1.
用C++优化软件
适用于Windows、Linux和Mac的优化指南
平台

作者阿格纳弗格。丹麦技术大学。最后更新于2024-03-15。

内容

1导言

本手册适用于想要使软件更快的高级程序员和软件开发人员。假设读者对C++程序有很好的了解-
minglanguage和对编译器工作原理的基本理解。选择C++语言作为本手册的基础，原因
见第页8以下。

这本手册主要基于我对编译器和微处理器如何工作的研究。这些建议基于英特尔、AMD和VIA的x86系列微处理器，包括64位版本。x86处理器用于最常见的平台
使用Windows、Linux、BSD和Mac OS X操作系统，尽管这些操作系统
系统也可以与其他微处理器和指令集一起使用。许多
建议也适用于其他平台和其他编译编程语言。

这是五本系列手册中的第一本：

1.用C++优化软件：Windows、Linux和Mac平台的优化指南。

2.在汇编语言中优化子程序：x86platforms的优化指南。

3.英特尔、AMD和威盛CPU的微体系结构：汇编程序员和编译器制造商的优化指南。

4.指令表：Intel、AMD和VIA CPU的指令延迟、吞吐量和微操作细分列表。

5.不同C++编译器和操作系统的调用约定。

这些手册的最新版本可从万维网。阿格纳。组织/优化。版权条件列于第页179以下。

满足于用高级语言制作软件的程序员只需要阅读第一本手册。接下来的几年是为那些想更深入地了解指令定时、汇编语言编程、编译器等技术细节的人准备的
技术和微处理器微体系结构。

请注意，我的优化手册被成千上万的人使用
世界。我只是没有时间回答每个人的问题。所以请不要把你的编程问题发给我。你不会得到任何答案。初学者是
我建议去别处寻找信息并获得大量的编程经验
在尝试本手册中的技术之前。互联网上有各种各样的论坛，如果你找不到编程问题的答案
答案在相关书籍和手册中。

我要感谢许多给我发来关于myoptimization手册的更正和建议的人。我总是很高兴收到新的信息。

1.1为什么软件通常很慢

我们仍然有许多软件产品的响应时间长得令人沮丧，这是一个悖论
几十年来微处理器性能呈指数级增长。在大多数
情况下，性能不令人满意的原因不是微处理器设计差，而是
糟糕的软件设计。通常，罪魁祸首是极其浪费的软件开发
工具、框架、虚拟机、脚本语言和bstractmany-layer软件
设计。根据摩尔定律，随着我们接近物理上可能的极限，硬件性能的增长正在缓慢下降。相反，沃思定律声称
开玩笑地说，软件速度下降的速度比硬件速度增加的速度快。

在这种情况下，建议软件开发人员改进软件，而不是依赖更快的微处理器：避免最浪费的软件工具和框架，并避免功能膨胀。降低软件开发中的抽象级别实际上会
更容易理解不同代码结构的性能后果，如本手册所述。

1.2优化的成本

如今的大学编程课程强调结构化和面向对象编程、模块化、可重用性、多层抽象和软件开发过程系统化的重要性。这些要求通常是
与优化软件速度或大小的要求相冲突。

今天，软件教师建议任何函数或met hod都不应该超过几行，这并不罕见。几十年前，推荐是
相反：如果某个东西只被调用一次，就不要把它放在一个单独的子例程中。The
软件写作风格转变的原因是软件项目变得更大更复杂，人们更加关注软件开发的成本，以及计算机变得更加强大。

结构化软件开发的高优先级与程序的低优先级
效率首先反映在编程语言的选择和
接口框架。对于最终用户来说，这通常是一个缺点，他们必须
越来越强大的计算机来跟上越来越大的软件包，但whois仍然对可接受的长响应时间感到沮丧，即使是简单的任务。

有时，为了使软件包更快、更小，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考虑因素之间取得合理的平衡。讨论了如何识别和分离程序中最关键的部分，并将优化效果集中在该部分上
特定部分。它讨论了如何克服相对原始的危险
不自动检查arraybounds冲突的编程样式，无效
指针等。并讨论了与执行时间相关的高级编程结构中哪些是昂贵的，哪些是便宜的。

2选择最优平台

2.1硬件平台rm的选择

硬件平台的选择变得不像以前那么重要了。The
RISC和CISC处理器、PC和大型机之间的区别，以及
简单处理器和矢量处理器之间的界限变得越来越模糊，因为具有CISC指令集的标准PC处理器具有gotRISC内核、矢量处理指令、多个内核，并且处理速度超过了昨天的大处理器
大型计算机。

如今，为给定任务选择硬件平台通常由以下因素决定
价格、兼容性、第二来源和商品的可用性等考虑因素
开发工具，而不是处理能力。在一个网络中连接几台标准PC可能比投资一台大型主机更便宜，也更有效
电脑。具有大规模并行向量处理能力的大型超级计算机仍然
在科学计算方面有一定的优势，但对于大多数目的，标准和PC处理器是首选，因为它们具有优越的性能/价格比。

从技术角度来看，标准PC处理器的CISC指令集（x86）并不是最佳的。维护此指令集是为了备份
与1980年左右的一系列软件兼容，当时RAM内存和磁盘空间是稀缺资源。然而，CISC指令集比它的名声更好。在高速缓存大小是有限资源的今天，代码的紧凑性使得计算更加高效。CISC指令集实际上可能比RISC更好
代码缓存至关重要的情况。顺式指令集的另一个优点是
做同样的工作需要更少的指令。原始X86指令集的一个严重问题是寄存器的稀缺。这个问题在64位中得到了缓解
扩展到x86指令集，其中通用寄存器的数量增加了一倍，向量寄存器的数量增加了四倍。

对于关键应用程序，不建议使用依赖于网络资源的瘦客户端，因为网络资源的响应时间无法控制。

小型手持设备正变得越来越受欢迎，并且用于越来越多的目的，例如电子邮件和网络浏览，这些以前需要aPC。同样，我们看到越来越多的设备和机器带有嵌入式微控制器。我
对于哪种平台和操作系统对此类应用程序最有效，我并没有提出任何具体的建议，但重要的是要认识到此类设备

通常具有比ANPC少得多的内存和计算能力。因此，它是均匀的
在这种系统上节约资源比在PC平台上节约资源更重要。然而，通过良好优化的软件设计，即使在如此小的设备上，也有可能为许多应用程序获得良好的性能，如第#页所述172.

本手册基于英特尔、AMD的标准PC平台，或通过处理器以及运行32位或64位模式的Windows、Linux、BSD或Mac操作系统。这里给出的许多建议可能也适用于其他平台，但是仅在PC平台上测试。

图形加速器

平台的选择显然受到所讨论任务的要求的影响。例如，重型图形应用优选地在具有图形协处理器或图形加速卡的平台上实现。有些系统还具有专用的
物理处理器，用于计算计算机游戏或动画中物体的物理运动。

在某些情况下，可以在
图形加速卡，用于在屏幕上渲染图形以外的其他目的。
然而，此类应用程序高度依赖于系统，因此如果可移植性很重要，则不推荐使用。本手册不包括图形处理器。

可编程逻辑器件

可编程逻辑器件是可以在硬件定义中编程的芯片
语言，如Verilog或VHDL。常见的器件有CPLD和FPGA。The
软件编程语言（例如C++）和硬件定义语言之间的区别在于，软件编程语言定义了顺序算法指令，其中硬件定义语言定义由数字构建块组成的硬件电路，如门、触发器、多路复用器、算术单元等。以及连接它们的导线。硬件定义语言本质上是并行的，因为它定义了电气连接而不是操作序列。

复杂的数字运算在可编程逻辑器件中通常比在微处理器中执行得更快，因为硬件可以用于特定目的。

可以在FPGA中实现微处理器作为所谓的软处理器。这种软处理器比专用微处理器慢得多，因此不
本身是有利的。但在某些情况下，软处理器激活在同一芯片中以硬件定义语言编码的关键应用特定指令的解决方案可能是一个非常有效的解决方案。一个更强大的解决方案是
专用微处理器内核和FPGA在同一芯片中的组合。这种混合解决方案现在被用于一些嵌入式系统中。

看看我的水晶球就会发现，类似的解决方案有一天可能会在PC处理器中实现。应用程序将能够定义可以用硬件定义语言编码的特定于应用的指令。除了代码高速缓存和数据高速缓存之外，这样的处理器将具有用于硬件定义代码的额外高速缓存。

2.2微处理器的选择

竞争品牌微处理器的基准性能非常相似
由于激烈的竞争。具有多核的处理器对于
可以分成多个并行运行的线程的应用程序。小灯
低功耗处理器实际上相当强大，可能足以满足不太密集的应用。

2.3操作系统的选择

x86系列中较新的微处理器可以在16位、32位和64位模式下运行。

16位模式用于旧操作系统DOS和Windows3.x。如果程序或数据的大小超过64K字节，se系统将使用内存分段。对于更大的程序来说，这是相当低效的。现代微处理器没有针对16位模式进行优化，一些操作系统也不向后兼容16位程序。不建议制作16位程序，小型嵌入式系统除外。

64位操作系统现在很常见。这些系统能够运行32位和64位程序。64位系统可以将性能提高5-10%
一些具有许多函数调用的CPU密集型应用程序，以及对于使用
大量的RAM内存。如果瓶颈在其他地方，那么32位和64位系统之间的性能没有区别。使用大量
内存将受益于64位系统更大的地址空间。

Windows和Linux操作系统为32位软件提供了几乎相同的性能，因为这两个操作系统使
用相同的函数调用
惯例。这里所说的关于Linux的一切也适用于BSD系统。

基于英特尔的Mac OS X操作系统基于BSD，但编译器使用
默认情况下，与位置无关的代码和惰性绑定，这使得它的效率较低。The
通过使用静态链接和不使用位置无关代码（option-fno-pic）可以提高性能。

64位系统比32位系统有几个优点：

·注册人数翻了一番。这使得可以存储中间数据
局部变量在寄存器中而不是在内存中。如果启用AVX512指令集，64位模式下的矢量寄存器数量甚至更多。

·函数参数在寄存器中传递，而不是在堆栈上传递。这使得函数调用更加高效。

·整数寄存器的大小扩展到64位。这在使用64位整数的应用程序中是一个优点。

•大内存块的分配和释放效率更高。

•所有64位CPU和操作系统都支持SSE2指令集。

·64位指令集支持数据的自相关寻址。这使得与位置无关的代码更加高效。

与32位系统相比，64位系统具有以下缺点：

·指针、引用和堆栈条目使用64位而不是32位。这使得数据缓存效率降低。

·如果不能保证映像基小于231，则访问静态或全局阵列需要很少的额外指令用于64位模式下的地址计算。

•在代码和数据的组合大小可能超过2GB的大内存模型中，地址计算更加复杂。这种大内存模型几乎从来没有
不过，用过。

·一些指令在64位模式下比在32位模式下长一个字节。

一般来说，如果有许多函数调用，如果有许多大内存块的分配，或者如果
程序可以利用64位整数计算。如果程序使用超过2千兆字节的数据，则必须使用64位系统。

在64位模式下运行时，操作系统之间的相似性消失了
因为函数调用约定不同。64位Windows只允许四个
函数参数在寄存器中传输，而64位Linux、BSD和Mac允许在寄存器中传输多达14个参数（6个整数和8个浮点）。
还有其他一些细节可以使64位Linux中的函数调用比64位Windows中的函数调用更有效（请参见49和手册5：“为不同的C++调用约定”
具有许多函数调用的应用程序在64位Linux中的运行速度可能比在64位Windows中略快。64位Windows的缺点可以通过使关键函数内联或静态或使用编译器来减轻
整个程序优化。

2.4编程语言的选择

在开始一个新的软件项目之前，决定哪种编程是很重要的
语言最适合手头的项目。低级语言有利于优化执行速度或程序大小，而高级语言有利于制作清晰和结构良好的代码，有利于快速和容易地开发用户界面和网络资源、数据库等的接口。

最终应用程序的效率取决于编程语言的方式
已实施。当代码作为二进制可执行代码编译和分发时，效率最高。C++、Pascal和Fortr的大多数实现都是基于编译器的。

其他几种编程语言是用解释实现的。程序代码按原样分发，并在运行时逐行解释。示例包括
JavaScript、PHP、ASP和UNIX shell脚本。解释代码是非常低效的，因为循环的每一次迭代都要一次又一次地解释循环的主体。

一些实现使用即时编译。程序代码作为itis分发和存储，并在执行itis时编译。一个例子是Perl。

几种现代编程语言使用中间代码（字节码）。源代码被编译成中间代码，中间代码是分发的代码。The
中间代码不能按原样执行，但必须经过
解释或编译才能运行。Java的一些实现基于解释器，该解释器通过模拟所谓的Java虚拟机来解释中间代码。最好的Java机器使用代码中最常用部分的即时编译。C#、托管C++和Microsoft.NET Framework中的其他语言都是基于中间代码的即时编译。

使用中间代码的原因是它旨在独立于平台且紧凑。使用中间代码的最大缺点是用户必须安装一个大型运行时框架来解释或编译中间代码。这个
框架通常比代码本身使用更多的资源。

中间代码的另一个缺点是它增加了额外的抽象级别，这使得详细的优化更加困难。另一方面，即时编译器可以
专门针对它运行的CPU进行优化，而在预编译代码中进行特定于CPU的优化更加复杂。

编程语言及其实现的历史揭示了一个曲折的过程
这反映了效率、平台依赖性和易用性之间的冲突考虑
发展。例如，第一台PC有一个Basic解释器。Basic的编译器很快就可用了，因为Basic的解释版本太慢了。今天，最流行的Basic版本是Visual Basic.NET，它是用
中间代码和即时编译。Pascal的一些早期实现使用了中间代码，就像今天Java使用的代码一样。但这种语言获得了
当真正的编译器可用时，它非常不受欢迎。

从这个讨论中可以清楚地看出，编程语言的选择是一个
效率、可移植性和开发时间之间的折衷。口译语言
当效率很重要时，就不可能了。当可移植性和易于使用时，基于中间代码和实时编译的语言可能是一种可行的折衷方案
发展比速度更重要。这包括C#、Visual等语言
Basic.NET和最好的Java实现。然而，这些语言具有
非常大的运行时框架的缺点是每次程序运行时都必须加载。加载框架和编译程序所需的时间通常很长
超过执行程序所需的时间。运行时框架在运行时可能使用比程序本身更多的资源。使用这种框架的程序
有时对于简单的任务（如按下按钮或移动鼠标）有令人无法接受的长响应时间。当速度较慢时，绝对应该避免使用.NET framework
关键。

毫无疑问，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、Fortran和其他几种众所周知的语言。我的
首选C++有几个原因。一些非常好的编译器和优化的函数库支持C++。C++是一种高级高级语言，具有丰富的
其他语言中很少发现的高级功能。但是C++语言也包括低级C语言作为子集，提供了对低级优化的访问。大多数C++
编译器能够生成汇编语言输出，这对于检查编译器优化代码的程度非常有用。此外，大多数C++编译器允许
类似汇编的内部函数、内联汇编或与汇编语言的简单链接
需要最高级别优化时的模块。heC++语言是可移植的，因为C++编译器适用于所有主要平台。帕斯卡·h
C++的优点，但它没有那么通用。Fortran也很有效，但是语法非常过时。

C++中的开发非常高效，这要归功于强大的开发功能
工具。一种流行的开发工具是Microsoft VisualStudio。这个工具可以制作两个
C++的不同实现，直接编译代码和中间代码
的公共语言运行时。NET框架。显然，当速度很重要时，直接编译的版本是首选。

C++的一个重要缺点与安全性有关。没有检查数组边界违规、整数溢出和无效指针。缺乏此类检查使得
代码比其他有这种检查的语言执行得更快。但是程序员有责任对这些错误进行明确的检查，以防它们不能被程序逻辑排除。下面第页提供了一些指南14.

