intel, AMD and VIA CPUs"）。将循环分割并存储中间结果，对打破一个非常长的依赖链是有帮助的。

12 使用向量操作

现如今的微处理器拥有向量指令，使得同时对向量的所有元素进行操作成为可能。这也称为单指令多数据操作（**SIMD**）。每个向量的总大小可以是64位（**MMX**）、128位（**XMM**）、256位（**YMM**）和512位（**ZMM**）。

在大型数据集中，对多个数据元素执行相同操作且程序逻辑允许并行计算时，向量操作非常有用。例如图像处理、声音处理以及向量和矩阵的数学运算。本质上是串行的算法，比如大多数排序算法，不太适合向量操作。严重依赖于表查找或需要大量数据变换的算法（如许多加密算法）可能也不太适合向量操作。

向量操作使用一组特殊的向量寄存器。如果**SSE2**指令集可用，每个向量寄存器的最大大小为128位（**XMM**）；如果微处理器和操作系统支持**AVX**指令集，则为256位（**YMM**）；当**AVX512**指令集可用时为512位。每个向量中的元素数量取决于数据元素的大小和类型，如下所示：

Type of elements	Size of each elements, bits	Number of elements	Total size of vector, bits	Instruction set
`char`	8	8	64	*MMX*
`short int`	16	4	64	*MMX*
`int`	32	2	64	*MMX*
`int64_t`	64	1	64	*MMX*
`char`	8	16	128	*SSE2*
`short int`	16	8	128	*SSE2*
`int`	32	4	128	*SSE2*
`int64_t`	64	2	128	*SSE2*
`float`	32	4	128	*SSE*
`double`	64	2	128	*SSE2*
`char`	8	32	256	*AVX2*
`short int`	16	16	256	*AVX2*
`int`	32	8	256	*AVX2*
`int64_t`	64	4	256	*AVX2*
`float`	32	8	256	*AVX*
`double`	64	4	256	*AVX*
`char`	8	64	512	*AVX512BW*
`short int`	16	32	512	*AVX512BW*
`int`	32	16	512	*AVX512*
`int64_t`	64	8	512	*AVX512*
`float`	32	16	512	*AVX512*
`double`	64	8	512	*AVX512*

Table 12.1. Vector sizes available in different instruction set extensions

例如，当**SSE2**指令集可用时，一个128位的**XMM**寄存器可以组织为一个包含8个16位整数或4个浮点数的向量。应该避免使用64位宽的老旧**MMX**寄存器，因为它们不能与**x87**样式的浮点代码混合使用。

128位的**XMM**向量必须按照16对齐，即存储在一个可以被16整除的内存地址中（见下文）。256位的**YMM**向量最好按32对齐，而512位**ZMM**寄存器需要按64对齐，但是在编译**AVX**和以后的指令集时，对齐要求不那么严格。

在较新的处理器上，向量操作特别快。许多处理器可以像标量一样快速地计算向量（标量意味着单个元素）。支持新向量大小的第一代处理器的执行单元、内存端口等通常只有最大向量大小的一半。为了处理向量中所有元素，这些单元必须被使用两次。

数据元素越小，向量运算的使用就越有优势。例如，你可以同时进行四个`float`的加法，而对于`double`则只有两个。在现在**CPU**中，如果数据很符合向量寄存器，则使用向量运算几乎总是有利的。如果要将正确的数据放入正确的向量元素中，需要进行大量的数据操作，那么这这么做可能并没有什么优势。

12.1 AVX 指令集和 YMM 寄存器

128位的 **XMM** 寄存器在 **AVX** 指令集中被扩展为256位的 **YMM** 寄存器。**AVX** 指令集的主要优点是它允许更大的浮点向量。还有其他一些优势可能会在一定程度上提高性能。**AVX2** 指令集也允许256位整数向量。

为**AVX**指令集编译的代码只有在**CPU**和操作系统都支持**AVX**的情况下才能运行。在**Windows 7**、**Windows Server 2008 R2**和**Linux**内核2.6.30及以上版本中支持**AVX**。**Microsoft**、**Intel**、**Gnu**和**Clang**的最新编译器支持**AVX**指令集。

在某些英特尔处理器上混合使用和不使用**AVX**支持编译的代码时会出现问题。当从**AVX**代码转换到**非AVX**代码时，由于**YMM**寄存器状态的变化，会造成性能损失。应该在从**AVX**代码转换到**非AVX**代码之前调用内部函数`_mm256_zeroupper()`来避免这种损失。在以下情况下，这是必要的：

1. 如果程序的一部分是使用**AVX**支持编译，而另一部分未使用**AVX**支持编译时，那么在离开**AVX**部分之前调用`_mm256_zeroupper()`。
2. 如果一个函数使用**CPU**调度，在多个版本中使用或者不适用 **AVX** 支持编译，那么在离开**AVX** 部分之前调用`_mm256_zeroupper()`。
3. 如果使用了 **AVX** 支持编译的代码调用了编译器附带的库之外的库中的函数，而该库不支持**AVX**，则在调用库函数之前调用`_mm256_zeroupper()`。

12.2 AVX512 指令集和 ZMM 寄存器

256位的 **YMM** 寄存器在 **AVX512** 指令集中被扩展为512位的 **ZMM** 寄存器。在64位模式下，向量寄存器的数量从16个扩展到32个，而在32位模式下只有8个向量寄存器。因此，最好将**AVX512**代码编译为64位模式。

AVX512 指令集还添加了一组掩码寄存器。它们被用作布尔向量。几乎任何向量指令都可以用掩码寄存器进行掩码，这样，只有当掩码寄存器中的对应位为1时，才会计算向量元素。这使得使用分支的代码向量化效率更高。

AVX512 还有几个额外的扩展。所有支持 **AVX512** 的处理器都有一些这样的扩展，但是到目前为止还没有一个处理器拥有全部这些扩展（写于2016年）。下面是对 **AVX512** 的已有和计划的扩展：

1. **AVX512F**。基础扩展。所有支持 **AVX512** 的处理器都有这个扩展。包含在512位向量中对32位和64位整数，`float` 和 `double` 的操作以及掩码操作。
2. **AVX512VL**。包含在128和256位向量中的相同操作。包含掩码操作和32个向量寄存器。
3. **AVX512BW**。包含在512位向量中8位和16位整数的操作。
4. **AVX512DQ**。64位整数的乘法和转换指令。以及其它一些浮点和双精度指令。
5. **AVX512ER**。 快速倒数、倒数平方根、指数函数。对于 `float`类型是准确值，对于 `double` 类型则是近似值。
6. **AVX512CD**。冲突检测。找到向量中的重复元素。
7. **AVX512PF**。带聚集/分散逻辑的预取指令。
8. **AVX512VBMI**。 8位粒度的置换和移位指令。
9. **AVX512IFMA**。打包52位整数的乘法和加法。
10. **AVX512_4VNNIW**。Iterated dot product on 16-bit integers.
11. **AVX512_4FMAPS**。 Iterated fused multiply-and-add, single precision.

这使得CPU调度更加复杂。你可以选择对特定任务有用的扩展，并为具有此扩展的处理器创建代码分支。

在 **AVX512** 代码中，`_mm256_zeroupper()` 的使用不那么重要，但是仍然推荐使用。参见手册2:“Optimizing subroutines in assembly language”的13.2节和手册5:“Calling conventions”的6.3节。

12.3 自动向量化

好的编译器（如 **Gnu**、**Clang** 和 **Intel** 编译器)可以在并行性明显的情况下自动使用向量操作。有关详细说明，请参阅编译器文档。例如：

```
// Example 12.1a. Automatic vectorization
const int size = 1024;
int a[size], b[size];
// ...
for (int i = 0; i < size; i++)
{
    a[i] = b[i] + 2;
```

一个好的编译器会在指定 **SSE2** 或更高的指令集时使用向量操作来优化这个循环。根据使用指令集的不同，代码将读取4个，或8个，或16个 `b` 中的元素到一个向量寄存器中，与另一个向量寄存器包含（2,2,2,...）做加法，并将结果存储到 `a`中。此操作将被重复多次，次数为数组大小除以每个向量的元素数量。速度相应地提高了。循环计数能最好能被每个向量的元素数整除。你甚至可以在数组的末尾添加多余的元素，使数组大小成为向量大小的倍数。

当数组是通过指针访问的时候，这回有一个缺点，例如：

```
// Example 12.1b. Vectorization with alignment problem
void AddTwo(int * __restrict aa, int * __restrict bb)
{
    for (int i = 0; i < size; i++)
    {
        aa[i] = bb[i] + 2;
    }
```

如果数组按向量大小对对齐，**XMM**、**YMM** 和**ZMM** 寄存器分别为16、32或64，则性能最佳。在**AVX**之前，指令集下的高效的向量操作要求数组按可被16整除的地址排列。在**示例12.1a**中，编译器可以根据需要对数组进行对齐，但在**示例12.1b**中，编译器无法确定数组是否正确对齐。循环仍然可以向量化，但是代码的效率会降低，因为编译器必须对未对齐的数组采取额外的预防措施。当通过指针或引用访问数组时，你可以做许多事情来提高代码的效率：

1. 如果使用的是 **Intel** 编译器，可以使用 `#pragma vector aligned` 或 `_assume_aligned` 指令，告诉编译器数组是对齐的，并确保它们是对齐的。
2. 将函数声明为 `inline`。这使编译器可能将 **示例12.1b**简化为**12.1a**。
3. 如果可能的话，启用向量最大的指令集。**AVX** 和之后的指令集对对齐的限制很少，无论数组是否对齐，生成的代码都是高效的。

如果满足以下条件，自动向量化效果最好：

1. 使用支持自动向量化的编译器，如 **Gnu**、**Clang**、**Intel** 或 **PathScale**。
2. 使用编译器的最新版本。编译器在向量化方面变得越来越好。
3. 使用适当的编译器选项来启用所需的指令集（ `/arch:SSE2`， `/arch:AVX` 等用于 **Windows**， `-msse2`， `-mavx` 等用于 **Linux**）。
4. 使用限制较少的浮点选项。对于** Gnu** 编译器，使用 `-O3 -fnotrapping-math -fno-math-errno`。
5. 对于 **SSE2**，数组和大结构的地址按16对齐，对于 **AVX** 最好是32，而 **AVX512** 则最好使 64。
6. 循环计数最好是一个能被向量中的元素数整除的常数。
7. 如果数组是通过指针访问的，因此在你想要向量化的函数的范围内对齐是不可见的，那么请遵循上面给出的建议。
8. 如果数组或结构是通过指针或引用访问的，那么显式地告诉编译器指针没有别名 （如果合适的话）。有关如何做到这一点，请参阅编译器文档。
9. 在向量元素级别上最小化分支的使用。
10. 避免在向量元素级别使用查找表。

你可以查看汇编代码输出清单，以查看代码是否确实按预期被向量化（参见第86页，TODO）。

如果对连续变量序列执行相同的操作，则编译器还可以在没有循环的情况下使用向量操作。例如：

```
// Example 12.2
__declspec(align(16)) // Make all instances of S1 aligned

struct S1
{ // Structure of 4 floats
    float a, b, c, d;
};
void Func()
{
```

```
    S1 x, y;
    ...
    x.a = y.a + 1.;
    x.b = y.b + 2.;
    x.c = y.c + 3.;
    x.d = y.d + 4.;
};
```

4个浮点数的结构适合128位的 **XMM** 寄存器。在**示例12.2**中，优化后的代码将结构 `y` 加载到向量寄存器中，添加常量向量 `(1,2,3,4)`，并将结果存储在 `x` 中。

编译器并不总是能够正确地预测向量化是否有利。**Intel** 编译器允许你始终使用 `#pragma vector always` 来告诉编译器进行向量化，或者使用 `#pragma novector` 来告诉编译器不要向量化。必须将 **pragmas** 语句放在循环或希望它们应用于的一系列语句之前。

使用适合应用程序的最小数据大小是有利的。在**例12.3**中，例如，您可以通过使用 `short int` 代替 `int` 以得到2倍的速度。`short int` 是16位的， 而 `int` 是32位的，所以在相同的向量中，你可以存储8个 `short int`类型的数字，而只能存储4个 `int` 类型的数。因此，在不会产生溢出的情况下，使用足够大的最小位宽的类型类存储问题中的数字是有利的。同样地，如果代码可以向量化，那么使用 `float` 代替，`double` 是有好处的，因为 `float` 占用32位，而 `double` 占用64位。

SSE2 向量指令集不能对大小大于 `short int`（16位）的整数进行乘法。没有指令可以在像两种进行整数除法。但是 vector class library 和 asm lib有函数可以进行整数向量除法。

12.4 使用指令集函数

很难预测编译器是否会将循环向量化。下面的例子显示了编译器可以自动向量化，也可以不自动向量化的代码。代码中有一个分支，它为数组中的每个元素选择两个表达式：

```
// Example 12.4a. Loop with branch
// Loop with branch
void SelectAddMul(short int aa[], short int bb[], short int cc[])
{
    for (int i = 0; i < 256; i++)
    {
        aa[i] = (bb[i] > 0) ? (cc[i] + 2) : (bb[i] * cc[i]);
    }
```

可以使用所谓的指令集函数显式地向量化代码。当类似**示例12.4a**等当前编译器不会自动向量化代码的情况下非常有用或在自动向量化的代码不够优化的情况下，它也很有用。

指令集函数是一种基本操作，即每个指令集函数调用都被翻译成一个或几个机器指令。 **Gnu**、**Clang**、**Intel**、**Microsoft** 和 **PathScale** 编译器都支持指令集函数（**PGI**编译器也支持指令集函数，但效率很低，**Codeplay**编译器支持部分指令集函数，但是函数名与其他编译器不兼容）。使用 **Gnu**、**Clang** 和 **Intel** 编译器可以获得最佳的性能。

我们想对**例12.4a**中的循环进行向量化，这样我们就可以在包含8个16位整数的向量中同时处理8个元素。根据可用的指令集，循环内部的分支可以以多种方式实现。最兼容的方法是制作一个位掩码，当 `bb[i] > 0` 为真时全为1，当为假时全为0。将 `cc[i]+2`与上掩码，对掩码取反并与上 `bb[i]**cc[i]`。表达式的结果是与上全部是1的掩码结果不变，而与上全是0的掩码得到的结果为0。然后对这两个记过进行或操作就能得到要选择的表达式。

示例12.4b显示如何使用 **SSE2**指令集的函数实现上述步骤。

```
// Example 12.4b. Vectorized with SSE2
#include <emmintrin.h> // Define SSE2 intrinsic functions
// Function to load unaligned integer vector from array
static inline __m128i LoadVector(void const * p)
{
    return _mm_loadu_si128((__m128i const*)p);
}
// Function to store unaligned integer vector into array
static inline void StoreVector(void * d, __m128i const & x)
{
```

```
    _mm_storeu_si128((__m128i *)d, x);
}
// Branch/loop function vectorized:
void SelectAddMul(short int aa[], short int bb[], short int cc[])、
{
    // Make a vector of (0,0,0,0,0,0,0,0)
    __m128i zero = _mm_set1_epi16(0);
    // Make a vector of (2,2,2,2,2,2,2,2)
    __m128i two = _mm_set1_epi16(2);
    // Roll out loop by eight to fit the eight-element vectors:
    for (int i = 0; i < 256; i += 8)
    {
        // Load eight consecutive elements from bb into vector b:
        __m128i b = LoadVector(bb + i);
        // Load eight consecutive elements from cc into vector c:
        __m128i c = LoadVector(cc + i);
        // Add 2 to each element in vector c
        __m128i c2 = _mm_add_epi16(c, two);
        // Multiply b and c
        __m128i bc = _mm_mullo_epi16 (b, c);
        // Compare each element in b to 0 and generate a bit-mask:
        __m128i mask = _mm_cmpgt_epi16(b, zero);
        // AND each element in vector c2 with the bit-mask:
        c2 = _mm_and_si128(c2, mask);
        // AND each element in vector bc with the inverted bit-mask:
        bc = _mm_andnot_si128(mask, bc);
        // OR the results of the two AND operations:
        __m128i a = _mm_or_si128(c2, bc);
        // Store the result vector in eight consecutive elements in aa:
        StoreVector(aa + i, a);
    }
```

生成的代码将非常高效，因为它一次处理8个元素，并且避免了循环中的分支。**示例12.4b**的执行速度是**示例12.4a**的3到7倍，具体取决于循环中分支的可预测性。

`__m128i` 类型定义了一个包含整数的128位向量。它可以包含16个8位的整数，8个16位的整数，4个32位的整数，或者2个64位的整数。`__m128` 类型定义了一个包含4个 `float` 变量的128位向量。`__m128d` 类型定义了包含2个 `double`类型变量的128为变量。

指令集向量函数的名称以 `_mm` 开头。编译器手册或 **Intel** 的编程手册：“IA-32 Intel Architecture Software Developer’s Manual” 2A and 2B卷中列出了这些函数。指令集中有数百种不同的函数，很难找到适合特定用途的函数。

示例12.4b中笨拙的 *AND-OR* 结构可以被**SSE4.1** 指令集中的 `blend` 指令替换：

```
// Example 12.4c. Same example, vectorized with SSE4.1
// Function to load unaligned integer vector from array
static inline __m128i LoadVector(void const * p)
{
    return _mm_loadu_si128((__m128i const*)p);
}
// Function to store unaligned integer vector into array
static inline void StoreVector(void * d, __m128i const & x)
{
    _mm_storeu_si128((__m128i *)d, x);
}
void SelectAddMul(short int aa[], short int bb[], short int cc[])
{
    // Make a vector of (0,0,0,0,0,0,0,0)
    __m128i zero = _mm_set1_epi16(0);
    // Make a vector of (2,2,2,2,2,2,2,2)
    __m128i two = _mm_set1_epi16(2);
```

```
    // Roll out loop by eight to fit the eight-element vectors:
    for (int i = 0; i < 256; i += 8)
    {
        // Load eight consecutive elements from bb into vector b:
        __m128i b = LoadVector(bb + i);
        // Load eight consecutive elements from cc into vector c:
        __m128i c = LoadVector(cc + i);
        // Add 2 to each element in vector c
        __m128i c2 = _mm_add_epi16(c, two);
        // Multiply b and c
        __m128i bc = _mm_mullo_epi16 (b, c);
        // Compare each element in b to 0 and generate a bit-mask:
        __m128i mask = _mm_cmpgt_epi16(b, zero);
        // Use mask to choose between c2 and bc for each element
        __m128i a = _mm_blendv_epi8(bc, c2, mask);
        // Store the result vector in eight consecutive elements in aa:
        StoreVector(aa + i, a);
    }
```

你必须为要编译的指令集包含适当的头文件。头文件的名称如下：

Instruction set	** Header file**
MMX	mmintrin.h
SSE	xmmintrin.h
SSE2	emmintrin.h
SSE3	pmmintrin.h
Suppl. SSE3	tmmintrin.h
SSE4.1	smmintrin.h
SSE4.2	nmmintrin.h (MS) smmintrin.h (Gnu)
AES, PCLMUL	wmmintrin.h
AVX	immintrin.h
AMD SSE4A	ammintrin.h
AMD XOP	ammintrin.h (MS) xopintrin.h (Gnu)
AMD FMA4	fma4intrin.h (Gnu)
all	intrin.h (MS) x86intrin.h (Gnu)

Table 12.2. Header files for intrinsic functions

您必须确保 **CPU** 支持相应的指令集。如果您包含了高于 **CPU** 支持的指令集头文件，那么您就有可能插入** CPU** 不支持的指令，程序就会崩溃。有关如何检查支持的指令集，请参见第125页（TODO）。

<u>数据对齐</u>

如果数据的地址按可被向量大小（16或32字节）整除方式对齐，那么将数据加载到向量中会更快。这对旧的处理器和英特尔 **Atom** 处理器都有很大的影响，但在大多数较新的处理器上不是很重要。下面的例子展示了如何对齐数组。

```
// Example 12.5. Aligned arrays
// Define macro for aligning data
#ifdef _MSC_VER // If Microsoft compiler
```

```
#define Alignd(X) __declspec(align(16)) X
#else // Gnu compiler, etc.
#define Alignd(X) X __attribute__((aligned(16)))
#endif
const int size = 256; // Array size
Alignd ( short int aa[size] ); // Make three aligned arrays
Alignd ( short int bb[size] );
Alignd ( short int cc[size] );
// Function to load aligned integer vector from array
static inline __m128i LoadVectorA(void const * p)
{
    return _mm_load_si128((__m128i const*)p);
}
// Function to store aligned integer vector into array
static inline void StoreVectorA(void * d, __m128i const & x)
{
    _mm_store_si128((__m128i *)d, x);
```

<u>查找表向量化</u>

查找表对于优化代码非常有用，如第135页（TODO）所述。不幸的是，表查找常常是向量化的一个障碍。最新的指令集包括一些可用于向量化表查找的指令。这些说明总结如下。

Intrinsic function	Max. number of elements in table	Size of each table element	Number of simultaneous lookups	Instruction set needed
*mm*shuffle_epi8	16	1 byte = char	16	SSSE3
*mm*perm_epi8	32	1 byte = char	16	XOP, AMD only
*mm*permutevar_ps	4	4 bytes = float or int	4	AVX
*mm256*permutevar_ps	4	4 bytes = float or int	8	AVX2
*mm*i32gather_epi32	unlimited	4 bytes = int	4	AVX2
*mm256*i32gather_epi32	unlimited	4 bytes = int	8	AVX2
*mm*i64gather_epi32	unlimited	8 bytes = int64_t	2	AVX2
*mm256*i64gather_epi32	unlimited	8 bytes = int64_t	4	AVX2
*mm*i32gather_ps	unlimited	4 bytes = float	4	AVX2
*mm256*i32gather_ps	unlimited	4 bytes = float	8	AVX2
*mm*i64gather_pd	unlimited	8 bytes = double	2	AVX2
*mm256*i64gather_pd	unlimited	8 bytes = double	4	AVX2

Table 12.3. Intrinsic functions for vectorized table lookup

使用指令集函数可能会非常繁琐，代码会变得非常庞大，难以阅读。如下一节所述，使用向量类通常更简单一些。

12.5 使用向量类

用**示例12.4b**和**示例12.4c**中的方式编写程序确实很乏味。通过将这些向量操作包装到 C++ 类中，并使用重载的操作符（如添加向量），可以以更清晰易懂的方式编写相同的代码。操作符是内联的，因此生成的机器码与直接使用指令集函数时的机器码相同。只是编写 `a + b` 比编写 `_mm_add_epi16(a,b)` 更容易。

目前可以使用几种不同的预定义的向量类库，包括一个来自 **Intel**的，一个来自我的。我编写的向量类库（**VCL**）有许多特性，请参见www.agner.org/optimize/#vectorclass。**Intel vector class library** 最近没有更新，我觉得可能有些过时。

Vector class library	Intel	VCL (Agner)

Available from	Intel and Microsoft C++ compilers	VCL
Include file	dvec.h	vectorclass.h
Supported compilers	Intel, Microsoft	Intel, Microsoft, Gnu, Clang
Supported operating systems	Windows, Linux, Mac	Windows, Linux, Mac, BSD
Instruction set control	no	yes
License	license included in compiler price GNU General Public License, optional commercial license	

Table 12.4. Vector class libraries

下表列出了可用的向量类。包含适当的头文件将使你能够访问所有这些类。

Size of each element, bits	Number of elements in vector	Type of elements	Total size of vector, bits	Vector class, Intel	Vector class,VCL
8	8	char	64	Is8vec8	
8	8	unsigned char	64	Iu8vec8	
16	4	short int	64	Is16vec4	
16	4	unsigned short int	64	Iu16vec4	
32	2	int	64	Is32vec2	
32	2	unsigned int	64	Iu32vec2	
64	1	int64_t	64	I64vec1	
8	16	char	128	Is8vec16	Vec16c
8	16	unsigned char	128	Iu8vec16	Vec16uc
16	8	short int	128	Is16vec8	Vec8s
16	8	unsigned short int	128	Iu16vec8	Vec8us
32	4	int	128	Is32vec4	Vec4i
32	4	unsigned int	128	Iu32vec4	Vec4ui
64	2	int64_t	128	I64vec2	Vec2q
64	2	uint64_t	128		Vec2uq
8	32	char	256		Vec32c
8	32	unsigned char	256		Vec32uc
16	16	short int	256		Vec16s
16	16	unsigned short int	256		Vec16us
32	8	int	256		Vec8i
32	8	unsigned int	256		Vec8ui
64	4	int64_t	256		Vec4q
64	4	uint64_t	256		Vec4uq
32	16	int	512		Vec16i
32	16	unsigned int	512		Vec16ui

Size of each element, bits	Number of elements in vector	Type of elements	Total size of vector, bits	Vector class, Intel	Vector class,VCL
64	8	int64_t	512		Vec8q
64	8	uint64_t	512		Vec8uq
32	4	float	128	F32vec4	Vec4f
64	2	double	128	F64vec2	Vec2d
32	8	float	256	F32vec8	Vec8f
64	4	double	256	F64vec4	Vec4d
32	16	float	512		Vec16f
64	8	double	512		Vec8d

Table 12.5. Vector classes defined in two libraries

不建议使用总大小为64位的向量，因为它们与浮点数代码不兼容。如果使用64位向量，那么必须在64位向量操作之后和浮点代码之前执行调用 `_mm_empty()`。较大的向量没有这个问题。

只有在 **CPU** 和操作系统支持的情况下，256位和512位大小的向量才可用 （参见第109页，TODO）。我的**VCL**向量类库可以用两个128位向量模拟一个256位向量，或者将用两个256位向量或四个128位向量模拟一个512位向量。下面的示例展示了与**示例12.4b**相同功能的代码，使用 **Intel vector classes** 重写：

```
// Example 12.4d. Same example, using Intel vector classes
#include <dvec.h> // Define vector classes
// Function to load unaligned integer vector from array
static inline __m128i LoadVector(void const * p)
{
    return _mm_loadu_si128((__m128i const*)p);
}
// Function to store unaligned integer vector into array
static inline void StoreVector(void * d, __m128i const & x)
{
    _mm_storeu_si128((__m128i *)d, x);
}
void SelectAddMul(short int aa[], short int bb[], short int cc[])
{
    // Make a vector of (0,0,0,0,0,0,0,0)
    Is16vec8 zero(0,0,0,0,0,0,0,0);
    // Make a vector of (2,2,2,2,2,2,2,2)
    Is16vec8 two(2,2,2,2,2,2,2,2);
    // Roll out loop by eight to fit the eight-element vectors:
    for (int i = 0; i < 256; i += 8)
    {
        // Load eight consecutive elements from bb into vector b:
        Is16vec8 b = LoadVector(bb + i);
        // Load eight consecutive elements from cc into vector c:
        Is16vec8 c = LoadVector(cc + i);
        // result = b > 0 ? c + 2 : b * c;
        Is16vec8 a = select_gt(b, zero, c + two, b * c);
        // Store the result vector in eight consecutive elements in aa:
        StoreVector(aa + i, a);
    }
```

同样的例子使用我的**VCL**向量类是这样的：

```
// Example 12.4e. Same example, using VCL
```

```
#include "vectorclass.h" // Define vector classes
void SelectAddMul(short int aa[], short int bb[], short int cc[])
{
    // Define vector objects
    Vec16s a, b, c;
    // Roll out loop by eight to fit the eight-element vectors:
    for (int i = 0; i < 256; i += 16)
    {
        // Load eight consecutive elements from bb into vector b:
        b.load(bb+i);
        // Load eight consecutive elements from cc into vector c:
        c.load(cc+i);
        // result = b > 0 ? c + 2 : b * c;
        a = select(b > 0, c + 2, b * c);
        // Store the result vector in eight consecutive elements in aa:
        a.store(aa+i);
    }
```

由于对齐的问题，**Microsoft**编译器不允许将向量对象作为函数参数。建议使用常量引用：

```

// Example 12.6. Function with vector parameters
Vec4f polynomial (Vec4f const & x)
{
    // polynomial(x) = 2.5*x^2 - 8*x + 2
    return (2.5f * x - 8.0f) * x + 2.0f;
```

<u>使用向量类进行CPU调度</u>

VCL向量类库使从相同的源代码位不同的指令集编译代码成为可能。该库具有为给定指令集选择最佳实现的预处理指令。

下面的示例展示了如何使用自动CPU调度实现**示例(12.4e)**中的 `SelectAddMul` 。本例中的代码应该编译三次，一次使用 **SSE2** 指令集，一次使用 **SSE4.1** 指令集，一次用于 **AVX2** 指令集，所有三个版本都应该被链接到同一个可执行文件中。**SSE2** 是 **VCL** 支持的最旧指令集，**SSE4.1** 在 `select` 函数中具有优势，**AVX2** 指令集具有更大的向量寄存器的优势。当使用 **AVX2**指令集编译时，**VCL** 将会为 `Vec16s`分配一个256位向量寄存器来为，使用低版本的指令集时，则分配两个128位向量寄存器。预处理宏 **INSTRSET** 用于在使用不同指令集时赋予函数不同的名称。更多内容详见 vectorclass manual。

```
// Example 12.7. Vector class code with automatic CPU dispatching
#include "vectorclass.h" // vector class library
#include <stdio.h> // define fprintf
// define function type
typedef void FuncType(short int aa[], short int bb[], short int cc[]);
// function prototypes for each version
FuncType SelectAddMul, SelectAddMul_SSE2, SelectAddMul_SSE41,
SelectAddMul_AVX2, SelectAddMul_dispatch;
// Define function name depending on instruction set
#if INSTRSET == 2 // SSE2
    #define FUNCNAME SelectAddMul_SSE2
#elif INSTRSET == 5 // SSE4.1
    #define FUNCNAME SelectAddMul_SSE41
#elif INSTRSET == 8 // AVX2
    #define FUNCNAME SelectAddMul_AVX2
#endif
// specific version of the function. Compile once for each version
void FUNCNAME(short int aa[], short int bb[], short int cc[])
{
    Vec16s a, b, c; // Define biggest possible vector objects
    // Roll out loop by 16 to fit the biggest vectors:
```

```
    for (int i = 0; i < 256; i += 16)
    {
        b.load(bb+i);
        c.load(cc+i);
        a = select(b > 0, c + 2, b * c);
        a.store(aa+i);
    }
}
#if INSTRSET == 2
// make dispatcher in only the lowest of the compiled versions
#include "instrset_detect.cpp" // instrset_detect function
// Function pointer initially points to the dispatcher.
// After first call it points to the selected version
FuncType * SelectAddMul_pointer = &SelectAddMul_dispatch;
// Dispatcher
void SelectAddMul_dispatch(short int aa[], short int bb[], short int cc[])
{
    // Detect supported instruction set
    int iset = instrset_detect();
    // Set function pointer
    if (iset >= 8)
        SelectAddMul_pointer = &SelectAddMul_AVX2;
    else if (iset >= 5)
        SelectAddMul_pointer = &SelectAddMul_SSE41;
    else if (iset >= 2)
        SelectAddMul_pointer = &SelectAddMul_SSE2;
    else
    {
        // Error: lowest instruction set not supported
        fprintf(stderr, "\nError: Instruction set SSE2 not supported");
        return;
    }
    // continue in dispatched version
    return (*SelectAddMul_pointer)(aa, bb, cc);
}
// Entry to dispatched function call
inline void SelectAddMul(short int aa[], short int bb[], short int cc[])
{
    // go to dispatched version
    return (*SelectAddMul_pointer)(aa, bb, cc);
}
```

12.6 为向量化转换串行代码

并不是所有具有并行结构的代码都可以轻松地使用向量组织的。很多代码都是串行的，也就是说每个计算都依赖于前一个的结果。然而，如果代码是重复的，则可以以一种可被向量化的方式组织代码。最简单的情况是一长串数字的和：

```
// Example 12.8a. Sum of a list
float a[100];
float sum = 0;
for (int i = 0; i < 100; i++)
```

上述的代码时串行的，因为每次迭代 sum的值都依赖于前一次迭代后 sum的值。诀窍是将循环按 n 展开并重新组织代码，每个值依赖于 n 个位置之前的值，其中 n 是向量中元素的数量。如果 n = 4，我们得到：

```
// Example 12.8b. Sum of a list, rolled out by 4
float a[100];
float s0 = 0, s1 = 0, s2 = 0, s3 = 0, sum;
```

```
for (int i = 0; i < 100; i += 4)
{
    s0 += a[i];
    s1 += a[i+1];
    s2 += a[i+2];
    s3 += a[i+3];
}
```

现在，s0、s1、s2 和 s3可以组合成一个128位的向量，这样我们就可以在一个操作中做4个加法。如果我们指定 *fast math* x和SSE或更高指令集的选项，一个好的编译器会自动将**示例12.8a**转换为**12.8b**，并向量化代码。

一些更复杂的情况下不能自动向量化。例如，让我们看看泰勒级数的例子。指数函数可由级数计算：

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

用 **C++** 实现看起来可能是这样的：

```
// Example 12.9a. Taylor series
float Exp(float x)
{
    // Approximate exp(x) for small x
    float xn = x; // x^n
    float sum = 1.f; // sum, initialize to x^0/0!
    float nfac = 1.f; // n factorial
    for (int n = 1; n <= 16; n++)
    {
        sum += xn / nfac;
        xn *= x;
        nfac *= n+1;
    }
return sum;
```

在这里每个 x^n 的值由前一个值计算而来，即 $x^n = x*(x^{n-1})*，每个*n!*的值也由前一个值计算而来，即*n!*= *n*(n-1)!
。如果我们想要将循环按4展开，那我们必要用4个位置之前的值来计算当前的值。因此，我们将用x^4*x^{n-4}*来计算*x^n*
。没有简单的方法来展开阶乘的计算，但是这个并不是必需的，因为阶乘并不依赖*x$ ，我们可以将值预先计算好，存一个一个表中。更好的方法是存储阶乘的倒数，这样我们就不需要除法了（如你所知，除法是很慢的）。现在上述的的代码可以按如下的方式向量化（使用* *Intel vector classes**）：

```
// Example 12.9b. Taylor series, vectorized
#include <dvec.h> // Define vector classes (Intel)
#include <pmmintrin.h> // SSE3 required
// This function adds the elements of a vector, uses SSE3.
// (This is faster than the function add_horizontal)
static inline float add_elements(__m128 const & x)
{
    __m128 s;
    s = _mm_hadd_ps(x, x);
    s = _mm_hadd_ps(s, s);
    return _mm_cvtss_f32(s);
}
float Exp(float x)
{
    // Approximate exp(x) for small x
    __declspec(align(16)) // align table by 16
    const float coef[16] = { // table of 1/n!
    1., 1./2., 1./6., 1./24., 1./120., 1./720., 1./5040.,
    1./40320., 1./362880., 1./3628800., 1./39916800.,
    1./4.790016E8, 1./6.22702E9, 1./8.71782E10,
```

```
    1./1.30767E12, 1./2.09227E13};
    float x2 = x * x; // x^2
    float x4 = x2 * x2; // x^4
    // Define vectors of four floats
    F32vec4 xxn(x4, x2*x, x2, x); // x^1, x^2, x^3, x^4
    F32vec4 xx4(x4); // x^4
    F32vec4 s(0.f, 0.f, 0.f, 1.f); // initialize sum
    for (int i = 0; i < 16; i += 4)
    {
        // Loop by 4
        s += xxn * _mm_load_ps(coef+i); // s += x^n/n!
        xxn *= xx4; // next four x^n
    }
    return add_elements(s); // add the four sums
```