当性能优化具有高优先级时，C++绝对是首选的编程语言。相对于其他编程语言的性能增益可能相当可观。当性能对最终用户很重要时，这种性能的提高可以很容易地证明开发时间的可能小幅增加是合理的。

可能存在基于中间代码的高级框架

出于其他原因需要，但是部分代码仍然需要仔细优化。在这种情况下，混合实现可能是一个可行的解决方案。代码中最关键的部分

可以用编译的C++实现，其余的代码，包括用户界面等。，可以在高级框架中实现。代码的优化部分可以
可能被编译为动态链接库（DLL）或共享对象，由其余代码调用。这不是一个最优的解决方案，因为高级框架仍然
消耗大量资源，两种代码之间的转换给出了一个
额外开销。但是，如果代码的时间关键部分可以完全包含在编译的DLL中或共享，则该解决方案仍然可以提供相当大的性能改进
对象。

2.5编译器的选择

近几十年来，C++编译器变得越来越复杂。与
AVX512指令集扩展对于高级矢量运算，我们现在有数千种不同的机器指令。C++蓝谷时代的许多新的高级功能
标准也增加了复杂性。在同一时期，C++的数量
市场上的编译器已经减少，一些过去流行的旧编译器现在要么过时，要么停产。我怀疑未来市场上的编译器数量可能会进一步减少。

下面提到了一些适用于x86和x86-64平台的最新编译器。这些编译器都支持32位和64位代码以及automaticvectorization。

Visual Studio

这是一个非常用户友好的编译器，有许多特性和良好的调试器。完整的
版本非常昂贵，但有限的非商业版本是免费的。VisualStudio可以用各种编程语言（C++、C#、VisualBasic）构建代码
各种平台。就代码优化而言，Visual Studio不是最好的编译器，但它在项目的开发阶段非常有用。您可以使用不同的
如果性能很重要，则为代码的发布版本编译器。可以将Visual Studio集成开发环境（IDE）与不同的编译器一起使用。可以推荐使用Clang编译器的插件

Gnu

gcc或Gnu编译器是最好的优化编译器之一。它不如Visual Studio用户友好，有时你必须是一个真正的书呆子才能理解许多命令行选项。这个编译器是免费和开源的，几乎可以在任何平台上使用。独立编译器是从命令行运行的，但是有几个集成开发环境可用，包括Eclipse、NetBeans、CodeBlocks等等。

铿锵

Clang编译器是一个非常好的优化编译器。在许多情况下，它实际上是最好的。Clangis与Gnu编译器非常相似，具有相同的特性和选项。Clang是基于x86的Macplatform上最常见的编译器，但它也支持Linux和
Windows平台。

Windows版Clang编译器有不同的版本。目前最好的版本是作为Microsoft Visual Studio版本17或更高版本的插件提供的版本。

默认情况下，Clang的Cygwin64版本使用中等内存模型。这是相当浪费的，因为它将为静态变量和常量使用64位绝对地址，而不是32位相对地址。您可以通过以下方式提高性能
指定-mcmodel=small。只有在直接链接到外部DLL中的变量时，才需要中等内存模型（这是不好的编程实践
反正）。Cygwin版本的另一个缺点是在分发可执行文件时必须包含CygwinDLL。

英特尔C++编译器

这个编译器没有自己的ide。它旨在作为Microsoft Visual的插件
为Windows编译时的Studio或为Linux编译时的Eclipse。当从命令行或makeutility调用时，它也可以用作独立的编译器。它
支持Windows和Linux以及基于英特尔的Mac OS。

英特尔编译器有两个版本：一个名为“经典”的旧版本和一个名为基于英特尔oneAPILLVM的编译器的新版本。后一个版本是Clang编译器的一个分支，其行为非常像Clang。它可以比经典版本更好地优化。

“经典”英特尔编译器支持CPU自动调度生成多个代码
不同英特尔CPU的版本。经典英特尔最重要的缺点
编译器是针对特定IntelCPU优化的代码。编译后的代码e检查它是否在英特尔CPU上运行。如果检测到另一个品牌的CPU（AMDor VIA），那么它将以降低的速度运行不同的代码分支，即使最佳代码分支与其运行的处理器兼容。

评论

开源编译器——Clang和Gnu——是代码中最好的
根据我的测试进行优化。参见第页76 fora编译器详细测试
上面提到的。

所有编译器都可以作为命令行版本使用，而无需anIDE。商业编译器有免费试用版或精简版。

混合来自不同编译器的目标文件通常是可能的，如果它们是为相同的平台编译的，并且具有相似的选项。英特尔编译器生成与Microsoft Visual Studio或Gnu编译器兼容的目标文件，具体取决于
平台。Gnu和Clang编译器的目标文件也是相互兼容的。编译器的其他组合要求链接函数在“C”外部声明。一些较旧的编译器使用与
上面提到的编译器。

在某些情况下，编译器的选择可能取决于与遗留代码的兼容性要求、IDE的特定偏好、调试工具、easy G UI
开发、数据库集成、web应用集成、混合语言编程等。用不同的编译器制作最关键的模块可能很有用
所选编译器不提供最佳优化的情况。如果主项目是用不同的编程语言编写的，则可能需要将动态链接库中优化的C++代码（.DLL或.so）。否则，首选静态链接。

2.6函数库的选择

一些应用程序将大部分执行时间花在执行库函数上。耗时的库函数通常属于以下类别之一：

- 文件输入/输出
- 图形和声音处理
- 内存和字符串操作
- 数学函数离子
- 加密、解密、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。

库函数通常是由许多用户在许多
不同的应用。因此，在优化库函数上投入比优化特定于应用程序的代码更多的精力是值得的。最好的函数库是高度优化的，使用自动CPU调度（参见135）用于最新的指令集扩展。

lfa概况（见第16）显示了一个特定的应用程序在
库函数，或者如果这是显而易见的，那么就有可能提高性能
非常简单地使用不同的函数库。如果应用程序使用其大部分
除了找到最有效的库和节省库函数调用之外，可能没有必要优化任何东西。建议尝试不同的库，看看哪一个效果最好。

下面讨论一些常见的函数库。许多特殊用途的图书馆也是可用的。

微软

附带微软编译器。有些功能优化得很好，有些则不行。支持32位和64位Windows。

Gnu

附带Gnu编译器。64位版本比32位版本好。Gnucompiler经常插入内置代码，而不是最常见的内存和字符串
说明。内置代码并不总是最佳的。请使用option-fno-builtin来获取库版本。

Mac

MacOS X（Darwin）的Gnu编译器中包含的库是Xnu的一部分
项目。一些最重要的功能包含在操作系统内核中，称为compage。这些功能针对英特尔酷睿及更高版本进行了高度优化
英特尔处理器。AMD处理器和早期的英特尔处理器根本不受支持。只能在Macplatform上运行。

英特尔

英特尔编译器包括标准函数库。还有几个专用库可用，如“英特尔MathKernel库”和“Integrated Performance
这些函数库针对大型数据集进行了高度优化。智能库针对英特尔处理器上的最佳性能进行了优化，但它们通常在AMD处理器上也能提供令人满意的出厂性能，除非它们与英特尔编译器的“经典”版本。参见第页143论坛讨论英特尔的使用
非英特尔处理器上的编译器和库。

AMD

AMD核心数学库包含优化的数学函数。它也适用于英特尔处理器。性能不如英特尔库。

阿斯姆利布

我自己的函数库是为了演示而制作的。可从
www.agner.org/optimize/asmlib.zip。目前包括优化版本的内存和
字符串函数和其他一些很难在别处找到的函数。比
在最新的处理器上运行时，许多其他库。支持所有x86和x86-64平台。

函数库比较

测试	处理器	微软	代码齿轮	英特尔	Mac	Gnu 32位	Gnu 32位 -fno-内置	Gnu64位 -fno-内置	阿斯姆利布
memcpy16kB 对齐 操作数	英特尔 Core2	0.12	0.18	0.12	0.11	0.18	0.18	0.18	0.11
memcpy16kB unaligned op。	英特尔 Core2	0.63	0.75	0.18	0.11	1.21	0.57	0.44	0.12
memcpy16kB 对齐 操作数	AMD OpteronK8	0.24	0.25	0.24	不适用。	1.00	0.25	0.28	0.22
memcpy16kB unaligned op。	AMD OpteronK8	0.38	0.44	0.40	不适用。	1.00	0.35	0.29	0.28
strlen128 字节	英特尔 Core2	0.77	0.89	0.40	0.30	4.5	0.82	0.59	0.27
strlen128 字节	AMD OpteronK8	1.09	1.25	1.61	不适用。	2.23	0.95	0.6	1.19

表2.1。比较不同函数库的性能。
表中的数字是每字节da ta的核心时钟周期（数字低表示好
性能）。对齐操作数意味着源和目标都具有可被16整除的地址。
测试的库版本（尚未更新！）：
Microsoft Visual studio 2008，
v.9.0CodeGear Borlandbcc，v.5.5
Mac：Darwin8 g++v 4.0.1。
Gnu：Glibc v.2.7，2.8。
Asmlib：v.2.00。
英特尔C++编译器，v.10.1.020.functions_intel_fast_m emcpy和
__intel_new_strlen在librarylibircmt.lib中。函数名未记录。

2.7用户界面框架的选择

典型软件项目中的大部分代码都用于用户界面。应用程序
不是计算密集型的，很可能会在用户界面上花费更多的CPU时间，而不是在程序的基本任务上。

应用程序程序员很少从头开始编写自己的图形用户界面。这不仅浪费程序员的时间，也给最终用户带来不便。菜单、按钮、对话框等。出于可用性的原因，应该尽可能标准化。程序员可以使用
编译程序和开发工具附带的操作系统或库。

适用于Windows和C++的流行用户界面库是Microsoft基础类
（MFC）。有几种图形界面框架可用于Linuxsystems。用户界面库可以作为runt ime DLL或staticlibrary链接。运行时DLL
比staticlibrary占用更多的内存资源，除非多个应用程序同时使用sameDLL。

用户接口库可能比应用程序本身大，需要更多的时间来加载。一个轻量级的替代方案是Windows模板库（WTL）。WTL应用程序是

通常比MFC应用程序更快、更紧凑。由于糟糕的文档和缺乏先进的开发工具，WTL应用程序的开发时间预计会更长。

有几个跨平台的用户界面库，比如Qt和wxWidgets。这些对于可以在多个平台和操作系统上运行的应用程序非常有用。

通过删除图形用户界面，可以获得最简单的用户界面
并制作控制台模式程序。控制台模式程序的输入通常在命令行或输入文件中指定。输出进入控制台或ou tputfile。控制台模式程序快速、紧凑且易于开发。它很容易移植到
不同的平台，因为它不依赖于特定于系统的图形接口调用。可用性可能很差，因为它缺乏图形用户的自我解释菜单
接口。控制台模式程序对于从其他应用程序（如makeutility）调用非常有用。

结论是用户界面框架的选择必须在开发时间、可用性、程序紧凑性和执行时间之间进行权衡。没有通用的解决方案最适合所有应用。

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
使开发人员选择其他编程语言的缺点。本节讨论在选择C++时如何克服这些缺点
优化。

便携性

C++是完全可移植的，因为语法是完全标准化的，并且在所有
主要平台。然而，C++也是一种允许直接访问硬件的语言
接口和系统调用。这些当然是特定于系统的。为了方便
在平台之间移植时，建议将用户界面和其他特定于系统的代码部分放在一个单独的模块中，并将特定于任务的代码部分（假设是独立于系统的）放在另一个模块中。

整数的大小和其他与硬件相关的细节取决于硬件平台和操作系统。参见第页29了解详情
。

开发时间

一些开发人员认为特定的编程语言和开发工具
比其他人使用更快。虽然有些差异只是习惯问题，但这是真的
一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。C++项目的开发时间和可维护性可以
通过一致的模块化和可重用类进行改进。

安全

C++语言最严重的问题与安全性有关。标准DC++实现不检查数组边界冲突和无效指针。这是一个
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，以防止安全性很重要的程序中出现此类错误。

通过使用引用而不是指针，通过将指针初始化为零，通过在指针指向的对象无效时将指针设置为零，以及通过避免指针算术和指针类型转换，可以避免无效指针的问题。
容器类模板，如第页所述97.避免使用scanf函数。

违反数组界限可能是C++程序出错的最常见原因。
写入数组末尾会导致其他变量被覆盖，甚至
更糟糕的是，它可以覆盖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。输入数据缓冲区溢出的missingcheck是黑客经常利用的常见错误。

这是防止用容器类替换数组出现错误的好方法。在页面上讨论了不同容器类的效率97.