这个循环在一个向量中计算四个连续的项。如果循环很长，那么进一步展开循环可能是值得的，因为这里的速度可能受到 `xxn` 相乘的延迟而不是吞吐量的限制（参见第106页，TODO）。这里的系数表是在编译时计算的。在运行时计算表可能更方便，只要确保只表只被计算一次，而不是每次调用函数时都会被计算一次。

12. 7 支持向量的数学函数

有很多的函数库可以用于 计算向量的对数函数、指数函数、三角函数等数学函数。这些函数库对于向量化数学代码非常有用。

向量数学库有两种：长向量库（long vector library）和短向量库（short vector library）。为了解释它们之间的区别，假设你想相同的函数对一千个数进行计算。使用长向量库时，您将一个包含一千个数字的数组作为参数提供给库函数，该函数一千个结果存储在另一个数组中。使用长向量库的缺点是，如果要进行一长串计算，则必须在进行进一步计算前之前，必须每个步骤的中间结果存储在临时数组中。使用短向量库时，您可以将数据集划分为子向量，这些子向量与 **CPU** 中向量寄存器的大小相匹配。如果向量寄存器可以容纳4个数字，那么您必须调用库函数250次，每次将4个数字装入向量寄存器。库函数将在向量寄存器中返回结果，向量寄存器可以在计算序列中的下一个步骤直接使用，而不需要将中间结果存储在**RAM**内存中。尽管有额外的函数调用，但这可能会更快，因为 **CPU** 可以在预取下一个函数的代码的同时进行计算。然而，如果计算序列形成了长依赖链，使用短向量的方法可能会处于不利地位。我们希望 **CPU** 在完成对第一个子向量的计算之前开始对第二个子向量的计算。长依赖链可能会填满 **CPU** 中挂起的指令队列，并阻止其充分利用乱序执行的计算能力。

下面是一些长向量数学库的列表：

1. Intel vector math library (VML, MKL)。支持所有 **x86** 平台。这个库降低了**非英特尔CPU**的性能，除非您重写了英特尔的**CPU调度程序**。见134页（TODO）。
2. Intel Performance Primitives (IPP)。支持所有 **x86** 平台。在非英特尔的**CPU**上也能工作的很好。包括许多用于统计学，信号处理和图像处理的函数。
3. Yeppp。开源库。支持 **x86** 和**ARM**平台以及多种编程语言。www.yeppp.info。

下面是一些短向量数学库的列表：

1. Sleef library。支持多种不同的平台。开源。www.sleef.org。
2. Intel short vector math library (SVML)。这是由 **Intel编译器**提供的，并通过自动向量化调用。**Gnu编译器**可以通过选项`-mveclibabi=svml`使用这个库。如果不使用Intel编译器，这个库通常可以很好地处理 **非Intel CPU**。见134页（TODO）。
3. AMD LIBM library。只在64位 **Linux**和 **Windows**平台上可用。这个库在没有 **FMA4** 指令集的情况下降低了 **CPU** 的性能（这个指令集最初是由英特尔设计的，但目前只有 **AMD**的 **CPU**支持）。**Gnu编译器**可以通过选项 `-mveclibabi=acml` 使用这个库。
4. VCL vector class library。支持所有 **x86**平台。支持 **Microsoft**、**Intel**、**Gnu** 和 **Clang**编译器。代码是内联的，不需要链接外部库。www.agner.org/optimize/#vectorclass。

所有这些库都具有很好的性能和精度。速度比任何非向量库快很多倍。

SVML 和 **LIBM** 库中的函数名没有很好的文档说明。如果你想直接调用库函数，可以参考下表中的例子：

Library	exp function of 4 floats	exp function of 2 double
Intel SVML v.10.2 & earlier	`vmlsExp4`	`vmldExp2`
Intel SVML v.10.3 & later	`__svml_expf4`	`__svml_exp2`
Intel SVML + ia32intrin.h	`_mm_exp_ps`	`_mm_exp_pd`
AMD Core Math Library	`__vrs4_expf`	`__vrd2_exp`

Library	exp function of 4 floats	exp function of 2 double
AMD LIBM Library	`amd_vrs4_expf`	`amd_vrd2_exp`
VCL vector class library	`exp`	`exp`

12.8 对齐动态分配的内存

使用 `new` 或 `malloc` 分配的内存通常按8对齐，而不是按16对齐。当矢量运算需要按16对齐时，这就是个问题了。**Intel 编译器**通过定义 `_mm_malloc` 和 `_mm_free` 解决了这个问题。

更通用的方法是将分配的数组封装到容器类中，由容器类负责对齐。参见www.agner.org/optimize/cppexamples.zip了解如何使用向量访问使数组对齐。

12.9 对齐RGB视频或三维向量

RGB 图像数据每个点有三个值。这不适用于向量，例如四个浮点数。这同样适用于三维几何和其他奇数大小的向量数据。为了提高效率，数据必须按向量大小对齐。使用未对齐的读和写可能会降低执行速度，从而降低使用向量操作的优势。你可以选择以下一种解决方案，具体哪种方案最适合取决于你的算法：

1. 加入不使用的第四个值，使数据适合于向量。这是一个简单的解决方案，但是它增加了内存使用量。如果内存访问是瓶颈，需要避免使用这种方法。
2. 将四个（或八个）点的的数据组成一组，其中一个向量中有四个R值，下一个向量中有四个G值，最后一个向量中有四个B值。
3. 首先用所有的R值组织成数据，然后是所有的G值，最后是所有的B值。

选择哪种方法取决于哪种方法最适合所讨论的算法。您可以选择可以写出最简单代码的方法。

如果点的数量不能被向量大小整除，那么在最后面添加几个未使用的点，以得到整数个向量。

12.10 总结

如果代码天生具有并行性，那么使用向量可以大大提高速度。增益取决于每个向量的元素数。最简单和最干净的解决方案是依赖于编译器的自动向量化。在并行性明显且代码只包含简单标准操作的情况下，编译器将自动向量化代码。您所要做的就是启用适当的指令集和限制少的浮点选项。

然而，在许多情况下，编译器无法自动对代码进行向量化，或者以一种不太理想的方式进行向量化。在这里，您必须显式地向量化代码，有很多方法可以做到：

1. 使用汇编语言。
2. 使用指令集。
3. 使用预定义的向量类。

使用向量类是向量化代码的最简单方法。如果你需要的操作向量类库并没有提供，那么你可以通过使用向量类外加几个指令集函数来实现。无论您选择使用内部函数还是向量类，这只是是否方便的问题 — 在性能上没有区别。一个好的优化编译器应该在这两种情况下生成相同的代码。内部函数看起来笨拙而乏味。当您使用向量类和重载运算符时，代码将变得更具可读性。

好的编译器通常能够在手动向量化代码之后进一步优化代码。编译器可以使用函数内联、公共子表达式消除、常量传播、循环优化等优化技术。这些技术很少用于手动编写编码，因为它使代码变得笨拙、容易出错，而且难以维护。因此，在许多情况下，手动向量化与编译器的进一步优化相结合可以得到最好的结果。当前的编译器并不总是擅长于在向量化的代码上进行常量传播和某些其他优化技术。因此，在编译器可以自动向量化而没有问题的情况下，依赖于编译器的自动向量化会更好。为了找到最佳解决方案，可能需要进行一些试验。

向量化代码通常包含许多额外的指令，用于将数据转换为正确的格式，并将它们放到向量中的正确位置。这种数据转换和变换有时会比实际计算花费更多的时间。在决定使用向量化代码是否有利可图时，应该考虑到这一点。

我将通过总结决定矢量化有多有利的因素来结束本节。

使向量化有利的因素：

1. 小数据类型：`char`、`short`、`int`、`float`。
2. 大数组中对所有数据进行相似的操作。
3. 数组大小可以被向量大小整除。
4. 在不可预测的分支中选择两个简单表达式。
5. 只有向量操作数可用的操作：取最小值、取最大值、饱和加法、快速近似倒数、快速近似倒数平方根、RGB色差。
6. 向量指令集可用时，如：**AVX**、**AVX2**、**AVX-512**。

7. 数学向量函数库。
8. 使用 **GNU**、**Clang**、**Intel**编译器。

使向量化不那么有用的因素：

1. 大数据类型：`int64_t`、`double`。
2. 未对齐的数据。
3. 额外的数据转换：需要 *shuffling*、*packing*、*unpacking* 等操作。
4. 分支可预测时，在未选中时可以跳过大量表达式。
5. 编译器没有足够的指针对齐和别名信息。
6. 在指令集中缺少合适类型的向量操作，如 **SSE4.1** 之前的32位整数乘法和整数除法。
7. 执行单元小于向量寄存器大小的老 **CPU**.

对于程序员来说，向量化的代码更不易编写，因此更容易出错。因此，矢量化代码最好放在可重用且经过良好测试的库模块和头文件中。

13 为不同指令集生成多个版本的关键代码

微处理器制造商不断地向指令集中添加新的指令。这些新的指令可以使某些类型的代码执行得更快。对指令集最重要的补充是第12章中提到的向量运算。

如果代码是为特定的指令集编译的，那么它将与支持该指令集或任何更高指令集的所有 **CPU** 兼容，但可能不与更早的 **CPU** 兼容。向后兼容指令集的顺序如下：

Instruction set	Important features
80386	32 bit mode
SSE	128 bit float vectors
SSE2	128 bit integer and double vectors
SSE3	horizontal add, etc.
SSSE3	a few more integer vector instructions
SSE4.1	some more vector instructions
SSE4.2	string search instructions
AVX	256 bit float and double vectors
AVX2	256 bit integer vectors
FMA3	floating point multiply-and-add
AVX-512	512 bit integer and floating point vectors

Table 13.1. Instruction sets

手册4 "Instruction tables"提供了对指令集的更详细的说明。在混合使用 **AVX** 或更高版本编译的代码和不使用 **AVX** 编译的代码时会有一定的限制，如第109页（TODO）所解释的。

使用最新指令集的一个缺点是缺失了与旧微处理器的兼容性。这个难题可以在关键部分通过为不同的 **CPU** 使用多个版本的代码中来解决。这称为 **CPU调度**。例如，您可能希望创建一个利用 **AVX2** 指令集优势的版本，另一个只使用 **SSE2** 指令集的，以及一个而不使用任何这些指令集与旧微处理器兼容的通用版本。程序应该自动检测 **CPU** 支持哪个指令集。

13.1 CPU 调度策略

在开发、测试和维护方面，将一段代码转换成多个版本，每个版本都针对一组特定的c进行了仔细的优化和微调，这代价是相当大的。对于在多个应用程序中使用的通用函数库，这些代价是合理的，但这并不总是针对特定于应用程序的代码。如果您考虑使用 **CPU调度** 来生成高度优化的1代码，那么如果可能的话，最好以可重用库的形式来实现。这也使得测试和维护更加容易。

我对**CPU调度**做了大量的研究，发现很多常用的程序都使用了不合适的**CPU调度**方法。

CPU调度最常见的陷阱是：

1. 优化当前的处理器而不是未来的处理器。考虑使用 **CPU调度**开发和发布函数库所需的时间。在应用程序程序员获得库的新版本之前，还需要额外的时间。再加上开发和推广应用程序所需的时间。在加上最终用户获得应用程序的最新版本所需的时间。总而言之，您的代码要在大

多数最终用户的计算机上运行通常需要几年的时间。此时，您所优化的任何处理器都可能已经过时。程序员常常低估了这种时间延迟。

2. 考虑特定的处理器模型而不是处理器特性。程序员通常会考虑汕尾是"什么任务在处理器X上工作得最好?"而不是"什么任务在具有这个指令集的处理器上工作得最好?"。为每个处理器模型使用哪个代码分支的列表将会非常长，而且很难维护。最终用户也不太可能拥有最新的版本。**CPU调度器** 不应该查看 **CPU** 品牌和型号，而应该查看它具有哪些指令集和其他特性。
3. 假设处理器型号组成一个逻辑序列。如果您知道处理器型号N支持一个特定的指令集，那么您就不能假定型号N+1至少支持相同的指令集。一个数字更大的型号不一定是更新的。**CPU** 系列和型号并不总是连续的，对于未知CPU，你不能根据的系列和型号对其进行任何假设。
4. 无法正确处理未知处理器。许多 **CPU调度器**被设计成只处理已知的处理器。在编程时未知的其他品牌或型号通常会使用通用的代码分支，这是性能最差的分支。我们必须记住，许多用户更愿意在最新的 **CPU** 上运行速度关键型程序，而这个 **CPU** 在编程时很可能是未知的。**CPU调度器** 应该给一个未知品牌或型号的CPU最好的分支，如果它支持该分支兼容的指令集。"我们不支持处理器X"的常见借口在这里根本不合适。它揭示了 **CPU调度** 的根本缺陷。
5. 低估维持更新 **CPU调度程序**的成本。很容易为特定 CPU 型号调优代码，然后认为在新的 CPU 型号上市时你可以进行更新。但是，调优、测试、验证和维护一个新的代码分支的成本是如此之高，以至于在未来的许多年里，对每一个进入市场新的处理器都这么做是不现实的。即使是大型软件公司也常常无法使它们的 CPU 调度程序保持最新。一个更现实的目标是，只有当一个新的指令集出现，使显著地提升性能成为可能时，才创建一个新的分支。
6. 创建太多的代码分支。如果您正在创建针对特定 CPU 品牌或特定型号进行调优的分支，那么您很快就会得到许多占用缓存空间且难于维护的分支。你在特定 CPU 型号中处理的任何特定瓶颈或任何特别慢的指令在一两年内都可能不相关。通常，只要有两个分支就足够了：一个用于最新的指令集，另一个与最多5年或10年前的 CPU 兼容。CPU 市场发展如此之快，以至于今天全新的 CPU 将在明年成为主流。
7. 忽略虚拟化。CPUID 指令能够真正表示已知 CPU 型号的时代已经结束了。虚拟化变得越来越重要。虚拟处理器会减少内核的数量，以便为同一机器上的其他虚拟处理器保留资源。虚拟处理器可能会给出一个错误的型号来反映这一点，或者为了与一些遗留软件兼容。它甚至可能有一个错误的供应商字符串。在未来，我们可能还会看到仿真处理器和FPGA软核不对应于任何已知的硬件 CPU。这些虚拟处理器可以有任何品牌和型号。我们唯一可以依赖的 CPUID信息是特性信息，比如支持的指令集和缓存大小。

幸运的是，这些问题的解决方案在大多数情况下都非常简单：CPU调度器应该拥有尽可能少的分支，并且分派应该基于 CPU 支持的指令集，而不是 CPU 的品牌、系列和型号。

我见过许多 CPU调度很差劲的例子。例如，Mathcad 的最新版本（v15.0）使用的是Intel的 Math Kernel LIbrary 6年前的版本（MKL v7.2）。这个库有一个 CPU 调度器，它不能以最优的方式处理现在的 CPU。如果将CPUID 人为地更改为旧的 Pentium 4，在当前 Intel CPU 上，某些任务的速度可以提高 33% 以上。原因是 MKL 中的 CPU 调度程序依赖于 CPU 的族号（family number），旧 Pentium 4的CPU族号是15，而所有较新的Intel CPU的族号为6！当 CPUID 被操纵为假的 Intel Pentium 4时，在非Intel CPU 上执行这项任务的速度提高了一倍多。更糟糕的是，许多软件产品无法识别 VIA 处理器，因为这个品牌在软件开发时不太受欢迎。

对不同品牌 CPU 的不平等的 CPU 调度机制可能会成为一个严重的法律问题，正如您可以在我的博客中看到的那样。在这里，您还可以找到更多关于糟糕的 CPU 调度的例子。

显然，您应该只对程序的最关键部分使用 CPU 调度 —— 最好隔离到单独的函数库中。只有当指令集相互不兼容时，才能使用将整个程序转换成多个版本这样激进的解决方案。一个具有定义良好的功能和接口的函数库比一个把调度分支分散在源文件中的程序更容易管理和测试、维护和验证。

13.2 特定号的调度

在某些情况下，特定的代码实现在特定型号的处理器表现糟糕。您可以忽略这个问题，并假设在下一个处理器型号将表现更好。如果这个问题太重要而不能忽略，那么解决方案是为该版本代码表现的不好的处理器型号创建一个负面清单（negative list）。为该版本代码表现良好的处理器型号列一个可用清单（positive list）不是一个好主意。原因是，每当市场上出现新的、更好的处理器时，都需要更新可用清单。这样一个清单几乎肯定会在您的软件生命周期内被淘汰。另一方面，在下一代处理器表现更好的情况下，负面清单不需要更新。每当处理器有一个特定的弱点或瓶颈时，生产者很可能会试图修复这个问题，使下一个型号表现的更好。

请再次记住，大多数软件在大部分时间内都是在软件编码时未知的处理器上运行的。如果软件包含运行最高级代码版本的处理器型号的可用清单，那么它将在编程时未知的处理器上运行较差的版本。但是，如果软件包含一个负面清单，其中列出了避免运行高级版本的处理器模型，那么它将在编程时未知的所有较新的模型上运行高级版本。

13.3 棘手的例子

在大多数情况下，可以根据所支持的指令集、缓存大小等 CPUID 信息选择最优分支。但是，在一些情况下，有不同的方法可以做相同的事情，而 CPUID 指令没有提供关于哪种实现最好的必要信息。这些情况有时用汇编语言处理。下面是一些例子：

1. `strlen` 函数。字符串长度函数遍历字符串的所有字节以找到第一个值为零字节。好的实现使用 XMM 寄存器每次测试16个字节，然后使用 `BSF`（bit scan forward）指令在16个字节的块中定位第一个值为零字节。有些 CPU 对这个位扫描指令的实现特别慢。单独测试过 `strlen` 函数的程序员对该函数在位扫描指令很慢的 CPU 上的性能不满意，并为特定的CPU型号 实现了一个单独的版本。但是，我们必须考虑到，对于每个函数调用，位扫描指令只执行一次，因此您必须在性能变得重要之前调用该函数数十亿次，而很少有程序会这样做。因此，为位扫描指令的很慢的 CPU 编写特殊版本的 `strlen` 函数是不值得的。我的建议是使用位扫描指令，并期待这是在未来的 CPU 上最快的解决方案。
2. 一般大小的执行单元。向量寄存器已经从64位的 MMX 增加到了128位的 XMM 和256位的 YMM 寄存器。第一个支持128位向量寄存器的处理器实际上执行单元只有64位。每个128位操作被分成两个64位操作，因此使用更大的向量大小几乎没有任何速度上的优势。后来的型号拥有128位的完整执行单元，因此速度更快。同样，最初支持256位指令的处理器将256位读取操作拆分为两个128位读取操作。对于即将到来

的512位指令集以及将来寄存器大小的进一步扩展，也可以期望得到相同的结果。通常，新寄存器大小的全部优势只出现在支持它的第二代处理器中。在某些情况下，向量实现仅在具有全尺寸执行单元的 CPU 上是最优的。问题是CPU调度程序很难知道最大的向量寄存器是以半速还是全速处理的。这个问题的一个简单解决方案是，只有在支持下一个更高的指令集时才使用新的寄存器大小。例如，只有在这种应用程序中支持 AVX2 时才使用 AVX。或者，使用一个不利于使用最新指令集的处理器负面清单。

3. 高精度数学。用于高精度数学的库允许用大整数加法。这通常在循环中使用 `ADC`（带进位）指令完成，其中进位必须从一个迭代传递到下一个迭代中。进位可以保存在进位标志中，也可以保存在寄存器中。如果进位保存在进位标志中，那么循环分支必须依赖使用零标志的指令，并且不修改进位标志（例如 `DEC`、`JNZ`）。在将标志寄存器的分割为进位标志位和零标志位存在问题的老版本 Intel CPUs上，由于所谓的部分标志位（so-called partical flags）在处理器上存在问题，这个方案可能会导致很长的延迟，但在 AMD CPUs 上不存在这个问题（详见手册3：“The microarchitecture of Intel, AMD and VIA CPUs”）。这是少数几种使根据 CPU 品牌变得合理的情况之一。在特定品牌的最新 CPU 上表现良好的版本也可能使未来同样品牌其它型号的最优选择。较新的处理器支持用于高精度数学的 `ADX` 指令。
4. 内存复制。复制内存块有几种不同的方法。这些方法在手册2“Optimizing subroutines in assembly language”第17.9节“移动数据块”中进行了讨论，其中还讨论了在不同的处理器上哪种方法速度最快。在 C++ 程序中，您应该选择一个最新的，很好地实现了 `memcpy` 函数的函数库。由于存在不同的微处理器、不同的对齐方式和不同大小的需要复制的数据块等很多不同情况，因此惟一合理的解决方案是使用一个具有 CPU 调度的标准函数库。这个函数非常重要，并且被广泛使用，大多数函数库的这个函数都有 CPU 调度功能，尽管不是所有库都有最好的和最新的解决方案。编译器在复制大型对象时可能会隐式地使用 `memcpy` 函数，除非有一个复制构造函数以其他方式指定。

在诸如此类的困难情况下，重要的是要记住，您的代码很可能大部分时间在编程时还未知的处理器上运行。因此，重要的是要考虑哪种方法可能对未来处理器效果最好,并所有未知的，但支持必要的指令集的处理器选择该方法。如果问题在未来由于微处理器硬件设计的总体改进而消失，那么就不值得花费精力来根据复杂的条件或者具体 CPU 型号的清单来编写 CPU 调度器。

最终的解决方案的选择将需要性能测试，该测试度量关键代码的每个版本的速度，以查看在实际处理器上哪个解决方案是最优的。然而，这涉及到时钟频率可能的动态变化，以及由于中断和任务切换而导致测量不稳定的问题。因此，为了做出可靠的决策，有必要对不同的版本进行多次交替测试。

13.4 测试和维护

当软件使用 CPU 调度时，有两件事情需要测试：

1. 通过使用特定版本的代码，你在速度上获得了多少增益。
2. 检查所有版本代码是否正确工作。

速度测试最好是针对每个代码分支所特定的 CPU 类型进行。换句话说，如果您想优化几个不同的 CPU，就需要在几个不同的 CPU 上进行测试。

另一方面，没有必要使用许多不同的 CPU 来验证所有代码分支是否正确工作。一个使用低版本指令集的代码分支仍然可以在一个支持高版本指令集的 CPU 上运行。因此，您只需要一个支持最高版本指令集的 CPU 来测试所有分支的正确性。因此，建议在代码中添加一个测试特性，使您能够覆盖 CPU 调度并运行任何代码分支来进行测试。

如果代码是作为函数库或单独的模块实现的，那么编写一个可以单独调用所有代码分支并测试其功能的测试程序是很方便的。这对以后的维护非常有帮助。然而，这不是一本关于测试理论的教科书。关于如何测试软件模块的正确性的建议一定可以在其他地方找到。

13.5 实现

CPU 调度机制可以在不同的地方实现，在不同的时间做出调度决策：

1. 在每次调用时调度。使用分支树或者 `switch` 语句为关键函数选择合适的版本。每次调用关键函数时都会判断分支。这存在需要花费时间去判断分支的问题。
2. 在第一次调用时调度。通过函数指针调用函数，该指针最初指向一个调度器。调度器更改函数指针并使其指向函数的正确版本。这样做的好处是，在函数从未被调用时，调度器不会花费时间决定使用哪个版本。下面的**示例13.1**演示了这种方法。
3. 在初始化时生成指针。程序或函数库有一个初始化路径，该初始化路径在第一次调用关键函数之前调用。初始化路径将函数指针设置为函数的正确版本。这样做的好处是，函数调用的响应时间是一致的。
4. 每个代码版本都在一个单独的动态链接库（*.dll 或 *.so）中实现。程序有一个初始化例程，它加载库的适当版本。如果库非常大，或者必须使用不同的编译器编译不同的版本，则该方法非常有用。
5. 在加载阶段调度。程序使用一个过程链接表（**PLT**），过程连接表在程序加载时候初始化。这个方法需要操作系统的支持，再最新版本的 Linux 系统中可用（Mac OS 也许也可以），见下面第132页（TODO）。
6. 在安装时调度。每个代码版本都在一个单独的动态链接库（*.dll 或 *.so）中实现。安装程序创建适当版本库的符号链接，应用程序通过符号链接加载库。
7. 使用不同的可执行文件。如果指令集互不相容，可以使用这种方法。您可以为32位和64位系统创建单独的可执行程序。程序的适当版本可以在安装过程中选择，也可以通过可执行文件选择。

如果关键代码的不同版本使用不同的编译器编译，那么建议对所有关键代码调用的库函数使用静态链接，这样你不需要分发应用程序中属于不同编译器的所有代码。

各种指令集的可用性可以通过调用系统函数来确定（例如：在 Windows 中可以调用 `IsProcessorFeaturePresent`）。或者，你可以直接调用 CPUID 指令，也可以使用我提供的函数库中的 CPU 检测函数，这个函数的名字是 `InstructionSet()`。下面的这个例子将展示如何使用 `InstructionSet()` 实现在第一次调用方法时进行 CPU 调度：

```
// Example 13.1
// CPU dispatching on first call

// Header file for InstructionSet()
#include "asmlib.h"

// Define function type with desired paramet
typedef int CriticalFunctionType(int parm1, int parm2);

// Function prototype
CriticalFunctionType CriticalFunction_Dispatch;

// Function pointer serves as entry point.
// After first call it will point to the appropriate function version
CriticalFunctionType * CriticalFunction = &CriticalFunction_Dispatch;

// Lowest version
int CriticalFunction_386(int parm1, int parm2) {...}

// SSE2 version
int CriticalFunction_SSE2(int parm1, int parm2) {...}

// AVX version
int CriticalFunction_AVX(int parm1, int parm2) {...}

// Dispatcher. Will be called only first time
int CriticalFunction_Dispatch(int parm1, int parm2)
{
    // Get supported instruction set, using asmlib library
    int level = InstructionSet();
    // Set pointer to the appropriate version (May use a table
    // of function pointers if there are many branches):
    if (level >= 11)
    {
        // AVX supported
        CriticalFunction = &CriticalFunction_AVX;
    }
    else if (level >= 4)
    {
        // SSE2 supported
        CriticalFunction = &CriticalFunction_SSE2;
    }
    else
    {
        // Generic version
        CriticalFunction = &CriticalFunction_386;
    }
    // Now call the chosen version
    return (*CriticalFunction)(parm1, parm2);
}

int main()
{
    int a, b, c;
    ...
    // Call critical function through function pointer
```

```
    a = (*CriticalFunction)(b, c);
    ...
    return 0;
}
```

函数 `InstructionSet()` 包含在函数库 asmlib。这个函数是独立于操作系统的，它检查 CPU 和操作系统是否支持不同的指令集。**例13.1**中 `CriticalFunction` 的不同版本可以在必要时放在单独的模块中，每个模块都为特定的指令集编译。

13.6 GNU 编译器中的 CPU 调度

Linux 中引入了一个名为"Gnu 间接函数"的特性，并被2010年的 Gnu 实用工具所支持。该特性用于 CPU 调度，并在Gnu C 库中被使用。它需要编译器、链接器和加载器的支持（*binutils*的版本为 2.20, `glibc` 版本为 2.11的 *ifunc* 分支)。

使用这个特性按照下述方法以一种普通的方式使用过程连接表（PLT)：同一函数有两个或多个版本，每个版本都针对特定的 CPU 或其他硬件条件进行了优化。调度函数决定使用哪个函数，并返回指向所需函数的指针。PLT 入口最初指向调度函数。当程序加载时，加载器调用调度函数，并用从调度函数获得的指针替换 PLT 入口。这将使对函数的任何调用转到所需的版本。注意，调度函数通常在程序开始运行之前调用，并且在调用任何构造函数之前调用。因此，调度函数不能依赖于正在初始化的任何其他东西。即使从未调用被调度函数，也很可能调用调度函数。

不幸的是，目前在 Gnu 手册中描述的语法并不能正确工作。相反，可以使用以下方案：

```
// Example 13.2. CPU dispatching in Gnu compiler
// Same as example 13.1, Requires binutils version 2.20 or later

// Header file for InstructionSet()
#include "asmlib.h"

// Lowest version
int CriticalFunction_386(int parm1, int parm2) {...}

// SSE2 version
int CriticalFunction_SSE2(int parm1, int parm2) {...}

// AVX version
int CriticalFunction_AVX(int parm1, int parm2) {...}

// Prototype for the common entry point
extern "C" int CriticalFunction ();
__asm__ (".type CriticalFunction, @gnu_indirect_function");

// Make the dispatcher function.
typeof(CriticalFunction) * CriticalFunctionDispatch(void)
    __asm__ ("CriticalFunction");
typeof(CriticalFunction) * CriticalFunctionDispatch(void)
{
    // Returns a pointer to the desired function version
    // Get supported instruction set, using asmlib library
    int level = InstructionSet();
    // Set pointer to the appropriate version (May use a table
    // of function pointers if there are many branches):
    if (level >= 11)
    {
        // AVX supported
        return &CriticalFunction_AVX;
    }
    if (level >= 4)
    {
        // SSE2 supported
        return &CriticalFunction_SSE2;
    }
```

```
    // Default version
    return &CriticalFunction_386;
}

int main()
{
    int a, b, c;
    ...
    // Call critical function
    a = CriticalFunction(b, c);
    ...
    return 0;
```

在 Gnu C 函数库中，间接函数特性被用于一些特别关键的函数。

13.7 Intel 编译器中的 CPU 分派

Intel 编译器有一个特性，可以为一个函数的生成多个版本对应多个 Intel CPU。每次调用方法都使用分派。当调用该函数时，为函数分配所期望的版本。通过使用选项 */QaxAVX* 或 *-axAVX* 编译模块，可以对模块中所有合适的函数进行自动分派。甚至会给非关键函数也生成多个版本。通过使用指令 `_declspec(cpu_dispatch(…))`，可以只对速度关键的函数执行分派。有关详细信息，请参阅 Intel C++ 编译器文档。注意，Intel 编译器中的 CPU 分派机制只适用于 Intel CPU，而不适用于 AMD 和 VIA 等其他品牌的 CPU。下一节展示了一种解决 CPU 检测机制中的这种限制和其他缺陷的方法。

Intel 编译器中的 CPU 分派机制的效率低于 Gnu 编译器的机制，因为它对关键函数的每次调用都进行分派。在某些情况下，每次调用函数时，Intel 的机制都会执行一系列分支，而 Gnu 机制则在过程链接表中存储指向所需版本的指针。如果一个分派函数调用另一个分派函数，那么后者的分派分支也将被执行，即使此时已经知道 CPU 的类型。这可以通过内联后一个函数来避免，但是更好的方法是像**示例13.1** 中（page 130，TODO）那样显式地执行 CPU 分派。

Intel 编译器和函数库具有自动 CPU 分派的特性。针对不懂的处理器和指令集，很多 Intel 函数库都有几个不同的版本。同样的，编译器可以使用自动 CPU 分派为用户写的代码生成多个版本的代码。

不幸的是，Intel 编译器的 CPU 检测机制存在几个缺陷：

1. 只有在运行在 Intel 处理器上时才会选择代码的最佳版本。CPU 调度程序在检查它支持的指令集之前，检查处理器是否是 Intel 的。如果处理器不是 Intel 的，则选择较差版本的代码，即使处理器与较好版本的代码兼容。这可能导致在 AMD 和 VIA 处理器上的性能急剧下降。
2. 显式 CPU 分派只适用于 Intel 处理器。对于非 Intel 处理器，通过简单地执行一个非法操作，使分派器发出错误信号，然后使程序崩溃。
3. CPU 分派器 不检查操作系统是否支持 XMM 寄存器。它将在不支持 SSE 的旧操作系统上崩溃。

由 Intel 发布的几个函数库具有类似的 CPU 分派机制，其中一些函数库还以次优方式处理非 Intel CPU。

英特尔 CPU 分派器以非最佳方式处理非英特尔 CPU 的事实已经成为一个严重的法律问题。详情请参阅我的博客。

Intel 编译器的行为使程序员陷入了一个糟糕的困境。您可能更喜欢使用 Intel 编译器，因为它具有许多高级优化特性，而且您可能希望使用经过良好优化的 Intel 函数库，但是谁愿意给程序加上一个说它在非 Intel机器上不能很好地工作的标签呢?

这个问题可能的解决方案如下所列：

1. 使用特定的指令集编译，例如 */arch:SSE2*。编译器将为这个指令集生成最优代码，并且大多数库函数直插入 SSE2 版本，而不进行 CPU 分派。测试一下程序是否在非英特尔 CPU 上令人满意地运行。如果没有，则可能需要替换 CPU 检测功能，如下所述。该程序将与不包含当前选择的指令集的旧版本微处理器不兼容。
2. 为代码中最关键的部分创建两个或多个版本，并使用指定的合适指令集分别编译它们。在代码中插入显式的 CPU 分派，以调用适合其运行的微处理器的版本。
3. 替换或绕过英特尔编译器的 CPU 检测功能。该方法将在下面一节中讨论。
4. 直接调用特定于 CPU 版本的库函数。特定于 CPU 的函数的名称带有后缀，例如对于 AVX 指令集后缀为 *.R*。这些后缀在手册5：“calling conventions”中的**表19**中列出。函数名中符号. 在 C++ 中是不被允许的，因此您需要使用汇编代码或 objconv 或类似的实用程序来修改对象文件中的名称。
5. 使用在所有品牌 CPU 上都运行良好的函数库。

如果程序中最耗时的部分包含 自动 CPU 分派或内存密集型函数，如 `memcpy`、`memmove`、`memset`或数学函数，如`pow`、`log`、`exp`、`sin`等，则可以使用上述一种或多种方法改进非 intel 处理器的性能。

<u>重载 Intel CPU 检测功能</u>

在某些情况下，CPU 检测功能有两个版本，一个区分 CPU 品牌，另一个不区分。

没有文档的 Intel 库函数 `_intel_cpu_features_init()` 设置变量 `_intel_cpu_feature_indicator`，其中每个位表示 Intel CPU上特定的 CPU 特性。只需将这些变量设置为零，然后调用`_intel_cpu_features_init_x()`，就可以绕过对 CPU 品牌的检查。

在其他情况下，可以通过使另一个具有相同名称的函数来替换 Intel 函数库和编译器生成的代码中的 CPU 检测函数。在 Windows 操作系统，这需要使用静态链接（例如，选项 */MT*）。在 Linux 和 Mac 系统中，静态链接和动态链接都能起作用。

http://www.agner.org/optimize/asmlib.zip 中的文件包含这些方法的完整代码示例。

如果您正在使用 Intel 编译器，那么请确保在编译启动代码和 `main()` 时没有任何限制 CPU 品牌的选项。然后，代码的关键部分可以放在一个单独的 C 或 C++文件中，并为所需的指令集编译。如果这些方法中有任何一种绕过了 CPU 品牌检查，那么关键部分就可以在任何 CPU 品牌上得到最佳的性能。

当 Intel 函数库与其他编译器一起使用时，这些方法也可以工作。包括叫做 MKL、VML 和 SVML 的函数库。IPP 函数库不需要任何补丁。

注意，这些方法是基于我自己的研究，而不是基于公开的信息。它们在Intel编译器版本 7 到 14 的测试中运行良好，每个版本都有一些变化。这些示例适用于 Windows 和 Linux, 32位和64位。它们还没有在 Mac 系统中进行测试。