文本字符串尤其有问题，因为字符串的长度。以字符顺序存储字符串的旧c样式，快速有效，但除非每个字符串的长度经过检查，否则不安全。标准
解决这个问题的方法是使用字符串类，如字符串或CString。这是安全和灵活的，但在大型应用程序中相当低效。字符串类分配了一个新的
每次创建或修改一个字符串时的记忆块。这是分散的，涉及高昂的货物管理和行李管理费用。一个不影响安全问题的更有效的解决方案是将所有字符串存储在一个系统中
memorypool.请参见应用程序附录中的示例
www.阿格纳公司。org/ optimize/ cppexamples.zip为如何存储字符串和内存池。

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，签名整数的行为是“未定义的”。这允许程序忽略
溢出或假设这没有发生。在Gnu编译器的情况下，假设没有发生有一个不幸的结果，它允许编译器优化溢出检查。对于这个问题，有许多可能的解决方法： (1)在溢出发生之前检查是否有溢出，(2)使用无符号的ed整数-
它们保证环绕，(3)使用ftrapv选项溢出，但是
这是非常有效的，(4)得到一个编译器的优化警告
选项-严格溢出=2，或(5)通过选项-fwrapv或-严格严格溢出明确定义溢出行为。

在速度很重要的代码的关键部分，您可能会偏离上述安全建议。如果不安全的代码仅限于测试良好的函数，则可以允许这样做，
与程序的其余部分具有定义良好的接口的类、模板或模块。

3.找到最大的时间的消费者

3.1时钟周期是多少？

在我的手册中，我有趣的C PU时钟循环，而不是秒或微秒。这是因为计算机使用的速度会非常不同。如果我写了一些东西
takes10μs今天，那么它可能只需要5μs在下一代电脑和我的手册很快就会过时。但是，我写了一些东西需要10个时钟周期，然后它仍然需要10个时钟周期，即使CPU时钟频率加倍。

时钟周期的长度是时钟频率的倒数。例如，如果时钟频率为4 GHz，那么一个时钟周期的长度是

$$\frac{1}{4\ \mathrm{GHz}} = 0.25\mathrm{ns}.$$

假设程序中的一个循环重复1000次，并且循环内有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，那么我们可以粗略地估计这个循环将需要1000*100*5*
在4 GHzCPU上为0.25 ns=125 μ s。我们应该尝试优化这个循环吗？当然不是！125

μ s不到刷新屏幕所需时间的1%。用户不可能
看到延迟。但是如果这个循环在另一个重复1000次的循环中，那么我们
估计执行时间为125毫秒。这种延迟足够长，可以被注意到，但还没有长到令人讨厌的程度。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上超过125毫秒。如果响应
时间只是用户实际上必须等待结果，然后我们会考虑是否有可以改进的地方。

3.2使用Profiler查找热点

在你开始优化任何东西之前，你必须确定程序的关键部分。
在某些程序中，99%以上的时间都花在innermostloop上
数学计算。在其他项目中，99%的时间花在阅读和
写入数据文件，而不到1%的时间实际上对这些数据做了一些事情。它是
优化重要的代码部分而不是
代码只使用总时间的一小部分。优化代码中不太关键的部分不仅浪费时间，还会使代码不那么清晰，更难调试和维护。

大多数编译器包都包含一个profiler，它可以告诉seach函数有多少时间
打电话和需要多少时间。还有第三方分析器，如AQtime、IntelVTune和AMD CodeAnalyst。

有几种不同的分析方法：

·Instrumentation：编译程序在每个函数调用中插入额外的代码，以计算函数被调用的次数和花费的时间。

•调试。探查器在每个函数或每个代码行中插入临时调试断点。

·基于时间的采样：Profiler告诉操作系统生成中断，例如。每一毫秒。Profiler计算程序在程序的每个部分中出现中断的次数。这不需要修改被测程序，但可靠性较低。

•基于事件的采样：Profiler告诉CPU在特定时刻生成中断
事件，例如每次发生一千次缓存未命中时。这使得可以看到程序的哪个部分有最多的缓存，bran ch
错误预测、浮点异常等。基于事件的采样需要CPU-
特定探查器。英特尔CPU使用英特尔VTune，AMD CPU使用AMD CodeAnalyst。

不幸的是，分析器通常是不可靠的。他们有时会给出误导性的结果，或者因为技术问题而完全失败。

分析器的一些常见问题是：

·粗略时间测量。如果以毫秒分辨率测量时间，并且关键函数需要微秒来执行，则测量可以变成
不精确或干脆为零。

•执行时间太短或太长。如果被测程序在短时间内完成，则采样产生少量数据用于分析。如果程序执行时间过长，那么分析器可能会采样比它所能处理的更多的数据。

·等待用户输入。大多数程序花费大部分时间等待用户输入或网络资源。这个时间包含在配置文件中。可能有必要修改

程序使用一组测试数据代替用户输入，以使分析可行。

·来自其他过程的干扰。分析器不仅测量在被测程序中花费的时间，还测量在同一台计算机上运行的所有其他进程使用的时间，包括分析器本身。

·函数地址在优化程序中是模糊的。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。但是一个高度优化的程序经常会以这样一种方式重组，使得函数名和代码地址之间没有明确的对应关系。The
内联函数的名称可能对探查器根本不可见。结果将是关于哪些功能花费最多时间的误导性报告。

•使用代码的调试版本。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行。代码的调试版本没有优化。

•CPU内核之间的跳跃。在多核CPU上，进程或线程不一定停留在同一个处理器内核中，但事件计数器会。对于在多个CPU内核之间跳跃的线程，这会导致meaninglessevent计数。您可能需要通过设置线程亲和掩码将线程锁定到特定的CPU核心。

·再现性差。程序执行的延迟可能是由不可再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，并使程序的某些部分看起来比正常情况下花费更长的时间。

使用Profiler有多种替代方案。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break。如果有一个热点占用了90%的CPU时间，那么有90%的机会在这个热点发生中断。重复中断几次可能足以识别热点。使用调试器中的调用堆栈来识别热点周围的环境。

有时，最好的方法是识别绩效瓶颈来衡量
仪器直接进入代码，而不是使用区域模型分析器。但事实并非如此
解决所有与配置分析相关的问题，但通常会提供更可靠的结果。如果你对自动分析器的工作方式不满意，那么你可以把所需的测量值
仪器进入程序本身。你可以添加计数器变量来计算有多少个的计数器变量
执行程序每个一部分的时间。此外，您还可以阅读程序中每个最重要或最关键的部分之前和之后的时间，以衡量每个部分所花费的时间。请参见第页167年，以便进一步讨论该方法。

您的测量代码应该有#if指令，这样可以在代码的最终版本中禁用。在代码本身中插入你自己的分析工具是一个问题
在程序开发过程中跟踪性能的非常有用的方法。

如果时间间隔s很短，那么时间测量可能需要一个很高的分辨率。在
您可以使用获取计数器查询权限来获得毫秒分辨率。使用时间戳可以获得更高的分辨率
CPU中的计数器，它计数CPU时钟频率（_rdtsc（）或rdtsc（））。

如果路径在不同的cpucore之间跳转，时间戳计数器将无效。您可能需要在时间测量期间修复线程到一个特定的CPU核心，以避免这一点。（在窗口中，设置线程设备掩码，inLinux，sched_setaffinity）。

该程序应该用一组真实的测试数据来进行测试。测试数据应该包含非典型的随机程度，以获得缓存丢失和分支预测的真实数量。

当找到程序中最耗时的部分时，重要的是只将优化工作集中在耗时的部分上。关键代码片段可以通过第页中描述的方法进一步测试和研究167.

Profiler对于查找与CPU密集型代码相关的问题非常有用。但很多
程序加载文件或访问数据库、网络和其他资源比进行算术运算花费更多的时间。最常见的时间消耗者将在下面的章节中讨论。

3.3程序安装

安装程序包所需的时间传统上不被认为是软件
优化问题。但它肯定会偷走用户的时间。时间
如果目标是
软件优化是为了节省用户的时间。随着现代的高度复杂性
软件，安装过程需要一个多小时并不罕见。用户必须多次重新安装软件包以便找到并解决兼容性问题。

软件开发人员在决定是否将软件包建立在需要
许多要安装的文件。

安装过程应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，以便安装的其余部分
进程可以在无人值守的情况下进行。卸载也应该以标准化的方式进行。

3.4自动更新

许多软件程序定期通过互联网自动下载更新
时间间隔。有些程序在计算机每次启动时都会搜索更新，即使这些程序从未被使用过。安装了许多此类程序的计算机可能需要几分钟才能启动，这完全是浪费用户的时间。其他程序花费时间
每次程序启动时搜索更新。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原因需要更新。更新进程应该在低优先级线程中运行，并且只有在程序被实际使用时才运行。没有
程序不使用时应该让后台进程运行。下载的程序更新的安装应该推迟到程序关闭并重新启动。

操作系统的更新可能特别耗时。有时需要
为操作系统安装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会在不方便的时候不可预测地出现。如果用户出于安全原因不得不关闭或注销计算机，这可能是一个非常大的问题
在离开工作场所之前，系统禁止用户在更新过程中关闭计算机。

3.5程序加载

通常，加载程序比执行程序花费更多的时间。加载时间可以是
对于基于bigruntime框架、中间代码、解释器、即时编译器等的程序来说，这是令人恼火的高，这通常是用
Java、C#、Visual Basic等。

但是即使对于用compiledC++实现的程序，程序加载也是非常耗时的。如果程序使用大量运行时DLL（动态链接库或共享对象）、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bi程序的所有模块
向上。一些模块可能仅在需要时才加载，或者如果RAM大小不足，则可以将其交换到硬盘上。

用户希望对按键或鼠标移动等简单动作立即做出响应。如果这样的响应延迟了几秒钟，这对于用户来说是不可接受的，因为
需要从磁盘加载模块或资源文件。内存密集型应用程序
强制操作系统将内存交换到磁盘。内存交换是对鼠标移动或按键等简单事情的响应时间过长的常见原因。

避免过多的DLL、配置文件、资源文件、帮助文件等分散在硬盘上。几个文件，最好
在与可执行文件相同的目录中，是可以接受的。

3.6动态链接和位置无关代码

函数库可以实现为静态链接库（*。lib，*。a）或动态链接库，也称为共享对象（*。dll，*。所以）。有几个因素可以
使动态链接库比静态链接库慢。第页详细解释了这些因素158以下。

在类UNIX系统的共享对象中使用与位置无关的代码。Mac系统
默认情况下，通常在任何地方使用与位置无关的代码。与位置无关的代码效率很低，尤其是在32位模式下，原因请参见158以下。

3.7文件访问

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，尤其是如果用户有
一个病毒扫描程序来扫描访问的所有文件。

对文件的顺序转发访问比随机访问更快。阅读或写作
块比一次读取或写入sm allbit更快。不要一次读取超过几千字节的无写数据。

您可以将整个文件镜像到内存缓冲区中，并在一次操作中读取或写入它，而不是以非顺序方式读取或写入小位。

访问最近访问过的文件通常比第一次访问要快得多。这是因为文件已被复制到磁盘缓存。

可能不会缓存远程或可移动介质（如u盘）上的文件。这可能会产生相当戏剧性的后果。我曾经做过一个windows程序，它通过调用WritePrivateProfileString来创建一个文件，它为写入的每一行打开和关闭文件。由于磁盘缓存，这在硬盘上运行得足够快，但需要几分钟
将文件写入软盘。

包含数值数据的大文件如果存储在
二进制格式，而不是数据以ASCII格式存储。二进制数据存储的一个缺点是它不是人类可读的，并且不容易移植到具有大端存储的系统。

优化文件访问比使用一个具有任何文件输入/输出操作的程序来优化CPU更重要。如果处理器在等待磁盘操作完成时可以做其他工作，则在sepa速率线程中访问顶部文件可能是有利的。

3.8系统数据库

在Windows中访问系统数据库可能需要几秒钟的时间。将应用程序专用计算机信息存储在一个单独的文件中比在大注册中更有效率

Windows系统中的数据库。请注意，如果您使用专用配置文件和写入私人配置文件来读写配置文件（*.inifiles），系统可以将信息存储在数据库中。

3.9其他数据库

许多软件应用程序都使用数据库来存储用户数据。一个数据库可以消耗大量的CPU时间、RAM和磁盘空间。在简单的情况下，可以用一个普通的数据文件来替换一个数据库。数据库查询通常可以通过使用索引进行优化，

使用集合而不是循环等。优化数据库查询超出了本手册的范围，但是您应该知道，通过优化通常可以获得很多好处

数据库访问。

3.10图形

图形用户界面可以使用大量计算资源。通常，特定的

使用图形框架。操作系统可以提供这样的框架初始化API。在某些情况下，在

操作系统API和应用软件。这种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是作为函数调用实现的

graphicslibrary或API函数，然后调用设备驱动程序。对图形函数的调用非常耗时，因为它可能会遍历多个层，并且需要切换到

保护模式，然后再回来。显然，单次调用绘制整个多边形或位图的Graphics函数比通过多个函数调用分别绘制每个像素或线更有效。

计算机游戏和动画中图形对象的计算当然也很耗时，尤其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在性能上有很大差异。我没有具体推荐哪一个是最好的。