14 具体的优化主题

14.1 使用查找表

如果列表是被缓存过的，从表中读取一个常数的值是很快的。通常情况下在缓存一级缓存的列表读取操作只需要花费几个时钟周期。如果函数只有有限数量的可能输入，我们可以利用这个事实，用查找表来替换函数调用。

让我们以整数阶乘函数（$n!$）为例。唯一允许的输入是从0到12的整数。更高的输入会导致溢出，负的输入导会得到无穷大。阶乘函数的一个典型实现是下面这样的：

```
// Example 14.1a
int factorial (int n)
{
    // n!
    int i, f = 1;
    for (i = 2; i <= n; i++)
        f *= i;
    return f;
```

这种计算需要 $n-1$次乘法，而这将会花费当长的时间。如果使用查找表的话效率会更高：

```
// Example 14.1b
int factorial (int n)
{
    // n!
    // Table of factorials:
    const int FactorialTable[13] = {1, 1, 2, 6, 24, 120, 720,
        5040, 40320, 362880, 3628800, 39916800, 479001600};
    if ((unsigned int)n < 13)
    {
        // Bounds checking (see page 137)
        return FactorialTable[n]; // Table lookup
    }
    else
    {
        return 0; // return 0 if out of range
    }
```

该实现使用查找表，而不是每次调用函数时重新计算值。我在这里添加了一个界限检查，因为当 `n` 是数组索引时，`n` 超出范围的后果可能比n是循环计数时更严重。边界检查的方法在下面的137页（TODO）解释。

表应该声明为 `const`，以便启用常量传播和其他优化。您可以将函数声明为内联的。

用查找表替换函数，在可能输入的数量有限且没有缓存问题的大多数情况下是有利的如果你期望每次调用后列表从缓存中被擦出，以及计算函数所花费的时间小于重新加载值从内存的时间，加上程序的其他部分占据缓存所导致的时间开销之和，那么使用查找表是没有好处的。

查找表无法使用当前的指令集进行向量化。如果这妨碍了更快的向量化代码，那么就不用使用查找表。

在静态内存中存储数据可能会导致缓存问题，因为静态数据可能分散在不同的内存地址。如果缓存是一个问题，那么将表从静态内存复制到最内层循环外的栈内存上可能是有用的。我们可以在函数中但在最内层循环之外声明表（不使用 `static` 关键字）：

```
// Example 14.1c
void CriticalInnerFunction ()
{
    // Table of factorials:
    const int FactorialTable[13] = {1, 1, 2, 6, 24, 120, 720,
        5040, 40320, 362880, 3628800, 39916800, 479001600};
    ...
    int i, a, b;
    // Critical innermost loop:
    for (i = 0; i < 1000; i++)
    {
        ...
        a = FactorialTable[b];
        ...
    }
}
```

例 14.1c中的 `FactorialTable` 在调用 `CriticalInnerFunction` 时从静态内存中复制到栈上。编译器将表存储在静态内存中，并在函数开始的地方插入代码，将表复制到栈内存中。当然，复制表需要额外的时间，但是当它位于关键的最内层循环之外时，这是被允许的。循环将使用存储在栈内存中的表的副本，这与其它本地变量相邻，因此缓存效率可能比静态内存更高。

如果您不喜欢手工计算表值并将值插入代码中，那么您当然可以让程序进行计算。只要只需要一次计算，那么计算表所花费的时间并不重要。有人可能会说，在程序中计算表比直接输入值更安全，因为手写表中的输入错误可能无法被检测到。

查找表的原理可用于程序在两个或多个常量之间进行选择的任何情况。例如，在两个常量之间进行选择的分支可以被一个包含两个条目的表替换。如果分支的可预测性很差，这可能会提高性能。例如：

```
// Example 14.2a
float a; int b;
```

如果我们假设 `b` 总是 0 或 1，并且它的值可预测性很差，那么使用查找表来代替分支是有利的：

```
// Example 14.2b
float a; int b;
const float OneOrTwo5[2] = {1.0f, 2.5f};
```

在这里，因为安全性的原因，我将 `b` 按位与上 1，`b & 1`的值肯定只有 0 或 1（参见第138页，TODO）。 如果 `b` 的值肯定为 0 或 1，那么就可以省略对 `b` 的额外检查。使用 `a = OneOrTwo5[b!=0]`，同样可以正确运行，但是效率稍低。但是，当 `b` 是 `float` 或 `double` 类型时，这种方法效率很低，因为我测试的所有编译器对`OneOrTwo5[b!=0]` 的实现都是 `OneOrTwo5[(b!=0) ? 1 : 0]`，在这种情况下，我们无法去掉分支。当 `b` 是浮点数时，编译器使用不同的实现似乎不合逻辑。我想，原因是编译器制的开发人员假定浮点数比较比整数比较更容易预测。对于表达式 `a = 1.0f + b * 1.5f`，当 `b` 是一个浮点数时是高效的，但如果 `b` 是一个整数则效率较低，因为整数到浮点数的转换比查找表花费更多的时间。

将查找表作为 `switch` 语句的替代尤其有利，因为 `switch` 语句的可预测性经常较差。例如：

```
// Example 14.3a
int n;
switch (n)
{
```

```
    case 0:
        printf("Alpha"); break;
    case 1:
        printf("Beta"); break;
    case 2:
        printf("Gamma"); break;
    case 3:
        printf("Delta"); break;
```

这可以使用查找表来提升效率：

```
// Example 14.3b
int n;
char const * const Greek[4] = {
    "Alpha", "Beta", "Gamma", "Delta"
    };
if ((unsigned int)n < 4)
{
    // Check that index is not out of range
    printf(Greek[n]);
```

表的声明有两个 `const`，因为它们指向的指针和文本都是常量。

14.2 边界检查

在 C++ 中，通常有必要检查数组索引是否超出范围。这常常看起来是这样的：

```
// Example 14.4a
const int size = 16; int i;
float list[size];
...
if (i < 0 || i >= size)
{
    cout << "Error: Index out of range";
}
else
{
    list[i] += 1.0f;
```

`i < 0` 和 `i >= size` 这两个比较可以使用一个比较替换：

```
// Example 14.4b
if ((unsigned int)i >= (unsigned int)size)
{
    cout << "Error: Index out of range";
}
else
{
    list[i] += 1.0f;
```

当 `i` 被解释为无符号整数时，`i` 可能的负值将以一个较大的正数出现，这将触发错误条件。用一个比较替换两个比较可以加快代码的速度，因为测试一个条件相对比较昂贵，而类型转换根本不会生成额外的代码。

这个方法可以扩展到一般情况下：你想要检查一个整数是否在一个特定的区间之内：

```
// Example 14.5a
```

```
const int min = 100, max = 110; int i;
...
```

可以修改成：

```
// Example 14.5b
```

如果所需区间的长度是2的幂，则有一种更快的方法来限制整数的范围。例如：

```
// Example 14.6
float list[16]; int i;
...
```

这需要略微解释一下。`i&15` 的值肯定在 0 到 15 的区间内。如果 `i` 在这个区间之外，例如 `i = 18`，那么 `&` 操作符（按位与）将 `i` 的二进制值截断为 4 位，结果将是 2。结果与 `i` 除上 16 的余数相同。如果我们不需要错误消息的话，这种方法在数组索引超出范围时可以防止程序错误。需要注意的是，这种方法只适用于2的幂（即2、4、8、16、32、64、.....）。通过按位与上2^{n-1}，我们可以确保一个数的值小于 2^n，并且不是负的。按位与操作隔离数字中有效的低 n 位，并将所有其他位设为零。

14.3 使用位运算符一次检查多个值

位运算符 `&`，`|`，`^`，`~`，`<<`，`>>` 可以在一次操作中测试或操作整数的所有位。例如，如果 32 位整数的每个位都有特定的含义，那么可以使用 `|` 运算符在一个操作中设置多个位；你用 `&` 运算符清除或遮掩掉多个位。你可以用 `^` 运算符转换多个位。

`&` 运算符对于测试单个操作中的多个条件也很有用。例如：

```
// Example 14.7a. Testing multiple conditions
enum Weekdays {
    Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday
};
Weekdays Day;
if (Day == Tuesday || Day == Wednesday || Day == Friday)
{
    DoThisThreeTimesAWeek();
```

本例中的 `if` 语句有三个条件，它们被实现为三个分支。如果将 `Sunday`、`Monday` 等常量定义为 2的幂，则可以将它们合并为一个分支：

```
// Example 14.7b. Testing multiple conditions using &
enum Weekdays {
    Sunday = 1, Monday = 2, Tuesday = 4, Wednesday = 8,
    Thursday = 0x10, Friday = 0x20, Saturday = 0x40
};
Weekdays Day;
if (Day & (Tuesday | Wednesday | Friday))
{
    DoThisThreeTimesAWeek();
```

通过在**例14.7b**中给每个常数的值设置成 一个2的幂，我们实际上是在使用 `Day` 中的每一位来表示星期几。我们可以用这种方法定义的常量的最大数量等于整数中的位的数量，通常是 32。在 64 位系统中，我们可以使用 64 位整数，这几乎没有任何性能上的损失。

在**例 14.7b**中的表达式 `(Tuesday | Wednesday | Friday)` 被编译器转换成 `0x2C`，这样的话 `if` 条件就可以通过一个 `&` 操作来计算，而这是很快的。如果变量 `Day` 中设置了 `Tuesday`、`Wednesday` 或 `Friday` 中的的任何位，`&` 操作的结果将是非零的，因此将被视为真。

注意布尔运算符 `&&`，`||`，`!` 以及相应的位运算符 `&`，`|`，`~`。布尔运算符产生一个结果，true（1）或 false （0），且第二个操作数只在需要时计算。位操作符在应用于32位整数时会产生32个结果，它们总是对两个操作数求值。然而，位运算符的计算速度比布尔运算符快得多，因为只要操作数是整数表达式而不是布尔表达式，它们就不需要使用分支。

当使用整数作为布尔向量时，位操作符可以做很多事情，而且这些操作非常快。这在有许多布尔表达式的程序中很有用。无论常量是用 `enum`、`const` 还是 `#define` 定义的，都不会影响性能。

14.4 整数乘法

整数乘法比加法和减法需要更长的时间（3 - 10个时钟周期，取决于处理器）。编译器优化通常会用一个常量替换整数乘法，并结合加法和移位操作。乘以2的幂要比乘以其他常数快，因为它可以通过移位操作完成。例如，`a*16` 使用 `a << 4`计算，`a * 17` 使用 `(a << 4) + a` 计算。当与常数相乘时，你可以通过使用2的幂来利用这个优势。编译器也有快速乘以 3、5 和 9 的方法。

在计算数组元素的地址时，会有隐式的乘法计算。在某些情况下，当因子为2的幂时，这个乘法会更快。例如：

```
// Example 14.8
const int rows = 10, columns = 8;
float matrix[rows][columns];
int i, j;
int order(int x);
...
for (i = 0; i < rows; i++)
{
    j = order(i);
    matrix[j][0] = i;
```

这里，`matrix[j][0]` 的地址在内部使用下面的式子计算：

`(int)&matrix[0][0] + j * (columns * sizeof(float))`。

现在，要乘以 `j` 的因子是 `(cloumns * sizeof(float)) = 8 * 4 = 32`。这是 2 的幂，所以编译器可以用 `j << 5` 替换 `j * 32`。如果列的大小不是 2 的幂，那么乘法会花费更长的时间。因此，如果以无序方式访问矩阵中的行，则将矩阵中的列数设置为 2 的幂是有利的。

这同样适用于结构体或类元素数组。如果以无序方式访问对象，则每个对象的大小最好是 2 的幂。例如：

```
// Example 14.9
struct S1
{
    int a;
    int b;
    int c;
    int UnusedFiller;
};
int order(int x);
const int size = 100;
S1 list[size]; int i, j;
...
for (i = 0; i < size; i++)
{
    j = order(i);
    list[j].a = list[j].b + list[j].c;
```

在这里，我们在结构体中插入了 `UnusedFiller`，以确保其大小是的 2 的幂，以使地址计算的更快。

使用2的幂的优势只适用于以无序方式访问元素的情况。如果 **示例 14.8**和 **14.9** 中的代码发生了更改，以 `i` 代替 `j` 作为索引，那么编译器可以看到地址是按顺序访问的，并且可以通过在前一个地址上添加一个常量来计算每个地址（参见第72页，TODO）。在这种情况下，大小是否为2的幂并不重要。

使大小为2次幂的建议并不适用于非常大的数据结构。相反，如果矩阵太大以至于缓存成为问题，则应该尽量避免大小为2的幂。如果矩阵中的列数是2的幂，并且矩阵大于缓存，那么就可以得到代价非常昂贵的缓存竞争，如第98页（TODO）所解释的那样。

14.5 整数除法

整数除法要比加法、减法和乘法（32位整数的时钟周期为27 - 80个，具体取决于处理器）耗时长得多。

整数除以2的幂可以用移位运算来做，这样会快得多。

除以一个常数比除以一个变量快的多，因为编译器优化可以通过选择合适的 n 以 $a*(2^n/b) >> n$ 来计算 a/b。 常量 $(2^n/b)$ 是被预先计算好的，并乘法是通过位的扩展数（extended number of bits）来完成的。该方法稍微复杂一些，因为必须添加符号和舍入误差的各种更正。该方法在手册2： "Optimizing subroutines in assembly language" 中有更详细的描述。当被除数是无符号的，该方法会快的多。

以下准则可用于改进包含整数除法的代码：

1. 整数除以常数比变量快。确保在编译时知道除数的值。
2. 如果常数是 2 的幂的话，整数除法会更快。
3. 当被除数是无符号时，整数除以常量会更快。

例如：

```
// Example 14.10
int a, b, c;
a = b / c; // This is slow
a = b / 10; // Division by a constant is faster
a = (unsigned int)b / 10; // Still faster if unsigned
a = b / 16; // Faster if divisor is a power of 2
```

相同的准则同样适用于取模运算：

```
// Example 14.11
int a, b, c;
a = b % c; // This is slow
a = b % 10; // Modulo by a constant is faster
a = (unsigned int)b % 10; // Still faster if unsigned
a = b % 16; // Faster if divisor is a power of 2
```

可以利用这些指导原则，如果可能的话，可以使用一个2的幂的常数做为除数，如果确定被除数不为负数，可以将被除数更改为无符号。

如果除数在编译时是未知，但程序不断重复除以同一个除数，仍然可以使用上述方法。在这种情况下，您必须在编译时（? 需要确认下，应该是运行的时候把？）对 $(2^n/b)$ 等进行必要的计算。www.agner.org/optimize/asmlib.zip中的函数库包含用于计算的各种函数。

将循环计数器除以一个常数，可以用按常数将循环展开来代替：

```
// Example 14.12a
int list[300];
int i;
for (i = 0; i < 300; i++)
{
    list[i] += i / 3;
```

这个可以使用下面的代码替换：

```
// Example 14.12b
int list[300];
int i, i_div_3;
for (i = i_div_3 = 0; i < 300; i += 3, i_div_3++)
{
    list[i] += i_div_3;
    list[i+1] += i_div_3;
    list[i+2] += i_div_3;
```

类似的方法也可以用于避免模运算：

```
// Example 14.13a
int list[300];
int i;
for (i = 0; i < 300; i++)
{
    list[i] = i % 3;
```

可以被替换成：

```
// Example 14.13b
int list[300];
int i;
for (i = 0; i < 300; i += 3)
{
    list[i] = 0;
    list[i+1] = 1;
    list[i+2] = 2;
```

示例 14.12b和**14.13b**中的循环展开仅当循环计数可被展开因子整除时才有效。如果不能被整除，则必须在循环之外执行额外的操作：

```
// Example 14.13c
int list[301];
int i;
for (i = 0; i < 301; i += 3)
{
    list[i] = 0;
    list[i+1] = 1;
    list[i+2] = 2;
}
```

14.6 浮点数除法

浮点数除法比加法、减法和乘法（20 - 45个时钟周期）耗时要长得多。

浮点数除以一个常数可以用乘以常数的倒数来代替：

```
// Example 14.14a
double a, b;
```

可以把这个改成：

```
// Example 14.14b
double a, b;
```

编译器将在编译时计算 `(1./1.2345)` 的值，并将倒数插入到代码中，因此您将不会在除法上花费时间。一些编译器会自动将**示例 14.14a**中的代码替换为**14.14b**的，但只有在某些选项被设置为放宽浮点精度时才会这样做（请参阅第74页，TODO）。因此显式地进行这种优化更加安全。

有时除法会被完全消除，例如：

```
// Example 14.15a
```

有时会被替换成：

```
// Example 14.15b
```

但是要注意这里的陷阱：如果 `c < 0`，不等式符号必须反转。如果 `b` 和 `c` 是整数，除法是不精确的，而乘法是精确的。

乘法和除法可以结合在一起，例如：

```
// Example 14.16a
double y, a1, a2, b1, b2;
```

这里我们可以通过公分母来消去一个除法：

```
// Example 14.16b
double y, a1, a2, b1, b2;
```

使用公分母的技巧甚至可以用于完全独立的除法。例如：

```
// Example 14.17a
double a1, a2, b1, b2, y1, y2;
y1 = a1 / b1;
```

这可以这样变化：

```
// Example 14.17b
double a1, a2, b1, b2, y1, y2, reciprocal_divisor;
reciprocal_divisor = 1. / (b1 * b2);
y1 = a1 * b2 * reciprocal_divisor;
```

14.7 不要将 float 和 double 混合在一起

不管您使用的是单精度还是双精度，浮点数的计算通常花费相同的时间。但是在为64位操作系统编译的程序和使用指令集 SSE2 或更高版本编译的程序中，混合使用单精度和双精度是有代价的。例如：

```
// Example 14.18a
float a, b;
```