3.11其他系统资源

对打印机或其他设备的写入最好是大块完成，而不是一次一小块，因为对驱动程序的每次调用都涉及切换到保护模式并再次切换回来的开销。

访问系统设备和使用操作系统的高级设施可以

耗时，因为它可能涉及加载几个驱动程序、配置文件和系统模块。

3.12网络接入

一些应用程序使用Internet或intranet进行自动更新、远程帮助文件，

数据库访问等。这里的问题是访问时间无法控制。The

在简单的测试设置中，网络访问可能很快，但在网络过载或用户远离服务器的使用情况下，网络访问可能会变慢或完全不存在。

在决定是在本地还是远程存储帮助文件和其他资源时，应该考虑这些问题。如果frequentupdates是必要的，那么它可能是
最适合本地镜像远程数据。

访问远程数据库通常需要使用密码登录。对于许多勤奋的软件用户来说，登录过程是一个令人讨厌的耗时过程。在一些
在这种情况下，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的时间。

3.13存储器访问

与对数据进行计算所需的时间相比，从RAM存储器访问数据可能需要相当长的时间。这就是为什么所有现代计算机都有内存缓存的原因。通常，存在8-64K字节的1级数据高速缓存和256K字节到2Mbytes的2级高速缓存。通常，还有几个字节的三级缓存。

如果程序中所有数据的总大小大于二级缓存和数据
分散在内存中或以无序方式访问，那么内存访问很可能是程序中最耗时的。读取或写入
如果缓存，内存中的变量只需要2-4个时钟周期，如果没有缓存，则需要几百个时钟周期
。参见第页25关于数据存储和页面91关于内存缓存。

3.14上下文切换

上下文切换是在多任务分配环境中的不同任务之间、在多线程程序中的不同线程之间、或在大程序的不同部分之间的切换。
频繁的上下文切换可能会降低性能，因为数据缓存的内容，代码高速缓存、分支目标缓冲区、分支模式历史记录等。可能不得不结婚。

上下文切换比分配给每个任务或线程的时间片更小。时间片的长度由操作系统决定，而不是由应用程序决定。

在有多个CPU或有多个核的计算机中，上下文切换的数量更小。

3.15依赖链

现代的微处理器可以进行无序的执行。这意味着这一块
软件指定了A和t henB的计算，以及A isslow的计算，然后单微处理器可以在A的计算完成之前开始B的计算。
显然，这只有在计算B不需要A值的情况下才可能。

为了利用无序执行的机会，您必须避免长时间的依赖关系
链依赖链是一系列的计算，其中每个计算都取决于前一个计算的结果。这会防止CPU进行多次计算
同时发生的或无序的。请参见第页113为如何打破稳定链的例子。

3.16执行单位吞吐量

这在延迟量和执行的吞吐量之间有一个重要的区别
单元例如，它可能需要三个时钟周期来完成一个浮点加法。但每个时钟周期都可以启动一两个新的浮动点添加。
这意味着每个加法取决于前面加法的结果，然后你

每三个时钟周期只有一个加法。但是加法是独立的，然后你可以有一个或两个附加时钟周期。

在计算过程中可能获得的高性能
当上述部分中提到的时间消费者都不占主导地位，也没有任何依赖链时，就实现了程序。在这种情况下，性能是
受执行单元的吞吐量的限制，而不是受时间或内存访问的限制。

现代微处理技术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，以及一个或两个浮点乘法单元。这意味着可以在同一时间进行积分加法、浮点加法和浮动点乘法。

因此，进行浮点计算的代码最好具有添加和乘法的平衡混合物。减去相同单位的附加物。师塔克朗格时间。可以在浮点运算之间进行整数运算
而不会因为整数操作使用不同的执行方式而降低性能
单位例如，执行浮点点计算的循环通常会使用整数
增加循环计数器，比较循环计数器及其限制等。在大多数情况下，您可以假设这些整数操作不增加到总数
计算时间

4性能和可用性

一个性能更好的软件产品是一种品味海瑟堡的软件产品。对许多计算机用户来说，时间是宝贵的资源，许多时间浪费在慢的软件上，
访问困难的，不兼容的，或容易出错的。所有这些问题都是可用性上的问题，我认为软件的性能应该从更广泛的角度来看。

这不是一个关于可用性的手册，但我认为有必要在这里来吸引人们的注意
软件程序员对有效使用软件的一些最常见的障碍。欲了解更多信息，请参阅我的免费电子书 Wikibooks书呆子的可用性。

下面的列表指出了一些典型的沮丧和浪费时间的来源
软件用户以及软件开发人员应该注意的重要可用性问题。

•大型运行时框架。.NET框架和Java虚拟机是
通常比程序占用更多资源的框架
跑步。这种框架是资源问题和兼容性问题的常见来源，
在框架下运行的程序的安装过程中，在频繁的
在程序启动期间和程序运行期间进行更新。主要原因是为什么为了跨平台可移植性而在allis使用这样的运行时框架。不幸的是，跨平台兼容性并不总是像预期的那样好。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。

·存储器交换。软件开发人员典型的盟友拥有更强大的计算机
比最终用户拥有的更多RAM。因此，开发人员可能看不到过多的内存交换和其他导致资源匮乏的资源问题
应用程序对最终用户来说性能不佳。

•安装问题。安装和卸载程序的过程应该标准化，由操作系统而不是个人来完成

安装工具。

•自动更新。如果网络不稳定，或者如果新版本导致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。
更新机制经常用烦人的弹出消息来打扰用户
请安装这个重要的新更新，甚至告诉用户在他或她忙于重要工作时重新启动计算机。更新机制
不应中断用户，而应仅显示一个离散图标，表示更新可用，或在程序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他们软件的新版本。这让用户很恼火。

•兼容性问题。所有软件都应该在不同的平台上测试，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。
软件应该使用标准的API调用，而不是自我风格的黑客和直接的
硬件访问。应使用可用的协议和标准化文件格式。
Web系统应该在不同的浏览器、不同的平台、不同的屏幕中进行测试
决议等。应遵守无障碍指南。

•版权保护。一些警察保护方案是基于违反或
规避操作系统标准。这样的计划是经常出现的来源
兼容性问题和系统故障。许多副本保护方案都是这样的
基于硬件识别。这种方案在硬件更新时会导致问题。大多数版权保护方案都恼用户，在不有效防止非法复制的情况下防止合法备份复制。的好处
版权保护方案应与使用问题和必要支持方面的成本进行权衡。

• Hardwareupdating.更换硬盘或其他硬件通常需要所有软件重新安装和设置丢失。重新安装工作需要整个工作日或更长时间并不少见。许多软件应用程序需要更好的备份功能，而当前的操作系统需要更好地支持硬盘复制。

沙痂安全。软件的网络访问病毒攻击的漏洞等
虐待对许多用户来说是极其昂贵的。防火墙、病毒扫描仪和其他保护手段是导致兼容性问题和系统的最常见原因之一
碰撞此外，病毒扫描仪在计算机上花费的时间都不少见。作为操作系统的一部分的安全软件通常比第三方安全软件更可靠。

沙痂背景服务。许多在后台运行的服务对用户来说是不必要的，而且是浪费资源的。考虑仅在激活时运行服务
theuser.

Subab功能膨胀。出于市场上的原因，软件向每个新版本添加新功能是很常见的。这可能会导致软件变慢或需要更多
资源，即使用户从不使用新的特性。

认真对待用户的反馈。用户投诉应该被视为关于信息漏洞、兼容性问题、可用性问题和期望的新信息的有价值的来源
特征用户的反馈应以系统的方式处理，以确保这些信息得到适当的利用。用户应该得到关于问题的调查和计划的解决方案的答复。补丁应该很容易从一个网站上获得。

5.选择最优算法

当您想要优化一个cpu密集型软件时，要做的第一件事就是找到最好的算法。算法的选择是非常重要的任务，比如分类，
搜索和数学计算。在这种情况下，你可以通过选择最佳算法比优化即将出现的第一个算法获得更多的信息。在某些情况下，您可能需要测试几种不同的算法，以找到在一个典型的测试数据集上工作得最好的算法。

话虽如此，我必须警告不要过度杀人。不要使用先进的和复杂的吗
如果一个简单的算法能足够快地完成工作。例如，一些程序显示表甚至是最小的数据列表。哈希表可以提高搜索时间
对于非常巨大的数据库，但没有理由使用它为这样的列表
一个二进制搜索，甚至是线性搜索，都足够快了。一个哈希表股份有限公司的回复
程序的大小以及数据文件的大小。这实际上会降低速度，如果
瓶颈是文件访问或高速缓存访问，而不是CPUtime。复杂算法的另一个缺点是使程序开发更昂贵
errorprone.

对针对不同目的的不同算法的讨论超出了此范围
用手的你必须查阅关于算法和数据结构的一般文献
标准的任务，如排序和搜索，或针对更复杂的数学任务的具体文献。

在开始编码之前，您可以考虑是否在您之前完成了其他工作。
用于许多标准任务的优化函数库可以从多个数据库中获得
来源例如，Boost集合包含了许多许多常见目的的经过良好测试的库(www.使增长org).“英特尔数学内核库”包含许多函数
常见的数学计算，包括线性代数和统计学，以及“英特尔”
性能原始程序“库包含许多音频和视频处理的功能，
信号处理、数据压缩和密码学(www.intel.com).如果您使用的是一个Intel函数库，那么请保证它在非智能处理器上工作良好，如第页所述143.

在开始编程之前选择最优算法说起来容易做起来难。许多程序员都发现，只有在他们将整个软件项目放在一起并进行测试之后，才有更聪明的做事方法。你所获得的洞察力
测试和分析程序性能以及研究瓶颈可以导致
更好地理解问题的整个结构。这种新的洞察力可以导致程序的彻底重新设计，例如当您发现有更聪明的方法来组织数据时。

对一个已经有效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，还可能导致结构更好、更易于维护的程序。时间你花在重新设计一个程序上的时间可能比你花在解决最初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理，存在着长期的争论。我不会推荐任何特定型号。相反，我将就开发过程如何影响
最终产品的性能。

对数据结构、数据流和算法进行彻底的分析是有好处的
规划阶段以预测哪些资源是最关键的。然而，在早期规划阶段可能有如此多的未知因素，以至于不容易获得问题的详细概述。在复杂的情况下，完全理解只有在很晚的时候才会到来
开发过程中的阶段。在这种情况下，您可以查看软件开发
作为一个学习过程，主要反馈来自测试和
编程问题。在这里，您应该为red esign的几次迭代做好准备，以防您发现不同的程序结构更有效或更易于管理。

一些软件开发模型具有严格的形式主义，需要在软件的逻辑架构中进行几层抽象。你应该知道有
这种形式主义的内在性能成本。将软件分割成过多的独立抽象层是性能降低的常见原因。

因为大多数开发方法本质上都是增量或迭代的，所以有一个策略来保存每个中间版本的备份副本是很重要的。单人论坛
项目，它足以使每个版本的zip文件。对于团队项目，建议使用版本控制工具。

7不同C++构造的效率

大多数程序员很少或根本不知道如何将一段程序代码翻译成
机器代码以及微处理器如何处理这些代码。例如，m任意
程序员不知道双精度计算和单精度计算一样快
精度。谁会知道模板类比多态类更有效呢？

本章旨在解释不同C++语言的相对效率
为了帮助程序员选择最有效的选择。这个
理论背景在本系列手册的其他卷中也有进一步的解释。

7.1不同类型的可变存储器

变量和对象存储在主题的不同部分中，这取决于它们在C-++程序中的声明方式。这影响了数据缓存的效率(参见页面91).如果数据在主题中随机分散，则数据缓存较差。因此，了解变量是如何存储的是很重要的。存储原则同样适用于简单的变量、数组和对象。

堆栈上的存储

在函数中声明的变量和对象存储在堆栈中，但下面章节中描述的情况除外。

堆叠是记忆的一部分，以最后的方式组织起来。它用于
存储函数返回地址（即函数从其中被调用的位置）、函数
参数、局部变量以及必须在
函数返回。每次调用函数时，它都会在堆栈上为所有这些目的分配所需的空间。当函数返回时，该内存空间被释放。下次调用函数时，它可以为新函数的参数使用相同的空间
功能。

堆栈是存储数据最有效的内存空间，因为相同范围的内存地址会被一次又一次地重复使用。可以肯定的是，这部分
如果没有大数组，内存将镜像到第1级数据缓存中。

我们可以从中吸取的教训是，所有变量和对象最好都应该在使用它们的函数中声明。

通过在{}括号内声明变量，可以使其范围更小。然而，大多数编译器直到函数
返回，即使Hit在退出声明变量的{}括号时可以释放内存。如果变量存储在寄存器中（见下文），那么在函数返回之前，它可能是空闲的。

全局或静态存储

在任何函数之外声明的变量称为全局变量。它们可以从任何函数访问。全局变量存储在内存的静态部分。静态内存也用于用statickeyword声明的变量，用于浮动
点常量、字符串常量、数组初始化列表、开关语句跳转表和虚函数表。

静态数据区域通常分为三个部分：一个用于从不
由程序修改，一个用于可由程序修改的初始化变量，一个用于可由程序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。缺点是整个内存空间被占用程序执行，即使变量只在一个小的程序中使用。这个
使数据隐藏的效率降低，因为内存空间不能被重用用于其他目的。

如果可以，不要使变量全局。可能需要全局变量
不同线程之间的交流，但这是关于它们不可避免的唯一情况。如果变量被几个不同的函数访问，并且您希望避免将变量作为函数传输的开销，那么使变量成为全局可能很有用
参数但这也许是一个更好的解决方案，使函数s访问保存
同类的变量，并将共享变量存储在类中。你喜欢哪种解决方案是节目风格。

最好声明一个连接表为静态和常数。样例

```
// Example7.1
浮点函数（int x）{
    静态常量浮点数列表[]={1.1、0.3、-2.0、4.4、2.5};
    返回列表
```

这样做的优点是不需要每次调用函数时都初始化列表。静态de claration帮助编译器决定表可以从一个调用到下一个调用重用。constdeclaration帮助编译器看到表从不
变化。函数内部初始化变量的静态声明意味着
变量必须在第一次调用函数时初始化，但在随后的调用中不会初始化。这是无效的，因为函数需要额外的开销来检查它是第一次调用还是以前被调用过。const声明帮助编译器确定不需要检查第一次调用。一些编译器能够优化查找
表，但是最好同时放置静态和常量查找表，以便优化所有编译器的性能。