C/C++ 标准规定所有浮点数常量在默认情况下都是双精度的。 所以在这个例子中， `1.2` 是一个双精度的常量。因此，在将 `b` 与双精度常数相乘之前，需要将 `b` 从单精度转换为双精度，然后再将结果转换回单精度。这些转换需要很多时间。您可以通过避免转换，来使代码达到 5 倍的效率，无论是通过使常数编程单精度或 使 `a` 和 `b`双精度编程双精度的：

```
// Example 14.18b
float a, b;
a = b * 1.2f; // everything is float
// Example 14.18c
double a, b;
```

当为没有 SSE2 指令集的旧处理器编译代码时，混合不同的浮点精度不会带来任何损失，但是最好在所有操作数中保持相同的精度，以防代码稍后被移植到另一个平台。

14.8 浮点数和整数之间的转换

<u>将浮点数转换成整数</u>

根据 C++ 语言的标准，所有从浮点数到整数的转换都使用向零的截断，而不是四舍五入。这是不幸的，因为除非使用 SSE2 指令集，否则截断要比舍入花费更长的时间。如果可能，建议启用 SSE2 指令集。SSE2 总是在64位模式下被启用。

在没有 SSE2 的情况下，从浮点数到整数的转换通常需要 40 个时钟周期。如果在代码的关键部分不能避免从 `float` 或 `double` 到 `int` 的转换，那么可以使用舍入而不是截断来提高效率。这大约快了三倍。程序的逻辑可能需要修改，以补偿舍入和截断之间的差异。

使用 `lrintf` 和 `lrint` 函数可以高效地将浮点数或双精度数转换为整数。不幸的是，由于对 C99 标准的争议，许多商业编译器中缺少这些这些函数。下面的**示例14.19**给出了 `lrint` 函数的实现。该函数将浮点数四舍五入到最近的整数。如果两个整数相等，则返回偶数。没有溢出检查。此函数适用于32位 Windows 和32位 Linux 以及微软、英特尔和 Gnu 编译器。

```
// Example 14.19
static inline int lrint (double const x)    { // Round to nearest integer
int n;
#if defined(__unix__) || defined(__GNUC__)
// 32-bit Linux, Gnu/AT&T syntax:
__asm ("fldl %1 \n fistpl %0 " : "=m"(n) : "m"(x) : "memory" );
#else
// 32-bit Windows, Intel/MASM syntax:
__asm fld qword ptr x;
__asm fistp dword ptr n;
#endif
return n;
```

这段代码只适用于 Intel/x86 兼容的微处理器。函数库amslib也提供了该函数。

在64位模式下或启用 SSE2 指令集时，四舍五入和截断之间的速度没有差别。缺失的功能在64位模式或启用SSE2指令集时可以按如下代码实现：

```
// Example 14.21. // Only for SSE2 or x64
#include <emmintrin.h>
static inline int lrintf (float const x)
{
    return _mm_cvtss_si32(_mm_load_ss(&x));
}
static inline int lrint (double const x)
{
    return _mm_cvtsd_si32(_mm_load_sd(&x));
```

示例14.21 中的代码比其他四舍五入方法更快，但当启用 SSE2 指令集时，它既不比截断快，也不比截断慢。

<u>将整数转换成浮点数</u>

整数到浮点数的转换比浮点数转换到整数快。转换时间通常在 5 到 20 个时钟周期之间。在某些情况下，对浮点变量进行简单的计算可能是有利的，以避免从整数到浮点的转换。

无符号整数转换为浮点数的效率低于有符号整数转换成浮点数。如果无符号整数转换为有符号整数不会导致溢出，那么在转换为浮点数之前将无符号整数转换为有符号整数效率会更高。例如：

```
// Example 14.22a
unsigned int u; double d;
```

如果你确定$u < 2^{31}$，那么在转换为浮点数之前先将其转换为有符号的:

```
// Example 14.22b
unsigned int u; double d;
```

14.9 使用整数操作来操作浮点型变量

根据 IEEE 754 (1985) 标准，浮点数以二进制表示形式存储。几乎在所有现代微处理器和操作系统中都使用这个标准（一些非常老的DOS编译器除外）。

`float`、`double` 和 `long double` 的表示法反映了$\pm 2^{eee}.1.ffff$的浮点值。$\pm$表示符号，eee 是指数，$fffff$ 是分数形式的二进制小数。符号位存储为单个位，0 表示正数，1 表示负数。指数存储为偏置二进制整数，分数存储为二进制数。如果可能的话，指数总是规格化的，所以小数点前的值是 1。这个“1”不包括在表示形式中，除非是 `long double` 类型。格式可以表示如下：

```
struct Sfloat
{
    unsigned int fraction : 23; // fractional part
    unsigned int exponent : 8; // exponent + 0x7F
    unsigned int sign : 1; // sign bit
};

struct Sdouble
{
    unsigned int fraction : 52; // fractional part
    unsigned int exponent : 11; // exponent + 0x3FF
    unsigned int sign : 1; // sign bit
};

struct Slongdouble
{
    unsigned int fraction : 63; // fractional part
    unsigned int one : 1; // always 1 if nonzero and normal
    unsigned int exponent : 15; // exponent + 0x3FFF
    unsigned int sign : 1; // sign bit
};
```

非零浮点数的值可以使用下面的方式计算：

$$floatvalue = (-1)^{sign} * 2^{exponent-127} * (1 + fraction * 2^{-23}),$$

$$doublevalue = (-1)^{sign} * 2^{exponent-1023} * (1 + fraction * 2^{-52}),$$

$$longdoublevalue = (-1)^{sign} * 2^{exponent-16383} * (1 + fraction * 2^{-63}).$$

如果除符号位之外的所有位都为 0，则值为 0。有没有符号位都可以是 0。

浮点格式是标准化的这一事实允许我们使用整数操作直接操作浮点表示的不同部分。这可能是一个优势，因为整数操作比浮点操作快。只有当你确信你知道你在做什么时，你才应该使用这些方法。有关注意事项，请参阅本节的末尾。

我们只需要反转一个符号位就可以改变浮点数的符号：

```
// Example 14.23
union
{
    float f;
    int i;
} u;
```

我们可以将符号位设置成0以得到绝对值：

```
// Example 14.24
union
{
    float f;
    int i;
```

```
} u;
```

我们可以通过测试除符号位以外的所有位来检查浮点数是否为零：

```
// Example 14.25
union
{
    float f;
    int i;
} u;
if (u.i & 0x7FFFFFFF)
{
    // test bits 0 - 30
    // f is nonzero
}
else
{
    // f is zero
```

我们对指数部分加上 n 就可以将一个非零浮点数乘上 2^n：

```
// Example 14.26
union
{
    float f;
    int i;
} u;
int n;
if (u.i & 0x7FFFFFFF)
{
    // check if nonzero
    u.i += n << 23; // add n to exponent
```

示例 14.26不会检查溢出，而且只有n是整数时才能有用。当没有下溢风险时，你可以对指数部分减去 n 以达到除以 2^n 的目的。

```
// Example 14.27
union
{
    float f;
    int i;
} u, v;
if (u.i > v.i)
{
    // u.f > v.f if both positive
```

在 **例 14.27**假设我们知道 `u.f`，`v.f` 都是正的。如果两者都是负数，或者其中一个为 0，另一个为 -0（符号位为0），则会失败。

我们可以将符号位移出来比较绝对值：

```
// Example 14.28
union
{
    float f;
    unsigned int i;
} u, v;
if (u.i * 2 > v.i * 2)
```

```
{
    // abs(u.f) > abs(v.f)
```

例 14.28中乘以 2 将移出符号位，使其余位表示浮点数绝对值的单调递增函数。

我们可以通过设置分数部分的位将在区间 $0 <= n < 2^{23}$的整数转换成在区间 $[1.0, 2.0)$的浮点数：

```
// Example 14.29
union
{
    float f;
    int i;
} u;
int n;
```

该方法对随机数生成器非常有用。

通常，如果浮点变量存储在内存中，那么以整数的形式访问它会更快，但如果它是寄存器变量，则不会更快。`union` 强制变量存储在内存中，至少是临时存储。因此，如果相同变量使用寄存器可以使其它临近代码获益时，那么使用上述示例中的方法将没有好处。

在这些例子中，我们使用 `union` 而不是指针的类型转换，时因为这种方法更安全。指针的类型转换可能不适用于遵循标准 C 的严格别名规则的编译器，该规则指定不同类型的指针不能指向同一对象，`char` 指针除外。

上面的例子都使用单一精度。在 32 位系统中使用双精度浮点数会变得更复杂。双精度浮点数用 64 位表示，但是 32 位系统不支持 64 位整数。许多 32 位系统允许您定义 64 位整数，但是它们实际上用两个32位整数来表示，效率较低。您可以使用双精度浮点数的高 32 位，它允许你访问符号位、指数和分数中的高几位。例如，可以这样测试双精度浮点数的符号：

```
// Example 14.23b
union
{
    double d;
    int i[2];
} u;
if (u.i[1] < 0)
{
    // test sign bit
    // u.d is negative or -0
```

不建议通过修改 `double` 类型的一半二进制位来修改它，比如，如果你想要通过 `u.i[1] ^= 0x80000000` 来反转上述示例中的符号位的话，但这很在 CPU 中产生存储转发延迟（参见手册3：“The microarchitecture of Intel, AMD and VIA CPUs”）。在64 位系统中，可以通过使用 64 位整数而不是两个 32 位整数表示 `double` 来避免这种情况。

访问双精度浮点数中的 32 位的另一个问题是，它不能移植到大端存储的系统中。因此，如果要具有大端存储的其他平台上实现，**示例14.23b** 和 **14.30** 将需要修改。所有 x86 平台（Windows、Linux、BSD、基于 Intel CPU 的 Mac OS等）都使用小端存储，但其他系统可能使用大端存储（如 PowerPC）。

我们可以通过比较 32 - 62 位来近似比较双精度浮点数。这在高斯消元法中求矩阵中值最大的主元是很有用的。例14.28 中的方法在主元搜寻中可以这么使用：

```
// Example 14.30
const int size = 100;
// Array of 100 doubles:
union {double d; unsigned int u[2]} a[size];
unsigned int absvalue, largest_abs = 0;
int i, largest_index = 0;
for (i = 0; i < size; i++)
{
    // Get upper 32 bits of a[i] and shift out sign bit:
    absvalue = a[i].u[1] * 2;
```

```
    // Find numerically largest element (approximately):
    if (absvalue > largest_abs)
    {
        largest_abs = absvalue;
        largest_index = i;
    }
```

例 14.30 找到数组中 数字（除去符号位）最大（或差不多最大的）的元素。它可能无法区分相对差小于2^{-20}的元素，但这对于寻找合适的主元来说是足够准确的。整数比较可能比浮点比较更快。在大的端系统中，你必须用 `u[0]` 替换 `u[1]`。

14.10 数学函数

最常见的数学函数如对数、指数函数、三角函数等都是在 x86 CPU 的硬件中实现的。然而，在大多数情况下，当 SSE2 指令集可用时，软件实现比硬件实现更快。如果启用了 SSE2 指令集，好的编译器将使用软件实现。

使用这些函数的软件实现而不是硬件实现的优势对于单精度比对于双精度更大。但在大多数情况下，软件实现要比硬件实现快，即使对于双精度也是如此。

通过包含与 Intel C++ 编译器一起提供的库：*libmmt.lib* 和头文件 *mathimf.h*，您可以在不同的编译器中使用 Intel 数学函数库。这个库包含许多有用的数学函数。*Intel's Math Kernel Library* 提供了许多高级数学函数，可以从www.intel.com获得（可参见第 122 页）。AMD 数学核心库包含类似的功能，但优化的较差。

注意，当在非 Intel 处理器上运行时，Intel 函数库没有使用最好的指令集（有关如何克服这个限制，请参阅第133页）。

14.11 静态库 VS 动态库

函数库可以实现为静态链接库（*.ilb，*.a），或动态链接库，也称为共享对象（.dll，*.so）。静态链接的机制是链接器从库文件中提取所需的函数并将它们复制到可执行文件中。只需要将可执行文件分发给最终用户。

动态链接的工作方式则不同。动态库中函数的链接在加载库或运行时解析。因此，当程序运行时，可执行文件和一个或多个动态库都被会加载到内存中。可执行文件和所有动态库都需要分发给最终用户。

静态链接相对于动态链接的优点是：

1. 使用静态链接，应用程序只需要包含库所需要的部分，而使用动态链接则需要将整个库（或至少库的大部分）加载到内存中，即使只需要库中的一个函数。
2. 当使用静态链接时，所有代码都包含在一个可执行文件中。而使用动态链接使得程序启动时必须加载多个文件。
3. 调用动态库中的函数要比调用在静态链接库中的花费更长的时间，因为它需要通过导入表中的指针进行额外的跳转，还可能需要在过程链接表（PLT）中进行查找。
4. 当代码分布在多个动态库之中时，内存空间变得更加碎片化。动态库加载在可被内存页大小(4096)整除的圆形内存地址（round memory addresses，这是啥？）处。这将使所有动态库争用相同的高速缓存线路。这降低了代码缓存和数据缓存的效率。
5. 动态库在某些系统中效率可能会较低，因为需要位置无关代码（参见下面的内容）。
6. 如果使用动态链接，安装使用相同动态库的更新版本的第二个应用程序，可以更改第一个应用程序的行为，但是如果使用静态链接，则不能更改第一个应用程序的行为。

使用动态链接的优点是：

1. 同时运行的多个应用程序可以共享相同的动态库，无需将库的多个实例加载到内存中。这适用于同时运行多个进程的服务器。实际上，只有代码节和只读数据节可以共享。任何可写数据部分，每个进程都需要一个单独的实例。
2. 无需更新调用程序，动态链接库就可以更新到新的版本。
3. 动态链接库可以被不支持静态链接的编程语言调用。
4. 使用动态链库可以用于为已有程序制作插件来添加新的功能。

权衡每种方法的上述优点，显然静态链接更适合于速度关键型函数。许多函数库都有静态和动态版本。如果速度很重要，则建议使用静态版本。

有些系统允许函数调用的延迟绑定。延迟绑定的原则是，在加载程序时不解析链接函数的地址，而是等到第一次调用该函数时才解析。延迟绑定对于大型库非常有用，因为在大型库中，在单个会话中实际调用的函数很少。但是延迟绑定肯定会降低所调用函数的性能。当一个函数第一次被调用时，由于它需要加载动态链接器，会出现相当大的延迟。

延迟绑定造成的延迟会导致交互程序的可用性问题，因为单击菜单的响应时间变得不一致，有时长得令人无法接受。因此，延迟绑定应该只用于非常大的库。

无法预先确定加载动态库的内存地址，因为固定地址可能与另一个需要相同地址的动态库冲突。有两种常用的方法来处理这个问题：

1. 重定位。如果需要，代码中的所有指针和地址都会被修改，以适应实际的加载地址。重定位由链接器和加载器完成。
2. 位置无关代码。代码中的所有地址都是相对于当前位置的。

在Windows中，dll 使用重定位。链接器将 dll 重新定位到特定的加载地址。如果这个地址不是空的，那么 dll 将被加载程序重新定位（rebase）到另一个地址。在主可执行文件中调用 dll 中的函数要经过导入表或指针。dll 中的变量可以通过 `main` 函数中导入的指针来访问（A variable in a DLL can be accessed from main through an imported pointer），但是很少使用这个特性。通过函数调用来交换数据或指向数据的指针更为常见。对 dll 内数据的内部引用在 32 位模式下使用绝对引用，在 64 位模式下使用相对引用。后者的效率略微高一点，因为相对引用在加载时不需要重新定位。

共享对象在类 Unix 系统中默认使用位置无关代码。这比重定位的效率要低，尤其是在 32 位模式下。下一章将描述这是如何工作的，并提出避免位置无关代码成本的方法。

14.12 位置无关代码

Linux、BSD 和 Mac 系统中的共享对象通常使用所谓的位置无关代码。“位置无关代码”的名称实际上比它所表达的含义更丰富。编译为位置无关的代码具有以下特性：

1. 代码部分不包含需要重新定位的绝对地址，只包含自相关地址。因此，代码段可以在任意内存地址加载，并在多个进程之间共享。
2. 数据部分不会在多个进程之间共享，因为它通常包含可写数据。因此，数据部分可能包含需要重新定位的指针或地址。
3. 在 Linux 和 BSD 中，所有公共函数和公共数据都可以被覆盖。如果主可执行文件中的函数与共享对象中的函数具有相同的名称，那么不仅在主可执行文件调用时，而且在从共享对象调用时，主可执行文件中的版本都将是优先的。同样的，当主可执行文件中的全局变量具有与共享对象中的全局变量相同的名称时，即使是从共享对象访问，也将使用主可执行文件中的实例。这种所谓的符号插入是为了模拟静态库的行为。为了实现这个“覆盖”特性，共享对象有一个指向其函数的指针表，称为过程链接表（PLT）和一个指向其变量的指针表，称为全局偏移表（GOT）。所有对函数和公共变量的访问都要经过 PLT 和 GOT。

允许在 Linux 和 BSD 中重写公共函数和数据的符号插入特性代价高昂，而且在大多数库中从未使用过。每当调用共享对象中的函数时，都需要在过程链接表（PLT）中查找函数地址。当访问共享对象中的公共变量时，则需要先在全局偏移表（GOT）中查找该变量的地址。即使访问同一个共享对象中访问函数或变量，也需要这些查找表。显然，所有这些表查找操作都会大大降低执行速度。更详细的讨论可以在 https://www.macieira.org/blog/2012/01/sorry-state-of-dynamic-libraries-on-linux/中找到。

另一个严重的负担是在 32 位模式下计算自相关引用。32 位 x86 指令集没有用于数据自相关寻址的指令。代码通过以下步骤访问公共数据对象：（1）通过函数调用获得其自身的地址。（2）通过一个自相关地址查找 GOT。（3）在 GOT 中查找数据对象的地址。最后，（4）通过这个地址访问数据对象。在 64 位模式下不需要步骤（1），因为 x86-64 指令集支持自相关寻址。