字符串常量和浮点常量通常存储在静态内存中。示例：

```
//示例7.2
a=b*3.5;
c=d+3.5;
```

这里，常量3.5将存储在静态内存中。大多数编译器会认识到这两个常量是相同的，因此只需要存储一个常量。整个程序中所有相同的常量将被连接在一起，以最小化用于常量的缓存空间。

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

有限数量的变量可以存储在寄存器中，而不是主存储器中。寄存器是CPU内部用于临时存储的一小块内存。变量是
存储在寄存器中的访问速度非常快。所有优化编译器都会自动选择函数中最常用的变量进行寄存器存储。同一寄存器可以
用于多个变量，只要它们的用途（活动范围）不重叠。

局部变量特别适合寄存器存储。这是首选局部变量的另一个原因。

寄存器的数量有限。在32位x86操作系统中大约有6个通用整数寄存器，在64位x86操作系统中有14个整数寄存器
系统。

浮点变量使用不同类型的寄存器。有八个浮动po int
32位操作系统中可用的寄存器，64位操作系统中可用的寄存器，以及
32当AVX512指令集在64位模式下启用时。除非启用了SSE指令集（或更高），否则一些编译器很难在32位模式下生成浮点寄存器变量。

挥发性

volatilekeyword指定变量可以被另一个线程更改。这防止编译器进行依赖于以下假设的优化
变量总是具有先前在代码中分配的值。示例：

```
//示例7.3。解释挥发性
易失性秒数；//每秒由另一个线程递增

void
    DelayFiveSeconds(){seco
    nds=0;
    while（秒<5）{
        //秒数到5时什么也不做}
}
```

在本例中，DelayFiveSeconds函数将一直等待到秒数
由另一个线程递增到5。如果seconds未声明为易失性，则
优化编译器将假设seconds在while循环中保持为零，因为循环中没有任何内容可以更改该值。循环将是while(0<5){}，这将是一个无限循环。

关键字volatile的作用是确保变量存储在内存中而不是寄存器中，并阻止对变量的所有优化。这可以用于测试情况，以避免某些表达式被优化。

注意，易失性并不意味着原子。它不会阻止两个线程同时尝试写入变量。如果在其他线程增量的同时尝试将seconds设置为零，则上例中的代码可能会失败

秒。更安全的实现将只读取seconds的值，并等待该值改变五次。

线程本地存储

大多数编译器可以通过使用关键字thread_local、threador declspec（thread）来实现静态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。线程本地存储效率低下，因为它被访问
如果可能的话，应该避免静态线程本地存储，代之以线程自己堆栈上的存储。
25).在线程函数或任何
由线程函数调用的函数，将存储在线程的堆栈上。这些变量
和对象将为每个线程提供一个单独的实例。设计用于静态线程局部
通过在第读函数中声明变量或对象，可以避免在大多数案例中使用存储。

远

具有分段内存的系统，如旧的DOS和16位Windows，允许
变量通过使用关键字远数据段(DOS数组可以
这也是巨大的)。远存储、远指针和远过程都是低效的。如果一个程序为一个段的大量数据，那么建议使用不同的操作系统，允许更大的段（32位或64位系统）。

动态内存分配

动态内存分配是用操作符新建和删除器来完成的
函数malloc和免费。这些操作符和函数消耗了大量的时间。用称为动态分配的内存。当不同大小的对象被分配和非随机顺序释放时，堆很容易变成碎片化。堆管理器可以花大量时间清理没有的空间
延长使用时间和寻找空的空间。这就是所谓的花环。按顺序分配的对象不一定是按存储器中的顺序存储的。当堆变得分散时，它们可能会分散在不同的地方。这使得
数据缓存效率低下。

动态内存分配也会使代码更复杂。程序必须保留指向所有分配对象的指针，并在它们时保持跟踪
不再使用。重要的是，所有被分配的对象也要在所有可能的位置上被释放
程序流的情况。不这样做是一个常见的错误来源，称为内存泄漏。一个更担忧的错误是在一个对象被释放后访问它。这个
程序逻辑可能需要额外的开销开销。

请参见第页为了进一步讨论使用动态内存分配的优点和缺点。

有些编程语言，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

在类中声明的变量存储在它们出现在类声明中的顺序中。存储的类型是在声明类的对象时确定的。类、结构或联合的一个对象可以使用上面提到的任何存储方法。除了一些简单的情况外，一个对象不能被存储在一个寄存器中，但它的数据成员可以被复制到寄存器中。

带有staticmodifier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与该类的每个实例一起使用。

将变量存储在类或结构中是一种很好的方法，可以确保在程序的同一部分中使用的变量也存储在彼此附近。参见第页52使用类的利弊。

7.2整数变量和运算符

整数大小

整数可以是不同的大小，它们可以是有符号的，也可以是无符号的。下表总结了可用的不同整数类型。

声明	尺寸，位	最小值	最大值	在stdint.h中
查尔	8	-128	127	int8_t
短int 在16位系统中：int	16	-32768	32767	int16_t
int 在16位系统中：长int	32	-231	231-1	int32_t
long long orint64_tMS编译器： int64 64位Linux：长int	64	-263	263-1	int64_t
无符号字符	8	0	255	uint8_t
无符号短int 在16位系统中：无符号int	16	0	65535	uint16_t
无符号int 在16位系统中：无符号长	32	0	232-1	uint32_t
无符号长长oruint64_t MS编译器：未签名ed长int	64	0	264-1	uint64_t

表7.1.dif铁整数类型的大小

不幸的是，声明一个特定大小的整数的方法是不同的
平台，如上表所示。建议使用标准数据头文件输入。h orinttypes.hfor是一种定义特定大小的整数类型的可移植方法。

整数运算在大多数情况下，无论大小。但是，输出比最大可用寄存器大小大的整数大小是低效的。换句话说，它对16位系统中的32位整数或32位系统中的64位整数的计算效率低下，特别是如果代码涉及乘法或除法。

编译器总是选择声明int的整数，而不指定大小。较小的整数（字符，短int）只稍微小一些高效。在许多情况下，编译器在进行计算时会将这些类型转换为默认大小的整数，然后只使用结果的低位8或16位。你可以
假设类型转换需要零个或一个时钟周期。在64位系统中，只要你不做除法，32位整数和64位整数的效率之间只有很小的差异。

建议在大小不重要的情况下使用defaultinteger大小
并且没有溢出的风险，例如简单变量、循环计数器等。在largearrays中，最好使用对于特定的
目的为了更好地利用datacache.Bit字段dsof sizes不是8，16，

32位和64位效率较低。在64位系统中，如果应用程序可以利用额外的位，则可以使用64位整数。

unsignedinteger类型size_size_type在32位系统中为32位，在64位系统中为64位。当您想要确保overflow时，它适用于数组大小和数组索引
即使对于大于2GB的阵列，也不会发生这种情况。

当考虑一个特定的整数大小对于一个特定的目的是否足够大时，你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，即使a、b、c和d都低于最大值，（b*c）也可能溢出。没有自动检查整数溢出。

有符号整数与无符号整数

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。但有几种情况很重要：

·被常数除法：当你用一个常数除一个整数时，Unsignedis比signed快（见第#页）。150).这也适用于模运算符%。

•对于大多数指令集，有符号整数转换为浮点比无符号整数更快。

·溢出对有符号变量和无符号变量的行为不同。的溢出
无符号变量产生低阳性结果。有符号变量的溢出是
官方未定义。正常行为是将正溢出包装到
负值，但编译器可能会基于溢出不会发生的假设，优化依赖于onoverflow的分支。

有符号整数和无符号整数之间的转换是免费的。这只是对相同的部分进行不同的解释的问题。当转换为无符号整数时，负整数将被解释为非常大的正数。

```
//示例7.4。有符号和无符号整数
int a，b；
双c；
b=(无符号整数)A/10；//转换为无符号以便快速除法
c=a*2.5；//转换为double时使用signed
```

在示例中 7.4为了使除法更快，我们将a转换为无符号。的
当然，这只有在确定一个人永远不会受益的情况下才有效。最后一行是隐式的
在乘以常数2.5之前将a转换为double，常数2.5是e的两倍。在这里，我们更喜欢ato besigned
。

确保不要混合有符号和无符号整数的比较，如<。的结果
比较有符号整数和无符号整数是不明确的，可能会产生不期望的结果。

整数运算符

整数运算通常非常快。简单的整数运算，如加法，
在大多数微处理器上，减法、比较、位运算和移位运算只需要一个时钟周期。

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
其他微处理器上需要3-4个时钟周期。整数
根据微处理器的不同，除法需要40-80个时钟周期。整数除法是
在AMD处理器上，整数大小越小，速度越快，但在Intel处理器上则不然。Details

关于指令延迟列在手册4：“指令表”中。关于如何加快乘法和除法的提示在第页给出149
和分别为150。

递增和递减运算符

增量前运算符++i和增量后运算符++与加法一样快。当简单地用于递增整数变量时，
无论使用前递增还是后递增都没有区别。效果完全相同。例如，
for（i=0；i<n；i++）与for（i=0；i<n；++i）相同。但是如果使用表达式的结果，则可能存在不同的低效率。例如，
x=array[i++]比x=array[++i]更有效，因为在后一种情况下，数组元素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pre-increment更改为post-increment，则必须调整i的初始值。

也有预增量比后增量更有效的情况。为了
例如，在A=++b的情况下；编译器将识别出a和b的值在此语句之后是相同的，因此它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，因此它们不能使用相同的寄存器。

这里所说的关于递增运算符的一切也适用于整数变量的递减运算符。

7.3浮点变量和运算符

x86系列的现代微处理器有两种不同类型的浮点寄存器和相应的两种不同类型的浮点指令。每种类型都有
优点和缺点。

进行浮点运算的原始方法包括八个浮点寄存器，它们被组织成一个寄存器堆栈，称为x87寄存器。这些寄存器有长双
精度（80位）。使用寄存器堆栈的优点是：

·所有计算均以长倍精度完成。

·不同精度之间的转换不需要额外时间。

•对数和三角函数等数学函数有内在的说明。

·代码是紧凑的，并且在代码缓存中占用很少的空间。寄存器堆栈
也有缺点：

·由于寄存器堆栈的组织方式，编译器很难创建寄存器变量。

•浮点比较很慢。

·整数和浮点数之间的转换效率低下。

•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来计算。

进行浮点运算的一种新方法涉及向量寄存器（XMM、YMM、或ZMM），它们可以用于多种目的。浮点运算完成于
单精度或双精度以及中间结果始终使用与操作数相同的精度计算。使用向量寄存器的优点是：

•创建浮点寄存器变量很容易。

•向量运算可用于对向量寄存器中的多个变量进行并行计算（参见第115).

缺点是：

•不支持长双精度。

•运算数具有混合精度的表达式的计算需要精度转换指令，这可能是相当耗时的（参见153).

数学函数必须使用函数库，但这通常比内部硬件函数快。

x87浮点寄存器可用于所有具有浮点的系统
功能（64位Windows的设备驱动程序除外）。XMM、YMM和ZMM
寄存器分别需要SSE、AVX和AVX512指令集。参见第页135关于如何测试这些指令集的可用性。

现代编译器将使用向量寄存器进行floa tingpoint计算，只要它们可用，即在64位模式下或当SSE或更高指令集启用时。很少
编译器能够混合这两种类型的浮点运算，并为每个计算选择最佳的类型。

在大多数情况下，只要不连接到向量，双精度计算不会比单精度计算花费更多的时间。当使用XMM寄存器时，单精度除法、平方根和数学函数的计算速度比双精度快，
而加法、减法、乘法等的速度。当不使用向量运算时，无论精度如何，在大多数处理器上仍然是相同的。

如果对应用有好处，您可以使用双精度，而不必太担心成本。如果您有大数组并且不想获得同样多的数组，则可以使用单精度
尽可能将数据放入数据缓存中。如果您可以利用向量运算的优势，单精度是很好的，如第页所述115.半精度可用于具有AVX512-FP16指令集扩展的处理器。

浮点加法需要2-6个时钟周期，具体取决于微处理器。
乘法需要3-8个时钟周期。除法需要14-45个时钟周期。当oldx87
使用浮点寄存器。

不要在同一个表达式中混合单精度和双精度。参见第页153.

如果可能的话，避免整数和浮点变量之间的转换。参见第页153.

在向量寄存器中生成浮点tunderflow的应用程序可以受益于设置刷新到零模式，而不是在Ounderflow的情况下生成低于正常数：

```
//示例7.5。设置冲洗至零模式（SSE）：
#include<xmmintrin.h>
__ _
```

除非有特殊原因，否则强烈建议设置刷新到零模式
使用次正规数。此外，如果向量寄存器可用，您可以设置denormals-are-zero模式：

```
//示例7.6。设置刷新为零和非常态为零模式（SSE2）：
#include<xmmintrin.h>
_mm_setcsr(_mm_getcsr()0x8040);
```

参见第页131和158了解更多关于数学函数的信息。

7.4枚举

枚举只是一个伪装的整数。枚举和整数一样有效。

请注意，枚举数（值名）将与任何同名的变量或函数冲突。因此，头文件中的Enumsin应该具有长且唯一的枚举器
名称或放入名称空间中。

7.5布尔

布尔操作数的顺序

布尔运算符&&和的操作数按以下方式计算。如果&&的第一个操作数为false，则根本不计算第二个操作数，因为
同样，如果的第一个操作数为真，则不计算第二个操作数，因为结果已知为真。

将最常为真的操作数放在&&表达式的最后，或者放在表达式的第一位可能是有利的。例如，假设ais在50%时间为真，且b为
10%的时间是真的。当a为真时，表达式a&&b需要计算b，这是50%的情况。等价表达式b&&A仅当b为真时才需要计算A，这只有10%的时间。如果A和B花费相同的时间来评估并且同样可能被分支预测机制预测，这会更快。参见第页43关于b牧场预测的解释。

如果一个操作数比另一个更容易预测，那么把最容易预测的操作数放在第一位。

如果一个操作数的计算速度比另一个快，则将计算速度最快的操作数放在第一位。

但是，在交换Booleanoperands的顺序时必须小心。如果操作数的计算有副作用，或者如果第一个操作数确定第二个操作数是否有效，则不能交换操作数。例如：

```
//示例7.7
无符号int i；常量int数组大小=100；浮点列表【数组大小】；
如果（i<数组大小&&列表[i]>1.0）{...
```

这里，您不能交换操作数的顺序，因为当i不小于ARRAYSIZE时，expressionlist[i]无效。另一个例子：

```
//示例7.8
if(handle! =invalid_handle_value&&WriteFile(handle,...)){...
```

这里不能交换布尔操作数的顺序，因为如果句柄无效，就不应该调用WriteFileif。

布尔变量超定

布尔变量存储为8位整数，值为0表示false，值为1表示true。

布尔变量是超确定的，因为所有以布尔变量为输入的运算符都会检查输入是否有除0或1之外的任何其他值，但以布尔变量为输出的运算符不能产生除0或1之外的任何值。这使得像putless中那样使用布尔变量的操作比必要的效率更高。举个例子：

```
//示例7.9 a
布尔A、b、c、d;
c=A&&b;
d=ab;
```

这通常由编译器通过以下方式实现：

```
布尔A、b、c、d;
如果(a! =0){
    如果(b! =0){
        c=1; }
    否则{
        gotoCFALSE; }
}
否则{
    CFALSE:
    c=0; }
如果(a==0){
    如果(b==0){
        d=0;
    }
    否则{
        转到DTRUE}
}
否则{
    DTRUE:
    d=1; }
```

这当然远非最佳。分支可能需要很长时间，以防
错误预测（见第43).如果可以肯定地知道操作数除了0和1之外没有其他值，布尔运算可以变得更加有效。原因
编译器不做这样的假设的原因是，如果变量未初始化或来自未知来源，它们可能有其他值。上面代码可以是
OptimizedIf a和b已被初始化为有效值，或者它们来自produceBoolean输出处的运算符th。优化后的代码如下所示：

```
//示例7.9 B
字符a=0，b=0，c，d;
c=A&b;
d=ab;
```

在这里，我使用char（或int）而不是boolin顺序，以便可以使用
按位运算符（&and）而不是布尔运算符（&&and）。按位
运算符是只需要一个时钟周期的单个指令。OR运算符()有效

即使a和b具有0或1以外的值。如果操作数具有0和1以外的值，AND运算符(&)和异或运算符(^)可能会给出不一致的结果。

请注意，这里有一些陷阱。你不能用~forNOT。相反，您可以通过与1异或对已知为0或1的变量进行布尔NOT：

```
//示例7.10 a
布尔A，b；
b=！A；
```

可以优化为：

```
//示例7.10 b
字符a=0，b；
b=a^1；
```

如果b是一个如果为假就不应该计算的表达式，则不能用A&b替换A&b。同样，如果b是一个如果为真就不应该计算的表达式，则不能用a b替换a b。

如果操作数是变量，则使用按位运算符的技巧比操作数是比较等更有利。例如：

```
//示例7.11
布尔A；浮动x，y，z；
a=x>y&&z！=0；
```

这在大多数情况下是最佳的。不要更改&&t o&除非您希望&&表达式生成许多分支错误预测。

布尔向量运算

整数可以用作布尔向量。例如，如果a和b是32位整数，则表达式y=a&b；将在一个时钟周期内执行32次和-操作。运算符&、、^、~对于布尔向量运算很有用。

7.6指针和引用

指针与引用

指针和引用同样有效，因为它们实际上在做同样的事情。示例：

```
//示例7.12
void FuncA(int*p){
    *p=*p+2；}

void FuncB(int&r){r=r+2
    ；
}
```

这两个函数做的事情相同，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是完全相同的。这个
差异只是一个编程风格的问题。使用指针路径比引用的优点是：

当您查看上面的函数时，很明显p是一个指针，但不清楚它是一个引用还是一个简单的变量。使用指针可以让读者更清楚地知道到底发生了什么。

可以使用没有引用时不可能使用的指针。您可以更改指针所指向的内容，并且可以使用指针进行算术运算。

使用r等量值而不是指针的优点是：

当使用引用时，语法性更简单。

·引用比指针使用起来更安全，因为在大多数情况下，它们必须指向一个有效的地址。如果指针未初始化，如果指针算术计算超出了有效地址的范围，或者如果指针被类型转换为错误的类型，指针可能是无效的并导致致命错误。

•引用对于复制构造函数和重载运算符非常有用。

·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需要变量。

效率

通过指针或引用访问变量或对象可能与
直接访问它。这种效率的原因在于微处理器的工作方式
建造的。函数中声明的所有非静态变量和对象都存储在堆栈上，实际上是相对于堆栈指针进行寻址的。
在类中声明的变量和对象通过名为“this”的隐式指针访问。因此，我们可以得出结论，结构良好的DC++中的大多数变量
程序实际上是通过指针以这样或那样的方式访问的。因此，微-
处理器必须被设计成使指针高效，这就是它们的本质。

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。寄存器是一个
稀缺资源，尤其是在32位模式下。如果没有足够的寄存器，那么每次使用时都会从内存中加载指针，这将使程序变慢。
另一个缺点是指针的值在可以访问所指向的变量之前需要几个时钟周期。

指针算法

指针是保存内存地址的整数。算术运算要点
因此与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：

```
//示例7.13
结构abc{int a; int b; int c; };
abc*p; 核心;
p=p+1;
```

这里，加到p上的值不是i而是i*12，因为abc的大小是12字节。因此，加法所需的时间等于乘法和加法所需的时间。如果abc的大小是2的幂，那么乘法可以被更快的移位操作代替。在上面的例子中，abc的大小可以是
通过向结构中添加一个整数，将字节增加到16字节。

递增或递减整数不需要乘法，只需要加法。比较两个指针只需要整数比较，速度很快
。

计算两个指针之间的差值需要除法，除非所指向的对象的typeof的大小是2的幂（参见150关于除法）。

在计算出指针的值之后大约两个时钟周期可以访问指向的对象。因此，建议计算a的值指针在使用指针之前。例如，x=*(p++)比
因为在后一种情况下，x的读取必须等到指针phas递增后的几个时钟周期，而在前一种情况下，x可以在p递增之前读取。参见第页关于递增和递减运算符的更多讨论，见图31。

7.7函数指针

如果目标地址可以被预测，通过函数指针调用函数可能比直接调用函数更需要几个时钟周期。如果函数指针的值与上次执行该语句时相同，则会预测目标地址。这个
如果函数后交换的值，目标地址很可能会被错误预测。一个错误的预测会导致很长时间的延迟。请参见第页43关于分支预测。

7.8成员指针

在简单的情况下，数据成员指针简单地存储了数据成员相对于对象的开头的偏移量，而成员函数指针只是对象的寻址
元函数但也有一些特殊的情况，比如多重遗传
还需要更复杂的实现。这些复杂的病例绝对应该避免。

如果一个编译器有关于成员指针所引用的类的不完整信息，那么它就会兜售最复杂的成员指针的实现。例如：

```
// Example7.14
类c1;
int c1::* MemberPointer;
```

在这里，编译器除了当时的名称外，没有关于c1的信息
已声明Memberpointis。因此，它必须假设最可能的情况
使成员指针的复杂标记。这可以通过在成员指针之前声明c1的完整声明来避免。避免多重继承，
虚拟函数和其他使成员指针无效的并发症。

大多数C-++编译器都有各种选项来控制成员指针的运行方式
实现如果可能的话，使用提供最简单实现的选项，并确保对使用相同的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类似于一个指针的对象。它有一个特殊的特点
当指针被删除时，它指向的对象也被删除。智能指针对于存储在动态分配内存中的对象来说很方便。使用smart的目的
指针用于确保对象被正确删除，并在不再使用对象时释放内存。智能指针可以被认为是实现adata容器或动态大小数组的简单方法，而无需定义containerclass。

C++11标准定义了智能指针asstd：：unique_ptr和
std：：shared_ptr。std：：unique_ptr具有这样的特性：总是有一个，并且只有一个指针拥有分配的对象。所有权从一个转移

unique_ptrto another by assignment.shared_ptr允许指向同一个对象的多个指针。

通过智能指针访问对象没有额外的成本。无论pi是简单指针还是智能指针，通过*por p->members访问对象都同样快。目标
每当智能指针被创建、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的这些成本高于
唯一_ptr。

在简单的情况下，最好的优化编译器（Clang，Gnu）可以剥离unique_ptr的大部分或全部开销，这样编译后的代码只包含对new和delete的调用。

在程序的逻辑结构要求对象必须由一个函数动态创建，然后由另一个函数删除的情况下，智能指针可能很有用
函数和这两个函数互不相关（不是同一个函数的成员
类）。如果同一个函数或类负责创建和删除对象，那么就不需要智能指针。

lfa对许多动态分配的小对象进行编程，每个对象都有自己的智能指针
然后你可以考虑这个解决方案的成本是否太高。将所有对象一起汇集到单个容器中可能更有效，最好是使用contiguousmemory。请参阅第页上关于容器类的讨论97.

7.10阵列

数组只需将元素连续存储在内存中即可实现。没有
存储有关数组维度的信息。这使得在C和C++中使用arr ays比在其他编程语言中更快，但也更不安全。这个保险箱
可以通过定义一个容器类来克服这个问题，该类的行为类似于具有边界检查的arr ay，
如本例所示：

```
//example7.15 a.带边界检查的数组
模板<typename T,unsigned int N>class SafeArray{
受保护：
   Ta[N]；//包含N个类型为T的元素的数组
公众：
   SafeArray(){//构造函数
      memset（a，0，sizeof（a））；//初始化为零}
   int Size(){//返回数组的大小
      返回N；}
   T&operator[](unsigned int i){//Safe[]数组索引运算符if(i>=N){
         //索引超出范围。下一行引发错误。
         //您可以在此处插入任何其他错误报告：
         返回*(T*)0；//返回空引用以引发错误
      }
      //没有错误
      返回a[i]；//返回对[i]的引用
   }}
;
```

给出了更多容器类的示例页面下方97.

通过将类型和大小指定为
模板参数，例如下面7.15 b示出。它是用一个正方形的br acketsindex访问的，就像一个普通的数组一样。构造函数将所有元素设置为零。您可以删除

如果您不想要此初始化，或者如果类型T是一个带有defaultconstructor的类，它执行必要的aryinitialization。编译器可以报告memset是
已弃用。这是因为如果size参数错误，它可能会导致错误，但它仍然是
将数组设置为零的最快方法。如果索引超出范围，[]运算符将检测到错误（请参见147关于边界检查）。在此引发错误消息
返回空引用的非常规方式。这将引发错误
如果访问了数组元素，则在受保护的操作系统中显示消息，并且使用调试器很容易跟踪此错误。您可以用任何其他形式的错误替换此行
报告。例如，在Windows中，您可以writeFatalAppExitA（0，“数组索引超出范围”）；或者更好的是，创建自己的errormessage函数。

以下示例说明了如何使用SafeArr ay：

```
//示例7.15 b
SafeArray<float，100>列表；//生成100个浮点的数组
for(int i=0;i<list.size();i++){//遍历数组
    cout<<list[i]<<endl；//输出数组元素
}
```

由列表初始化的数组最好是静态的，如第26.可以使用memset将数组初始化为零：

```
//示例7.16
浮点列表[100]；
memset(list，0，sizeof(list))；
```

多维数组的组织方式应使lastindex变化最快：

```
//示例7.17
常量int行=20，列=50；
浮点矩阵【行】【列】；
英蒂，j；floatx；
    对于(i=0;i<行;i++)
        对于(j=0;j<列;j++)
            矩阵[i][j]+=x；
```

这确保了元素被顺序访问。两个循环的相反顺序将使访问不连续，从而降低数据缓存的效率。

如果以无序顺序索引行，则除了第一维度之外的所有维度的大小可以优选地是2的幂，以便使地址计算更有效：

```
//示例7.18
int FuncRow（int）；int FuncCol（int）；
常量int行=20，列=32；
浮点矩阵
int i；浮动x；
为（i=0；i<100；i++）
    矩阵[FuncRow(i)][FuncCol(i)]+=x；
```

在这里，代码必须计算（FuncRow(i)*列+FuncCol(i)）*
sizeof（浮点），以找到主题元素的地址。当柱的幂为2时，柱的乘法更快。在前面示例中，这不是问题，因为优化编译器可以看到访问第e行
并且可以通过将一行的长度添加到前一行的地址来计算每一行的地址。

同样的建议也适用于结构或类主题的数组。的大小（字节）
如果元素以连续顺序访问，对象最好为2。

使列数为2的建议并不总是适用于大于1级数据缓存并不按顺序访问的序列，因为它可能会导致缓存争论。请参见第页 为了讨论这个问题。

7.11类型转换

C-++语法有几种不同的类型版本制作方法：

```
// Example7.19
inti; floatf;
f=i; //隐式类型转换
f=(浮动）i; //C型式铸造
f=浮点数(i); //构造函数样式的类型铸造
f=stici_cast<浮点>(i); //C++铸造运算符
```

这些不同的方法也有完全相同的效果。您使用哪种方法是一个编程风格的问题。下面讨论不同类型转换的时间消耗。

签名/无签名转换

```
// Example7.20
inti;
if((unsignedint)i<10){...
```

有符号整数和无符号整数之间的转换简单地使编译器以不同的方式解释整数的位。没有关于漫游流的检查，而代码需要
没有额外的时间。这些转换可以自由使用，而没有任何成本不性能。

整数大小转换

```
// Example7.21
短的inti;
i=s;
```

如果整数是有符号的，则通过扩展有符号位将整数转换为更长的大小；如果整数是无符号的，则通过扩展零位将整数转换为更长的大小。如果源是算术表达式，这通常需要一个时钟周期。如果在
与从内存中的变量读取值的连接，如示例中所示7.22.

```
//示例7.22
短int a[100]; int i, sum=0;
对于（i=0; i<100; i++）sum+=a[i];
```