在 32 位 Linux 和 BSD 中，所有静态数据都使用较慢的 GOT 查找过程，包括不需要“覆盖”特性的本地数据。这包括静态变量、浮点常量、字符串常量和初始化过的数组（initialized arrays）。我无法解释为什么不必要的时候使用这种延迟很高的流程。

显然，避免繁重的位置无关代码和表查找的最佳方法是使用静态链接，如前一节（第149页，TODO）所述。在无法避免动态链接的情况下，有多种方法可以避免位置无关代码的时间消耗特性。这些解决方法依赖于系统，如下所述。

<u>32 位 Linux 中的共享对象</u>

根据 Gnu 编译器手册共享对象通常都是使用 *-fpic* 选项编译的。该选项使代码段是位置无关的，为所有函数生成 PLT，为所有公共和静态数据生成 GOT。

不使用 *-fpic* 选项也可以编译共享对象。这样我们就可以摆脱了上面提到的所有问题。代码将运行得更快，因为我们可以访问内部变量和内部函数只需要一个步骤，而不是前面介绍的复杂的地址计算和表查找机制。在没有 *-fpic* 的情况下编译共享对象要快得多，除非是一个非常大的共享对象，而其中大多数函数都不会被调用。在 32 位 Linux 中不使用 *-fpic* 编译的缺点是加载器将有更多的引用需要重新定位，但是这些地址计算只执行一次，而在每次访问时必须执行运行时地址计算。在不使用 *-fpic* 选项的情况下编译代码部分时，每个进程都需要一个实例，因为代码部分中的重新定位对每个进程来说是不同的。显然，我们失去了覆盖公共符号的能力，但无论如何很少需要使用这个特性。

为了可以移植到64位模式，您最好避免全局变量或者隐藏它们，解释如下。

<u>64 位 Linux 中的共享对象</u>

在 64 位模式下，计算自相关地址的过程要简单得多，因为 64 位指令集支持数据的相对寻址。在 64 位模式下，由于默认使用相对地址，对特殊的位置无关代码需求更小。然而，我们仍然希望消除对本地引用的 GOT 和 PLT 查找。

在 64 位模式下，如果我们不使用 *-fpic* 选项编译共享对象，我们会遇到其它的问题。编译有时会使用 32 位的绝对地址，主要是静态数组。这在主可执行文件中是没有问题的，因为它肯定是在低于 2GB 的地址加载的，但对于共享对象则不是这样的，共享对象通常加载在 32 位（signed？）地址无法表示的较高地址。在这种情况下，连接器会产生一条错误信息。最佳的解决方案是使用 *-fpie* 选项而不是 *-fpic* 选项来编译。这将在代码部分生成相对地址，但对于内部引用它不会使用 GOT 和 PLT。因此，它将比用 *-fpic* 编译时运行得更快，并且对于32位的情况，它不会有上面提到的缺点。在32位模式下，*-fpie* 选项的作用没有那么大，因为它仍然使用 GOT。

另一种方法是使用 *-mcmodel=large* 选项编译，但这将对所有内容使用 64 位地址，这是非常低效的，而且它将在代码部分生成重定位，因此不能被共享。

使用 *-fpie* 选项时，在 64 位共享对象中，不可以有公共变量，因为当链接器看到一个公共变量的相对引用时，它会产生一个错误消息，因为它期望在这个公共变量有一个 GOT 入口。你可以通过避免任何公共变量来避免该错误。所有全局变量（即定义在任何函数外部的变量）都应该使用声明 `static` 或 `__attribute__((visibility ("hidden"))` 来隐藏。

Gnu 编译器 5.1 及以后版本有一个选项：*-fno-semantic-interposition*，可以使它能够避免使用 *PLT* 和 *GOT*，但仅限于同一文件中的引用。通过使用内联汇编代码为变量提供两个名称，一个全局名称和一个本地名称，并使用本地名称作为本地引用，可以得到相同的效果。

尽管有这些技巧，当使用多个模块（源文件）生成共享对象时，并且存在一个模块调用另一个模块时，您可能仍然会得到错误消息:“ "relocation R_X86_64_PC32 against symbol `functionname' can not be used when making a shared object; recompile with -fPIC"。我至今没有找到该问题的解决方法。

BSD 中的共享变量 BSD 中的共享对象与 Linux 中的工作方式相同。

<u>32-bit Mac OS X</u>

32 位 Mac OS X 的编译器默认情况下使位置无关代码和延迟绑定，即使不使用共享对象。目前在 32-bit Mac 代码中用于计算自相关地址的方法使用了一种不幸的方法，它会导致错误地预测返回地址，从而延迟执行（有关返回预测的解释，请参阅手册3:“The microarchitecture of Intel, AMD and VIA CPUs”）。

只要在编译器中关闭与位置无关代码的标志，就可以显著加速不属于共享对象的所有代码。因此，请记住，在为 32-bit Mac OS X 编译时，总是要指定编译器选项 *-fno-pic*，除非您正在创建一个共享对象。

使用选项 *-fno-pic* 编译共享对象并使用选项 *-read_only_relocs suppress* 链接共享对象时，可以不使用位置无关代码。

对于内部引用不会使用 GOT 和 PLT。

<u>64-bit Mac OS X</u>

代码部分始终与位置无关，因为这是这里使用的内存模型的最有效的解决方案。编译器选项 *-fno-pic* 显然没有效果。

对于内部引用不会使用 GOT 和 PLT。

在 Mac OS X 中，不需要采取特别的预防措施来加速 64 位共享对象。

14.13 系统编程

设备驱动程序、中断服务路由、系统核心和高优先级线程是速度特别关键的地方。在系统代码或高优先级线程中非常耗时的函数可能会阻塞其他所有内容的执行。

系统代码必须遵守寄存器使用的某些规则，如手册5中的“Calling conventions for different C++ compilers and operating systems”中 ”内核代码中的寄存器用法”一章所述。因此，您只能使用针对系统代码的编译器和函数库。系统代码应该使用 C、C++ 或汇编语言编写。

在系统代码中节约资源的使用是非常重要的。动态内存分配特别有风险，因为它涉及在不方便的时候激活非常耗时的垃圾收集器的风险。队列应该实现为固定大小的循环缓冲区，而不是链表。不要使用STL容器。见92页（TODO）。

15 元编程

元编程意味着编写生成代码的代码。例如，在解释脚本语言中，通常可以编写一段生成字符串的代码段，然后将该字符串解释为代码。

如果计算的所有输入在编译时都可用，元编程在编译语言（如 C++）中非常有用，可以在编译时期而不是运行时期做一些计算。（当然，在所有事情都在运行时发生的解释语言中，则没有这样的优势）。

在 C++ 中，可以考虑使用以下技术进行元编程：

1. 预处理指令。例如使用 `#if` 代替 `if`。这是一个移除无效代码的有用方法，但是，由于预处理器先于编译器，并且只理解最简单的表达式和运算符，所以它所能做的工作受到了严重的限制。
2. 编写一个 C++ 程序，生成另一个 C++ 程序（或它的一部分）。在某些情况下，这可能很有用，例如生成最终程序中作为静态数组的数学函数表。当然，这需要编译第一个程序的输出。
3. 编译器优化可能会在编译时尽可能多地执行操作。例如，所有好的编译器都会将 `int x = 2 * 5` 化简位 `int x = 10`;
4. 模板在编译时实例化。在编译模板实例之前，将其参数替换为它们的实际值。这就是为什么使用模板实际上没有成本的原因（见第58页，TODO）。使用模板元编程可以表达任何算法，但是这种方法非常复杂和笨拙，稍后您就会看到。

下面的例子解释了当指数是编译时已知的整数时，如何使用元编程来加速幂函数的计算。

```
// Example 15.1a. Calculate x to the power of 10

double xpow10(double x)
{
    return pow(x,10);
```

在一般情况下，`pow` 函数使用对数，但在上面这种情况下，它将识别到 10 是整数，因此结果只能使用乘法计算。当指数为正整数时，在 `pow` 函数中使用以下算法：

```
// Example 15.1b. Calculate integer power using loop

double ipow (double x, unsigned int n)
{
    double y = 1.0; // used for multiplication
    while (n != 0)
    {
        // loop for each bit in nn
        if (n & 1)
            y *= x; // multiply if bit = 1
        x *= x; // square x
        n >>= 1; // get next bit of n
    }
    return y; // return y = pow(x,n)
}
double xpow10(double x)
{
    return ipow(x,10); // ipow faster than pow
```

当我们展开循环并重新组织时，**例15.1b** 中使用的方法将更容易理解：

```
// Example 15.1c. Calculate integer power, loop unrolled

double xpow10(double x)
{
    double x2 = x *x; // x^2
    double x4 = x2*x2; // x^4
    double x8 = x4*x4; // x^8
    double x10 = x8*x2; // x^10
    return x10; // return x^10
```

正如我们所看到的，只需要四次乘法就可以计算出 `pow(x,10)`。那怎么才能将 **例 15.1b** 转换到 **例 15.1c**呢？我们利用了在编译时已知 `n` 的事实，消除了只依赖于 `n` 的所有内容，包括 `while` 循环、`if` 语句和所有整数计算。**例 15.1c**中的代码比 **15.1b** 更快，在这种情况下，它可能也更小。

从 **例15.1b** 到 **15.1c** 的转换是由我手动完成的，但是如果我们想生成一段代码，使它可以用于编译时已知的常量 `n`，那么我们需要元编程。我测试过的所有编译器都不能自动将 **例15.1a** 转换为 **15.1c**，只有 **Gnu** 编译器才能将 **例15.1b** 转换为 **15.1c**。我们只能希望将来的编译器能够自动进行这样的优化，但只要不是这样，我们就可能需要元编程。

下一个示例显示使用模板元编程来实现计算。如果你不懂也不要惊慌。我给出这个示例只是为了说明模板元编程是多么复杂。

```
// Example 15.1d. Integer power using template metaprogramming

// Template for pow(x,N) where N is a positive integer constant.
// General case, N is not a power of 2:
template <bool IsPowerOf2, int N>
class powN
```

```
{
public:
    static double p(double x) {
    // Remove right-most 1-bit in binary representation of N:
    #define N1 (N & (N-1))
    return powN<(N1&(N1-1))==0,N1>::p(x) * powN<true,N-N1>::p(x);
    #undef N1
    }
};

// Partial template specialization for N a power of 2
template <int N>
class powN<true,N>
{
public:
    static double p(double x)
    {
        return powN<true,N/2>::p(x) * powN<true,N/2>::p(x);
    }
};

// Full template specialization for N = 1. This ends the recursion
template<>
class powN<true,1>
{
public:
    static double p(double x)
    {
        return x;
    }
};

// Full template specialization for N = 0
// This is used only for avoiding infinite loop if powN is
// erroneously called with IsPowerOf2 = false where it should be true.
template<>
class powN<true,0>
{
public:
    static double p(double x)
    {
        return 1.0;
    }
};

// Function template for x to the power of N
template <int N>
static inline double IntegerPower (double x)
{
    // (N & N-1)==0 if N is a power of 2
    return powN<(N & N-1)==0,N>::p(x);
}

// Use template to get x to the power of 10
double xpow10(double x)
{
    return IntegerPower<10>(x);
```

如果你想知道这是怎么回事，请看下面的解释。如果您不确定是否需要，可以跳过下面的解释。

在 C++ 模板元编程中，循环被实现为递归模板。`powN` 模板正在调用自己，以便模拟 **例15.1b** 中的 `while` 循环。分支是通过（部分）模板特化实现的， 这就是对 **例15.1b**中的 `if` 分支的实现。递归必须始终以非递归模板特化结束，而不是在模板中包含分支。

`powN` 模板是类模板而不是函数模板，因为只允许对类进行部分模板特化。将 `N` 分解成二进制表示的各个位是非常需要技巧的的。我使用的技巧是 `N1 = N&(N-1)` 给得到 `N` 的去掉最右边的 1 位的值。如果 `N` 是 2 的幂，那么 `N&(N-1)` 为 0。常量 `N1` 可以用其他方法定义，而不是只能宏定义，但是这里使用的方法是我尝试过的所有编译器中唯一全部适用的方法。

微软、英特尔和 Gnu 编译器实际上按照预期地将 **例15.1d** 化简到 **15.1c**，而 Borland 和 Digital Mars 编译器产生的代码不太理想，因为它们无法消除公共子表达式。

为什么模板元编程如此复杂？因为 C++ 的模板特性从来不是为此目的设计。这只是碰巧可行。模板元编程非常复杂，我认为使用它是不明智的。复杂的代码本身就是一个风险，而且验证、调试和维护这些代码的成本非常高，因此很少有理由在获得相对较小的性能收益时使用它。

然而在某些情况下，模板元编程是确保在编译时完成某些计算的唯一方法。（可以在我的 vector class library 中找到例子）。

D 语言允许*编译时 if* 语句（称为*静态 if*），但不允许编译时循环或编译时生成标识符名称。我们只能希望这样的功能在将来能够实现。如果 C++ 的未来版本应该会允许 *编译时 If* 和*编译时 while 循环*，那么将 例15.1b*转换为元编程将非常简单。MASM 汇编语言具有完整的元编程特性，包括通过字符串函数来定义函数名和变量名的能力。在手册2"Optimizing subroutines in assembly language"的"宏循环"一节中，提供了一个类似于*"例 15.1b**和 **例 15.1d**的使用汇编语言的元编程实现。

当我们在等待更好的元编程工具出现时，我们可以选择那些最擅长在任何可能的情况下自动进行等价化简的编译器。使用自动将 **例15.1a** 化简到**15.1c**的编译器当然是最简单和最可靠的解决方案。(在我的测试中，Intel 编译器将**15.1a** 化简为内联的**15.1b**， Gnu 编译器将**15.1b**化简为**15.1c**，但是没有一个编译器能将**15.1a**化简为**15.1c**）。

16 测试速度

测试程序的速度是优化工作的重要组成部分。你必须检查你的修改是否真的提高了速度。

有多种可用的分析器，它们对于查找热点和测量程序的总体性能非常有用。然而，分析器并不总是准确的，而且当程序花费大部分时间等待用户输入或读取磁盘文件时，可能很难准确地测量您需要的是什么。有关分析的讨论请参见第16页（TODO）。

当确定了热点之后，隔离热点并仅对代码的这一部分进行测量可能是有用的。这可以通过使用所谓的时间戳计数器来获得 CPU 时钟的分辨率来实现。这是一个计数器，用来测量 CPU 启动以来的时钟脉冲数。时钟周期的长度是时钟频率的倒数，如第16页（TODO）所述。如果您在执行一段关键代码之前和之后读取时间戳计数器的值，那么您可以得到确切的时间消耗，即两个时钟计数之间的差值。

使用**例16.1**中列出的函数 `ReadTSC` 可以获得时间戳计数器的值。此代码仅适用于支持指令集函数的编译器。或者，您可以使用www.agner.org/optimize/testp.zip中的头文件 **timingtest.h**，或者从www.agner.org/optimize/asmlib.zip获得 `ReadTSC` 作为库函数来使用。

```
// Example 16.1

#include <intrin.h> // Or #include <ia32intrin.h> etc.
long long ReadTSC()
{
    // Returns time stamp counter
    int dummy[4]; // For unused returns
    volatile int DontSkip; // Volatile to prevent optimizing
    long long clock; // Time
    __cpuid(dummy, 0); // Serialize
    DontSkip = dummy[0]; // Prevent optimizing away cpuid
    clock = __rdtsc(); // Read time
    return clock;
```

您可以使用此函数来测量执行关键代码前后的时钟计数。测试设置可能是这样的：

```
// Example 16.2

#include <stdio.h>
#include <asmlib.h> // Use ReadTSC() from library asmlib..
                    // or from example 16.1
void CriticalFunction(); // This is the function we want to measure
...
const int NumberOfTests = 10; // Number of times to test
```

```
int i; long long time1;
long long timediff[NumberOfTests]; // Time difference for each test
for (i = 0; i < NumberOfTests; i++)
{
    // Repeat NumberOfTests times
    time1 = ReadTSC(); // Time before test
    CriticalFunction(); // Critical function to test
    timediff[i] = ReadTSC() - time1; // (time after) - (time before)
}
printf("\nResults:"); // Print heading
for (i = 0; i < NumberOfTests; i++)
{
    // Loop to print out results
    printf("\n%2i %10I64i", i, timediff[i]);
```

例16.2中的代码调用关键函数十次，并将每次运行的时间消耗存储在一个数组中。然后在测试循环之后输出这些值。以这种方式测量的时间包括调用 `ReadTSC` 函数所需的时间。你可以从计数中减去这个值。这个值简单地通过移除**例16.2**中的 `CriticalFunction` 函数的调用来测量。

测量的时间按以下方式解释。第一次调用地计数通常高于随后的数。这是当代码和数据没有被缓存时执行 `CriticalFunction` 函数所需要的时间。随后的计数给出当代码和数据被经可能缓存好时所需的执行时间 。第一个计数和随后的计数分别表示"最坏情况"和"最佳情况"的值。这两个值中哪一个最接近真实情况取决于最终程序中对 `CriticalFunction` 函数的调用一次还是多次，以及对 `CriticalFunction` 调用之间是否有其他代码使用缓存。如果您的优化工作集中在 CPU 效率上，那么它是"最好的情况"就很重要，您应该看看某个修改是否有利可图。另一方面，如果您的优化工作集中于按顺序排列数据以提高缓存效率上，然后您还可以查看"最坏情况"下的计数。在任何情况下，典型应用程序中，用户可能经过的时间延迟应该时这么计算的：时钟计数 * 时钟周期 * 调用 `CriticalFunction` 函数的次数。