将整数转换为更小的大小只需忽略高位即可。没有溢出检查。示例：

```
//示例7.23
核心；短整数;
s=(shortint)i;
```

这种转换不需要额外的时间。它只存储32位整数的低16位。

浮点精度转换

float、double和long doubl之间的转换不需要额外的时间
使用浮动点器堆栈。当使用XMM寄存器时，它需要2到15个时钟周期（取决于处理器）。
请参见第页31号文件的解释
寄存器堆栈与XMM寄存器之间的关系。样例

```
// Example7.24
浮a；双b；
a+=b;
```

在本例中，如果使用XMM寄存器，则转换是代价的。a和b应该属于相同的类型，以避免这种情况。请参见第页153进行进一步讨论。

用于浮动转换的整数

将签名整数转换为浮动双重整数需要4 -16个时钟周期，这取决于处理器和所使用的寄存器类型。未签名整数的转换取
使用时间比有符号整数更长，除非启用了AVX512指令集（AVX512DQfor64位无符号整数）。首先将未符号整数转换为有符号整数的速度更快
如果没有溢出的风险，则为整数：

```
// Example7.25
无符号的u，双d；
d=（双）（签名int）u；//更快，但有溢出的风险
```

整数到浮点的转换有时可以通过用浮点变量替换整数变量来避免。示例：

```
//示例7.26 a
浮点A[100]；int i；
对于（i=0；i<100；i++）a[i]=2*i；
```

本例中ITO floating的转换可以通过添加一个额外的floatingpoint变量来避免：

```
//示例7.26 B
浮点A[100]；int i；浮点i2；
对于（i=0，i2=0；i<100；i++，i2+=2.0f）a[i]=i2；
```

浮点到整数的转换

将浮点数转换为整数需要很长时间，除非启用了2或更高版本的指令集。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，因此浮点舍入模式必须更改为截断和反向增益。

如果在代码的关键部分有浮点到整数的转换，那么对此做些什么是很重要的。可能的解决方案是：

通过使用不同类型的变量来避免转换。

通过将中间结果存储为浮点，将转换移出最里面的循环。

使用64位模式或启用SSE2指令集（需要支持此功能的微处理器）。

使用舍入代替截断，并使用汇编语言创建一个舍入函数。参见第页154有关舍入的详细信息。

指针类型转换

指针可以转换为不同类型的指针。
转换为整数，或者整数可以转换为整数。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代码。这意味着以不同的方式解释相同的位或绕过语法检查。

当然，这些转换并不安全。程序员有责任确保结果有效。

重新解释对象的类型

可以使编译器处理一个变量或对象当作它有一个不同的类型，按它的地址：

```
// Example7.27
浮子x;
*（int*）&x|=0x80000000；//设置x的符号位
```

这里的语法似乎有点奇怪。xistype的地址-被转换为指向
整数，然后取消这个指针，以便作为整数访问x。这个
编译器不会产生任何额外的代码来实际制作一个指针。指针是
简单地优化，结果是xis视为一个整数。但是和操作员
强制编译器存储x内存而不是在区域。上面的示例通过使用|操作符来设置x的符号位，否则该操作符只能应用于整数。它比x=-abs(x)快；。

当使用类型铸造指针时，需要注意许多危险：

这个技巧违反了严格的标准c的相同规则，指定不同类型的两个指针不能指向同一个对象（除了字符指针）。一
优化编译器可以在两个不同的寄存器中存储浮动点和整数表示。您需要检查编译器是否做了您想要做的事情。使用工会更安全，例如14.22页155.

如果客观对象比实际更大，诀窍就会失败。如果int使用的位数大于浮点数，那么上面的代码将会失败。（both位32位inx86系统）。

如果您访问一个变量的部分，例如64位双的32位，那么该编码将无法移植到您设置的环境存储的平台。

如果您访问一个可变的内置，例如您在atime上写一个64位双32位，那么代码的执行速度可能比预期的要慢
转发CPU的延迟（参见手册3：“Intel、AMD和VIACPU的主题架构”）。

常量铸造

Theconst_castoperator用于缓解指针的限制。它需要进行语法检查，因此比没有ec风格的类型转换更安全
添加任何额外的代码。样例

```
// Example7.28
类c1{
    常量数据，常量数据
    平民
    c1（）：x(0){}；//构造函数将x初始化为0
    空白x+2（）{//这个功能可以修改x
```

```
*const_cast<int*>（&x）+=2；}//添加2到x}；
```

这里的theconst_cast算符的效果是去掉了对x的常量限制。它可以了语法限制，但它不生成任何额外的代码，也不需要任何额外的时间。这是一个确保一个函数可以修改x的有用方法，而其他函数则不能。

静态铸件

static_castoperator执行与C样式类型转换相同的操作。例如，它用于将float转换为int
。

重新诠释演员阵容

其中，Interpret_CastOperator用于指针转换。它做同样的sC风格的类型转换，只是多了一点语法检查。它不产生ny外码。

动态铸造

Dynamic_CastOperator用于将指向一个类的指针转换为指向另一个类的指针。它进行运行时检查转换是否有效。例如，当指向基类的apointer转换为指向派生类的apointer时，它会检查originalpointer是否实际指向派生类的对象。这张支票使

dynamic_cast比简单的类型转换更耗时，但也更安全。它可能会捕捉编程错误，否则这些错误不会被检测到。

转换类对象

只有当程序员定义了构造函数、重载赋值运算符或重载赋值运算符时，转换才可能涉及类对象（而不是指向对象的指针）。

指定如何执行转换的加载类型转换运算符。构造函数或重载运算符与成员函数一样有效
。

7.12分支和开关语句

现代微处理器的高速是通过使用流水线获得的，其中

指令在被执行之前在几个阶段被获取和解码。然而，管道结构有一个大问题。每当代码有一个分支（例如anif-elsesstructure）时，微处理器事先不知道将两个分支中的哪一个馈送到流水线中。如果错误的分支被fedinto到管道，则不会检测到错误

直到10-20个时钟周期之后，它在这段时间内通过获取、解码以及可能推测性地执行指令所做的工作都被浪费了。结果是，每当微处理器将一个分支馈送到

管道，后来发现它选择了错误的分支。

微处理器设计者已经竭尽全力来减少这个问题。最

使用的重要方法是分支预测。现代微处理器正在使用

高级算法根据该分支和附近其他分支的过去历史来预测该分支的走向。用于分支预测的算法对于每种类型的微处理器是不同的。这些算法在手册3：“英特尔、AMD和VIACPUs的微架构”中有详细描述。

在微处理器做出正确预测的情况下，Abranch指令通常需要0-2个时钟周期。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，具体取决于处理器。这被称为分支错误预测惩罚。

如果大部分时间都被预测到，分支相对便宜，但如果经常被错误预测，分支就会变得昂贵
。总是走同一条路的分支是可以预测的

课程大多数时候是单向的，很少是相反的
只有当它相反时，才会被错误预测。一条可以单向运行很多次的分支，
很多时候，只有在变化时才会被错误预测。一个遵循简单周期性模式的分支也可以很好地预测，如果它是循环内的分支。例如，一个简单的周期性模式可以是单向两个时间，另向三次。然后是两次，三次，等等。最坏的情况是一个分支随机单向或另一个50-50
任何方向的机会。这样的分支在50%的情况下会被误诊。

一个为循环或当循环也是一种分支。每次迭代后，它都会决定是否这样做
重复或退出该循环。循环分支通常是被预测的，如果重复计数是
小的，而且总是一样的。可以预测的最大循环数在9到64之间，取决于处理器。嵌套的循环只有在
someprocessors.在许多处理器上，包含多个分支的循环没有很好的预测。

切换语句是一种可以进行两个以上的分支。开关语句是最有效的，这些标记遵循一个序列，其中每个标签等于
前面的标签加上1，因为它可以实现一个跳跃目标表。带有大量标签的替代语句是低效
的
因为编译器必须转换为空白树。

在较老的处理器上，带有顺序标签的开关语句被简单地预测为与上次执行时相同。因此，无论它比上次不同，肯定会被错误预测。较新的处理器有时能够预测一个开关状态，如果它遵循一个简单的周期模式，或者如果它与之前的分支相关，并且不同目标的数量很小。

分支和开关语句的数量最好保持较小
一个程序的关键部分，特别是如果分支是不可预测的。如下一段所述，如果这可以消除分支，那么推出一个循环可能是有用的。

分支和函数调用的目标保存在一个称为分支的特殊命令中
targetbuffer.如果一个程序有很多，分支目标缓冲区可能会发生约束
分支或函数调用。这种竞争的结果是分支可能被错误预测，即使它们在其他情况下会被很好地预测。由于这个原因，即使是直接的函数调用也可能是不可预测的。在
因此，代码的关键部分可能会受到错误预测的影响。

在某些情况下，可以用tablelookup替换不可预测的分支。例如：

```
//示例7.29 a
浮动A；布尔b；
A=b？1.5 F：2.6 F；
```

那个？：这里的操作员是分支机构。如果它很难预测，那么用表查找替换它：

```
//示例7.29 B
浮动A；布尔b=0；
常量浮点查找[2]={2.6 f，1.5 f}；
A=查找[b]；
```

如果bool被用作数组索引，那么确保它被初始化或来自可靠的来源是很重要的，这样它就不能有0或1以外的值。参见第页34.

在某些情况下，编译器可以自动用条件移动替换分支。

页面上的示例147和图148示出了减少分支数量的各种方式。

手册3：“英特尔、AMD和VIACPUs的微体系结构”给出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预测循环的能力
控制分支。有关分支预测的解释，请参见上一段和手册3：“英特尔、AMD和威盛CPU的微体系结构”。一个重复计数小且固定且内部没有分支的循环可以被完美地预测。如上所述，
可以预测的最大循环计数取决于处理器。嵌套循环是
仅在某些具有特殊循环预测器的处理器上预测良好。在其他方面
处理器，只有innermostloop预测良好。具有高重复计数的Aloop是
只有在退出时才会出现错误预测。例如，如果循环重复一千次，那么循环控制分支只有千分之一的错误预测，因此错误预测惩罚对总执行时间的贡献可以忽略不计。

循环展开

在某些情况下，展开aloop可能是一个优势。示例：

```
//示例7.30 a
核心;
对于（i=0; i<20; i++）{
    如果(i%2==0){
        FuncA(i);
    }
    否则{
        FuncB(i);
    }
    FuncC(i)
; }
```

这个循环重复20次，依次调用FuncA和FuncB，然后调用FuncC。展开循环两个给出：

```
//示例7.30 b
核心;
对于（i=0; i<20; i+=2）{FuncA（i）
    ;
    FuncC(i);
    FuncB(i+1);
    FuncC(i+1)
; }
```

这有三个优点：

·i<20循环控制分支被执行10次而不是20次。

·重复计数从20减少到10，这意味着在大多数CPU上可以很好地预测它。

·if分支被消除。

循环展开也有缺点：

●unrolledloop在代码缓存或微操作缓存中占用更多空间。

·许多CPU都有一个环回缓冲区，可以提高非常小循环的性能，正如我的微体系结构手册中所解释的那样。展开的循环不太可能适合
环回缓冲区。

·如果重复计数是奇数，并且你将其展开2，那么就有一个额外的迭代必须在循环之外完成。一般来说，当重复计数不能被展开因子整除时，你就会遇到这个问题。

只有在可以获得特定的辅助年龄时，才应该使用循环展开。如果aloop包含浮点计算或向量指令，并且循环计数器是整数，那么通常可以假设总计算时间由浮点代码决定，而不是由循环控制分支决定。没有任何好处
在这种情况下展开循环。

当存在循环携带的依赖链时，循环展开很有用，如第页所述113.

在具有微操作高速缓存的处理器上最好避免循环展开，因为节省微操作高速缓存的使用是重要的。这也是ca se
带有环回缓冲区的处理器。

编译器通常会自动展开aloop，如果这样做看起来有利可图的话（参见72).程序员不必手动展开循环，除非有特定的
要获得的优势，例如在示例中消除if分支7.30 b.