有时候，你测量的时钟计数比正常情况下要高得多。当在 `CriticalFunction` 函数执行期间发生任务切换时，就会发生这种情况。您无法在受保护的操作系统中避免这种情况，但是您可以通过在测试前增加线程优先级并在测试后将优先级设置为正常来减少这个问题的发生。

时钟计数经常波动，测试结果的可重复性可能不是很好。这是因为现代 CPU 可以根据工作负载动态地改变时钟频率。工作负荷大时时钟频率增大，工作负荷小时时钟频率减小，以节约电能。有多种方法可以获得可重复的时间测量值：

1. 通过在测试代码之前给 CPU 一些繁重的工作来预热 CPU。
2. 禁用 BIOS 设置中的省电选项。
3. 在 Intel CPU 上：使用内核时钟周期计数器（见下面内容）。

16.1 使用性能监视器计数器

许多 CPU 都有一个内置的测试特性，称为性能监视计数器。性能监视器计数器是 CPU 中的一个计数器，可以设置它来计数某些事件，比如执行的机器指令数量、缓存丢失、分支错误预测等。这些计数器对于研究性能问题非常有用。性能监视计数器是特定于 CPU 的，每个 CPU 模型都有自己的一组性能监视参数。

CPU 厂商会提供适合他们 CPU 的分析工具。英特尔的分析器叫做 *VTune*；AMD 的分析器叫做 *CodeAnalyst*。这些分析器对于识别代码中的热点非常有用。

在我自己的研究中，为了使用性能监视器计数器，我开发了一个测试工具。我的测试工具同时支持 Intel、AMD 和 VIA 处理器，可以从www.agner.org/optimize/testp.zip获得。这个工具不是分析器。它不是用于寻找热点的，而是用于在确定了热点之后研究代码段。

我的测试工具可以以两种方式使用。第一种方法是将要测试的代码插入测试程序本身并重新编译它。我使用它来测试单个汇编指令或小段代码。第二种方法是在运行要优化的程序之前设置性能监视器计数器，并在要测试的代码段之前和之后读取程序内部的性能计数器。您可以使用与上面**例16.2**相同的原则，但是读取一个或多个性能监视器计数器，替换（除了）时间戳计数器。测试工具可以在所有 CPU 内核中设置并启用一个或多个性能监视器计数器，并保持启用它们（每个CPU内核中有一组计数器）。计数器会一直开着，直到你关掉它们，或者直到电脑重置或进入睡眠模式。有关详细信息，请参阅我的测试工具手册（www.agner.org/optimize/testp.zip）。

英特尔处理器中一个特别有用的性能监视器计数器称为核心时钟周期计数器。核心时钟周期计数器是按照 CPU 核心运行时的实际时钟频率而不是外部时钟计算时钟周期的。这给出了一个几乎与时钟频率变化无关的度量。当测试一段代码的哪个版本最快时，核心时钟周期计数器非常有用，因为您可以避免时钟频率上升和下降的问题。

记得在程序中插入一个开关，以便在不测试时关闭计数器的读取。当性能监视器计数器被禁用时，试图读取它们将导致程序崩溃。

16.2 单元测试的陷阱

在软件开发中，通常单独测试每个函数或类。这种单元测试对于验证优化函数的功能是必要的，但是不幸的是，单元测试并没有提供关于函数性能在速度方面的全部信息。

假设你有两个不同版本的关键函数，你想找出哪个是最快的。测试这一点的典型方法是编写一个小型测试程序，使用一组合适的测试数据多次调用关键函数，并测量所需的时间。在此单元测试下性能最好的版本可能比其他版本占用更大的内存。在单元测试中看不到缓存命中失败的损失，因为测试程序使用的代码和数据内存总量可能小于缓存大小。

当在最终程序中插入关键函数时，代码缓存和数据缓存很可能是关键资源。现代 CPU 的速度如此之快，以至于时钟周期花费在执行指令上不太可能像内存访问和缓存大小那样成为瓶颈。如果是这种情况，那么关键函数的最佳版本可能是单元测试中花费更长的时间但内存占用更小的版本。

例如，如果您想知道展开一个大循环是否有利的，那么您不能依赖单元测试而不考虑缓存效果。

通过为链接器使用"生成映射文件"选项，您可以查看链接映射或汇编代码列表来计算函数使用了多少内存。代码缓存使用和数据缓存使用都很重要。分支目标缓冲区也是一个关键的缓存。因此，还应该考虑函数中跳转、调用和分支的数量。

一个实际的性能测试不仅应该包含单个函数或热点，还应该包含包含关键函数和热点的最内层循环。应该使用一组真实的数据来进行测试，以便为分支错误预测获得可靠的结果。性能度量不应该包括程序中等待用户输入的任何部分。用于文件输入和输出的时间应该分开测量。

不幸的是，用单元测试来度量性能的谬论非常普遍。即使是一些最佳优化的函数库也会使用过多的循环展开，因此内存占用非常大。

16.3 最差条件测试

大多数性能测试都是在最佳条件下进行的。消除了所有干扰的影响，所有资源都是充足的，缓存条件是最优的。最佳条件测试是有用的，因为它提供了更可靠和可重复的结果。如果您想比较同一算法的两种不同实现的性能，那么您需要消除所有干扰影响，以便使测量尽可能准确和可重现。

然而，在某些情况下，在最坏的情况下测试性能更为相关。例如，如果您想确保对用户输入的响应时间永远不会超过可接受的范围，那么您应该在最坏情况下测试响应时间。

产生流媒体音频或视频的程序也应该在最坏的情况下进行测试，以确保它们始终保持预期的实时速度。输出中的延迟或故障是不可接受的。

在测试最坏情况下的性能时，以下每一种方法都可能是相关的：

1. 第一次激活程序的某个特定部分时，由于代码的延迟加载、缓存未命中和分支预测错误，它可能比之后的速度慢。
2. 测试整个软件包，包括所有运行时库和框架，而不是隔离单个函数。在软件包的不同部分之间切换，以增加程序代码的某些部分未被缓存或甚至被交换到磁盘的可能性。
3. 依赖于网络资源和服务器的软件应该在流量较大的网络和被充分使用的服务器上进行测试，而不是专用的测试服务器。
4. 使用包含大量数据的大型数据文件和数据库。
5. 使用 CPU 速度慢、RAM 不足、安装了大量无关软件、运行了大量后台进程、硬盘速度慢且碎片化的旧计算机。
6. 使用不同品牌的 CPU、不同类型的显卡等进行测试。
7. 使用杀毒程序，扫描所有文件的访问。
8. 同时运行多个进程或线程。如果微处理器支持超线程，那么尝试在同一个处理器内核中运行两个线程。
9. 尝试分配比现有内存更多的 RAM，以便强制将内存交换到磁盘。
10. 通过使最内层循环中使用的代码大小或数据大于缓存大小来触发缓存未命中。或者，您可以主动地使缓存失效。操作系统可能有一个用于此目的的函数，或者您可以使用指令集函数 `_mm_clflush`。
11. 使数据比正常情况更随机，从而引发分支错误预测。

17 在嵌入式系统中优化

在小型嵌入式应用程序中使用的微控制器比标准 PC 拥有更少的计算资源。时钟频率可以低 100 倍甚至 1000 倍；而且 RAM 内存的数量甚至可能比 PC 少一百万倍。尽管如此，如果您避免使用大型图形框架、解释器、即时编译器、系统数据库以及通常用于大型系统的其他额外软件层和框架，则可以使软件在这样的小型设备上运行得相当快。

系统越小，选择一个占用较少资源的软件框架就越重要。在最小的设备上，甚至没有操作系统。）

可以通过选择可以在 PC 上交叉编译的编程语言来获得最佳的性能。任何要求在目标设备上编译或者解释的语言都会对资源产生极大的浪费。：由于这些原因，首选的语言通常是 C 或 C++。关键设备驱动程序可能需要汇编语言。

如果遵循下面的指导原则，C++ 只需要比 C 多一点点的资源。您可以根据最适合所需程序结构的方式选择 C 或 C++。

节约内存的使用是很重要的。大数组应该在函数中声明，以便在函数返回时释放它们。或者，您可以将相同的数组重用于多个目的。

应该避免所有使用 `new` / `delete`或 `malloc` / `free`的动态内存分配，因为管理内存堆的开销很大。堆管理器有一个垃圾收集器，它可能以不可预测的间隔消耗时间，这可能会干扰实时应用程序。

请记住，STL （标准模板库）和其他容器类库中的容器类使用 `new` 和 `delete` 来动态内存，而且常常过多地使用动态内存分配。除非您有足够的资源，否则绝对应该避免使用这些容器。例如，FIFO 队列应该被实现为一个固定大小的循环缓冲区，以避免动态内存分配。不要使用链表（请参阅第95页，TODO）。

字符串类的所有常见实现都使用动态内存分配。您应该避免这些，并以老式 C 风格使用字符数组处理字符串。注意，C 风格的字符串函数不会检查数组是否溢出。程序员需要确保数组足够大，可以处理字符串（包括终止符 0），并在必要时进行溢出检查（参见第98页，TODO）。

C++ 中的虚函数比非虚函数占用更多的资源。尽可能避免使用虚函数。

较小的微处理器没有本地浮点执行单元。此类处理器上的任何浮点运算都需要一个很大的浮点库，这非常耗时。因此，应该避免使用浮点表达式。例如，`a = b * 2.5` 可能改为`a = b * 5 / 2`（注意中间表达式 `b * 5` 可能会溢出）。只要程序中有一个浮点常量，就会加载整个浮点库。如果你想用两个小数来计算一个数字，那么你应该把它乘以100，这样它就可以表示为一个整数。

整数变量可以是 8 位、16 位或 32 位（很少有 64 位）。如果需要，可以使用不会导致特定应用程序溢出的最小整数大小来节省 RAM 空间。整数大小没有跨平台标准化。有关每种整数类型的大小，请参阅编译器文档。

中断服务程序和设备驱动程序尤其重要，因为它们可以阻止其他所有东西的执行。这通常属于系统编程领域，但在没有操作系统的应用程序中，这是应用程序程序员的工作。当没有操作系统时，程序员更容易忘记系统代码是很关键的，因此系统代码没有与应用程序代码分离。中断服务程序应该做尽可能少的工作。通常，它应该将接收到的数据的一个单元保存在静态缓冲区中，或者从缓冲区发送数据。它永远不应该响应某个命令，或者执行它所服务的特定事件之外的其他输入/输出。中断接收到的命令最好以较低的优先级响应，通常在主程序的消息循环中响应。有关系统代码的进一步讨论，请参见第153页（TODO）。

在本章中，我描述了一些对资源有限的小型设备特别重要的考虑。本手册其余部分的大部分建议也与小型设备有关，但由于小型微控制器的设计会存在一些差异：

1. 较小的微控制器没有分支预测（见第43页，TODO）。软件中不需要考虑分支预测。
2. 较小的微控制器没有缓存（见第89页，TODO）。不需要组织数据来优化缓存。
3. 较小的微控制器没有无序执行。没有必要打破依赖链（见第22页，TODO）。

18 编译器选项一览

Table 18.1. Command line options relevant to optimization

	MS compiler Windows	Gnu compiler Linux	Intel compiler Windows	Intel compiler Linux
Optimize for speed	/O2 or /Ox	-O3 or -Ofast	/O3	-O3
Interprocedural optimization	/Og			
Whole program optimization	/GL	--combine -fwhole-program	/Qipo	-ipo
No exception handling	/EHs			
No stack frame	/Oy	-fomit-frame-pointer		-fomit-frame-pointer
No runtime type identification (RTTI)	/GR–	-fno-rtti	/GR-	-fno-rtti
Assume no pointer aliasing	/Oa			-fno-alias
Non-strict floating point		-ffast-math	/fp:fast /fp:fast=2	-fp-model fast, -fp-model fast=2
Simple member pointers	/vms			
Fastcall functions	/Gr			
Function level linking (remove unreferenced functions)	/Gy	-ffunction-sections	/Gy	-ffunction-sections

	MS compiler Windows	Gnu compiler Linux	Intel compiler Windows	Intel compiler Linux
SSE instruction set (128 bit float vectors)	/arch:SSE	-msse	/arch:SSE	-msse
SSE2 instruction set (128 vectors of integer or double)	/arch:SSE2	-msse2	/arch:SSE2	-msse2
SSE3 instruction set		-msse3	/arch:SSE3	-msse3
Suppl. SSE3 instr. set		-mssse3	/arch:SSSE2	-mssse3
SSE4.1 instr. set		-msse4.1	/arch:SSE4.1	-msse4.1
AVX instr. set	/arch:AVX	-mAVX	/arch:AVX	-mAVX
Automatic CPU dispatch			/QaxSSE3, etc. (Intel CPU only)	-axSSE3, etc. (Intel CPU only)
Automatic vectorization		-O3 -fno-trapping-math -fno-math-errno -mveclibabi		
Automatic paralleli-zation by multiple threads			/Qparallel	-parallel
Parallelization by OpenMP directives	/openmp	-fopenmp	/Qopenmp	-openmp
32 bit code		-m32		
64 bit code		-m64		
Static linking	/MT	-static	/MT	-static
(multithreaded)				
Generate assembly listing	/FA	-S -masm=intel	/FA	-S
Generate map file	/Fm			
Generate optimization report			/Qopt-report	-opt-report

Table 18.2. Compiler directives and keywords relevant to optimization

	MS compiler Windows	Gnu compiler Linux	Intel compiler Windows	Intel compiler Linux
Align by 16	`__declspec(align(16))`	`__attribute((aligned(16)))`	`__declspec(align(16))`	`__attribute((aligned(16)))`
Assume pointer is aligned			`#pragma vector aligned`	`#pragma vector aligned`
Assume pointer not aliased	`#pragma optimize("a", on) __restrict`	`__restrict`	`__declspec(noalias) __restrict #pragma ivdep`	`__restrict #pragma ivdep`
Assume function is pure		`__attribute((const))`		`__attribute((const))`

	MS compiler Windows	Gnu compiler Linux	Intel compiler Windows	Intel compiler Linux
Assume function does not throw exceptions	`throw()`	`throw()`	`throw()`	`throw()`
Assume function called only from same module	`static`	`static`	`static`	`static`
Assume member functioncalled only fromsame module		`__attribute__((visibility ("internal")))`		`__attribute__((visibility ("internal")))`
Vectorize			`#pragma vector always`	`#pragma vector always`
Optimize function	`#pragma optimize (...)`			
Fastcall function	`__fastcall`	`__attribute((fastcall))`	`__fastcall`	
Noncached write			`#pragma vector nontemporal`	`#pragma vector nontemporal`

Table 18.3. Predefined macros

	MS compiler Windows	Gnu compiler Linux<\br>	Intel compiler Windows	Intel compiler Linux
Compiler identification	`MSC_VER` and not `__INTEL_COMPILER`	`__GNUC__` and not `_INTEL_COMPILER`	`__INTEL_COMPILER`	`__INTEL_COMPILER`
16 bit	not `_WIN32`	n.a.	n.a.	n.a.
platform				
32 bitplatform	not `_WIN64`		not `_WIN64`	
64 bit platform	`_WIN64`	`_LP64 _WIN64 _LP64`		
Windows platform	`_WIN32`		`_WIN32`	
Linux platform	n.a.	`__unix__ __linux__`		`__unix__ __linux__`
x86 platform	`_M_IX86`		`_M_IX86`	
x86-64 platform	`M_IX86` and `_WIN64`		`_M_X64`	`_M_X64`

19 文献

Agner Fog 的其它手册 本手册是五本系列中手册的第一本。有关手册列表，请参见第3页（TODO）。

关于代码优化的文献 Intel："Intel 64 and IA-32 Architectures Optimization Reference Manual"。developer.intel.com。许多用于在英特尔 CPU 优化 C++和汇编代码的建议。定期更新版本；

AMD："Software Optimization Guide for AMD Family 15h Processors"。 www.amd.com。许多用于在 AMD CPU 优化 C++和汇编代码的建议。定期更新版本；

Intel："Intel® C++ Compiler Documentation"。包含在英特尔 C++编译器中，可以从 www.intel.com 上找到。使用 Intel C++ 编译器优化特性的手册

维基百科关于编译器优化的文章。en.wikipedia.org/wiki/Compiler_optimization。

ISO/IEC TR 18015, "Technical Report on C++ Performance"。 www.openstd.org/jtc1/sc22/wg21/docs/TR18015.pdf。

OpenMP。www.openmp.org。用于并行处理的OpenMP指令的文档。

Scott Meyers: "Effective C++". Addison-Wesley. Third Edition, 2005; and "More Effective C++". Addison-Wesley, 1996。这两本书包含了许多关于高级c++编程的技巧，如何避免难以发现的错误，以及一些提高性能的技巧。

Stefan Goedecker and Adolfy Hoisie: "Performance Optimization of Numerically Intensive Codes", SIAM 2001。关于 C++ 和 Fortran 代码优化的高级书籍。主要关注具有大数据集的数学应用。涵盖个人电脑，工作站和科学向量处理器。

Henry S. Warren, Jr.: "Hacker's Delight". Addison-Wesley, 2003。包含许多位操作技巧。

Michael Abrash: "Zen of code optimization", Coriolis group books 1994。大部分已经过时了。

Rick Booth: "Inner Loops: A sourcebook for fast 32-bit software development", AddisonWesley 1997。大部分已经过时了。

微处理器文档 Intel: "IA-32 Intel Architecture Software Developer's Manual", Volume 1, 2A, 2B, and 3A and3B. developer.intel.com.

AMD: "AMD64 Architecture Programmer's Manual", Volume 1 - 5. www.amd.com。

网络论坛 一些互联网论坛和新闻组包含关于代码优化的有用讨论。参见www.agner.org/optimization和新闻组 comp.lang.asm.x86 的一些链接。

20 版权声明