循环控制条件

最有效的循环控制条件是一个简单的整数计数器。微处理器无序能力（见第113）将能够在几次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，效率就会降低。以下示例将以零结尾的ASCII字符串转换为低ercase：

```
//示例7.31 a
字符串[100]，*p=字符串；
while(*p!=0)*(p++)|=0x20;
```

如果字符串的长度是已知的，那么使用环计数器更有效：

```
//示例7.31b
字符串字符串[100]，*p=字符串；int i，字符串长度；
为（i=弦长度；i>0；i--）*（p++）|=0x20；
```

循环控制分支的一种常见情况下，依赖于循环内的计算是数学迭代，如泰勒扩展和牛顿-拉夫森迭代。
在这里，重复迭代，直到残差低于一定的公差。这个
计算残差的绝对值并将其与
公差可能如此之高，以至于确定最大情况更有效
重复计数，并始终使用这个迭代次数。该方法的优点是，微处理器可以提前执行循环控制分支，并解决任何问题
在环内的浮点计算完成之前的分支错误预测。该方法有利于在最大代表吃计数附近的典型重复计数，并且每次迭代的残差计算对总计算时间有显著贡献。

循环计数器最好应该是整数。lfa环需要一个浮动的点计数器，然后做一个额外的点计数器。样例

```
// Example7.32a
双x，n，阶乘=1.0；
用于（x=2.0；x<=n；x++）阶乘*=x；
```

这可以通过添加一个整数计数器并使用循环控制条件中的整数来改进：

```
//示例7.32b
双x，n，阶乘式=1.0；int i；
用于（i=（int）n-2，x=2.0；i>=0；i--，x++）阶乘*=x；
```

请注意循环中的逗号和分号之间的区别，例如　　7.32 b.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，而每个子句内的多个语句
用逗号分隔。条件子句中应该只有一条语句。

将整数与零进行比较有时比将其与任何其他整数进行比较更有效
号码。因此，使循环计数到零比
使它计数到某个正值，n。但是如果循环计数器被用作arrayindex。数据缓存针对向前访问数组而不是向后访问数组进行了优化。

复制或清除数组

对于诸如复制数组或将数组设置为全零等琐碎任务，使用aloop可能不是最佳选择。示例：

```
//示例7.33 a
常量int大小=1000；int i；
浮动A【大小】，b【大小】；
//将A设置为零
对于（i=0；i<大小；i++）a[i]=0.0；
//将A复制到b
对于（i=0；i<大小；i++）b[i]=a[i]；
```

使用函数memset和memcpy通常更快：

```
//示例7.33 B
常量int大小=1000；
浮动A【大小】，b【大小】；
//将A设置为零
memset（a，0，sizeof（a））；
//将A复制到b
memcpy（b，a，sizeof（b））；
```

大多数编译器会自动通过调用memset和memcpy来替换这样的循环，至少在简单的情况下是这样。显式使用memset和memcpy是不安全的，因为如果sizeparameter大于destinationarray，就会发生许多错误。但同样
如果循环计数太大，循环可能会发生错误。

7.14功能

函数调用可能会减慢程序的速度，原因如下：

·函数调用使微处理器跳转到不同的代码地址并返回。这可能需要多达4个时钟周期。在大多数情况下，micr处理器可以将调用和返回操作与其他计算重叠以节省时间。

·如果代码碎片化并分散在内存中，代码缓存将无法有效工作。

·函数参数以32位模式存储在堆栈上。将参数存储在堆栈上并再次读取它们需要额外的时间。延迟是显著的ifa
参数是关键依赖链的一部分。

•设置堆栈帧、保存和恢复寄存器以及可能保存异常处理信息都需要额外的时间。

·每个函数调用语句占用分支目标缓冲区(BTB)中的空间。
如果程序的关键部分有许多调用和分支，BTB中的争用会导致分支错误预测。

以下方法可用于减少程序关键部分中函数调用所花费的时间。

避免不必要的功能

一些编程教科书建议每个长于几行的函数都应该分成多个函数。我不同意这条规则。将一个函数拆分成多个更小的函数只会提高无程序的效率。仅仅因为函数长就拆分它并不能使程序更清晰，除非函数正在做
多重逻辑上不同的任务。如果可能的话，关键的innermostloop最好完全保留在一个函数内。

使用内联函数

内联函数像宏一样展开，这样调用函数的每个语句都被函数体替换。如果在类定义中定义了函数的主体，则函数通常是内联的。内联函数是有利的
函数很小，或者只从程序中的一个地方调用。小函数通常由编译器自动内联。另一方面，在某些情况下，如果内联函数导致技术问题或性能问题，编译器可能会忽略内联函数的请求。

避免在最里面的循环中嵌套函数调用

调用其他函数的函数称为框架函数，而不调用任何其他函数的函数称为叶函数。叶函数比框架函数更有效，原因见第页63.如果程序的关键innermostloop
包含对框架函数的调用，那么代码可能可以通过内联框架函数或通过内联所有
它调用的函数。

使用宏代替函数

用#defineis声明的Amacro肯定会内联。但请注意，每次使用宏参数时都会对其进行评估。示例：

```
//example7.34 a.使用宏作为内联函数
#defineMAX(a,b)(a>b? a: b)
y=MAX（f（x），g（x））；
```

在此示例中，f(x)或g(x)被计算两次，因为宏引用了ittwice。

您可以通过使用内联函数而不是宏来避免这种情况。如果您希望函数使用任何类型的参数，请将其设置为模板：

```
//示例7.34 b.用模板替换宏
模板<类型名称>
静态内联Tmax（T const&A，T con st&b）{返回A>b? A: b
   ;
}
```

宏的另一个问题是名称不能重载或限制范围。A
宏将干扰任何具有相同名称的函数或变量，无论作用域或名称空间如何。因此，使用长而独特的名称是很重要的，
尤其是在头文件中。

使用fastcall和vectorcall函数

关键字fastcall在32位模式下更改函数调用方法，以便前两个整数参数在寄存器中而不是在堆栈中传输。这可以提高整数参数函数的速度。

浮点参数不受快速调用的影响。中的隐式“this”指针
成员函数也被视为参数，因此可能只有一个自由寄存器
用于传输附加参数。因此，当您使用fastcall时，请确保最关键的integerparameter放在第一位。函数参数是
在64位模式下通过defau ltin在寄存器中传输。因此，在64位模式下无法识别
fastcallkeyword。

关键字VectorCall更改Windows系统中浮点和向量操作数的函数调用方法，以便在向量中传递和返回参数 _
寄存器。这在64位窗口中尤其有利。微软、英特尔和Clang编译器目前支持vectorcall方法。

使函数局部化

仅在同一模块中使用的函数（即当前。cppfile）应该是本地的。这使得编译器更容易内联函数和优化
跨函数调用。有三种方法可以使functionlocal：

1.将关键字staticto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，因为statichas有不同的含义。

2.将函数或类放入anonymousnamespace中。

3.GNU编译器all ows“__attribute__((visibility(“hidden”)))”。

使用整个程序优化

一些编译器有一个选项来优化整个程序或将多个。cpp文件合并成一个目标文件。这使得编译器能够优化所有的寄存器分配和参数传递。组成程序的cppmodules。整个程序优化不能用于作为对象或库文件分发的函数库。

使用64位模式

参数传输在64位T模式下比在32位模式下更有效，在64位Linux中比在64位Windows中更有效。在64位Linux中，前6个整数参数和前8个浮点参数被传送到寄存器中，总共多达14个
寄存器参数。在64位窗口中，前四个参数在寄存器中传输，无论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更有效。在这方面，32位Linux和32位Windows没有区别。

可用于浮点和向量变量的寄存器数量在32位模式下为8个寄存器，在64位模式下为16个寄存器。它进一步增加到32个64位寄存器
启用AVX512指令集时的模式。大量的寄存器可以提高性能，因为编译器可以将变量存储在寄存器中而不是内存中。

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局部变量。这对于简单类型（如int、float、double、bool、enum以及指针和引用）非常有效。

数组总是作为指针传递，除非它们被包装到类或结构中。

情况更复杂，参数具有复合类型，如Structure或Class。如果满足以下所有条件，则传递复合类型的参数是最有效的
符合条件：

·对象非常小，可以放入单个寄存器中。
•对象没有cop y构造函数，也没有析构函数
·对象没有虚拟成员
·对象不进行用户运行时类型标识(RTTI)

如果不满足这些条件中的任何一个，那么将指针或引用传递到对象通常会更快。如果对象很大，那么复制整个对象显然需要时间。将对象复制到theparameter时，必须调用Anycopy构造函数，并且
析构函数（如果有）必须在函数返回之前调用。

将复合对象传递给函数的首选方法是通过常量
参考。常量引用确保原始对象未被修改。与inter或非const引用不同，const引用允许函数参数是表达式或匿名对象。编译器可以很容易地优化一个常量
引用如果函数已内联。

另一种解决方案是使函数成为对象的类或结构的成员。这同样有效。

在32位系统中，简单的函数参数在堆栈上传输，但在64位系统中，在寄存器上传输。后者效率更高。64位Windows最多允许四个
要在寄存器中传输的参数。64位UNIX系统最多允许14个
要在寄存器中传递的参数（8个浮点或双精度加6个整数、指针或
参考参数）。成员函数中的thispointer计算一个参数。
手册5：“不同C++编译器和操作系统的调用约定”给出了更多细节。

7.16函数返回ty pes

函数的返回类型最好是一个简单的类型，aInter，areference或void。返回复合类型的对象更复杂，而且通常效率很低。

只有在最简单的情况下，才能在寄存器中返回复合类型的对象。参见手册5：“不同C++编译器和操作系统的调用约定”
何时可以在寄存器中返回对象的详细信息。

除了最简单的情况，复合对象是通过将它们复制到调用者通过hiddenpointer指示的位置来返回的。复制构造函数（如果有）通常是
在copyingprocess中调用，在销毁原件时调用析构函数。在简单的情况下，编译器可能能够避免调用复制构造函数和
析构函数通过在其最终目的地上构造对象，但不计算它。

您可以考虑以下替代方案，而不是返回复合对象：

·使函数成为对象的构造函数。

·让函数修改一个现有的对象，而不是创建一个新的对象。The
现有对象可以通过指针或引用对函数可用，或者函数可以是对象类的成员。

·使函数返回对在
功能。这很有效，布特里斯基。返回的指针或引用仅有效
直到下一次调用函数并覆盖本地对象，可能在不同的线程中。如果您忘记将本地对象设置为静态，那么函数一返回，它就会变得无效。

·使函数构造一个具有new的对象，并返回一个inter。由于动态内存分配的成本，这是低效的。如果您忘记删除对象，此方法还涉及内存泄漏的风险。

7.17函数尾部调用

尾部调用远离优化函数调用。如果函数的最后一次执行是调用
另一个函数，那么编译器可以用跳转到第二个函数来替换调用。优化编译器会自动执行此操作。第二个函数不会返回到第一个函数，而是直接返回到调用第一个函数的地方。这更多高效，因为它消除了回报。示例：

```
//Example7.35.尾部呼叫
void function2(int x);

无效函数1(int y){...
    函数2(y+1); }
```

这里，通过直接跳转到Function2，消除了Function1的返回。即使有返回值：

```
//Example7.36.带有返回值的尾部调用
int function2（int x）;

int函数1（int y）{...
    返回函数2(y+1); }
```

只有当两个函数具有相同的返回类型时，尾部调用优化才有效。如果函数在堆栈上有参数（这通常是32位模式下的情况），那么这两个函数必须为参数使用相同数量的堆栈空间。

7.18递归函数

递归函数是一个自用的函数。递归函数调用可以对
处理了递归的数据结构。递归函数的代价，所有参数和局部变量为每个递归得到一个新的实例，这占用了堆栈空间。预测也使返回地址的预测无效。这个问题
通常出现在递归级别大于16的递归级别上（参见手册3中对返回堆栈缓冲区的解释：“Intel，AMDD和VIACPUs的微架构”）。

递归函数调用仍然可以是处理分支数据树结构的最有效的解决方案。树结构的递归效率比深度更高。A
非分支递归总是可以被循环所取代，这是非常有效的。递归函数的一个典型的教科书
例子是阶乘函数：

```
// Example7.37.阶乘式作为递归函数
无符号长int阶乘（无符号int n）{如果（n<2）返回1;
   返回n*阶乘（n-1）；}
```

这种实现效率非常低，因为nand的所有实例和所有返回地址都占用堆栈上的存储空间。使用aloop更有效：

```
//Example7.38.作为循环的阶乘函数
unsigned long int阶乘(unsigned int n){unsigned
   long int product=1;
   而(n>1){product*=n
      ;
      n--
   ; }
   返回产品}
```

递归尾部调用比其他递归调用更有效，但仍然不如aloop有效。

新手程序员有时会调用main来重启他们的程序。这是一个错误的想法，因为每次递归调用main时，堆栈都会被所有局部变量的新实例填满。重启程序的正确方法是在main中循环。

7.19结构和类

如今，编程教科书推荐面向对象编程作为使软件更加清晰和模块化的一种手段。所谓的对象是
结构和类。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：

如果一起使用的变量是相同结构或类的成员集，那么它们也会存储在一起。它使数据的变化更有效。

作为类成员的变量不需要作为参数传递给类成员函数。避免了这些变量的参数转移开销。

面向对象编程的协调效果有：

有些程序员正在将代码划分为太多的小类。这是低效的。

非静态成员函数去掉一个“此”指针，它作为隐含参数传递到函数。“This”的参数传输开销不涉及所有非静态成员函数。

“这个”指针占用一个寄存器。寄存器是32位系统中的一种稀缺资源。

虚拟成员函数的效率较低55).

对于积极的还是消极的影响
面向对象的编程占主导地位。至少，我们可以说，类和成员函数的使用并不昂贵。您可以使用一种面向对象的编程样式

如果程序的逻辑结构和程序结构良好，只要你避免在程序最关键的部分使用过多的功能。结构的使用（没有成员功能）对性能有非阴性的影响。

7.20类数据成员（实例变量）

类或结构的数据成员连续存储在顺序中，每当创建类或结构的一个实例时，它们就会被声明。没有
将数据组织成类或结构的性能损失。访问数据
一个类或结构对象的成员所花费的时间并不比访问一个可简单转移的对象更多。

大多数编译器将对齐数据成员绑定地址，以优化访问，如下表所示。

类型	大小，字节	对齐，字节
布尔	1	1
字符，已签名或未签名	1	1
短程、有符号或无符号	2	2
签名，签名或签名	4	4
64位整数，有符号或无符号	8	8
指针或参考，32位模式	4	4
指针或参考，64位模式	8	8
使漂浮	4	4
两倍的	8	8
长双	8,10,12 or16	8 or16
表7.2.数据成员的对齐情况。		

这种对齐可以在具有混合大小的成员的结构或类中产生未使用的字节孔。例如：

```
// Example7.39a
结构S1{
    短的2字节。第一个字节在0，最后一个字节在1//6未使用的字节
    双b；//8字节。第一个字节在8处，最后一个字节在处15
    4个字节。第一个字节在16处，最后一个字节在19
                    //4个未使用的字节
};
S1 ArrayOfStructures[100];
```

这里，在a和b之间有6个未使用的b ytes，因为bhas从一个地址开始
能被8整除。末尾还有4个未使用的字节。其原因是数组中S1的下一个实例必须从可被8整除的地址开始，以便对齐其b
成员由8。通过将最小成员放在最后，可以将tes未使用的数量减少到2：

```
//示例7.39 B
结构S1{
    双b；//8字节。第一个字节在0，最后一个字节在7
    int d；//4字节。第一个字节在8，最后一个字节在11
    短int a；//2字节。第一个字节在12，最后一个字节在13
                    //2个未使用的字节
};
S1结构阵列[100]；
```

这种重新排序使结构缩小了8个字节，数组缩小了800个字节。

结构和类对象通常可以通过重新排序数据成员来变小。如果
类至少有一个virtualmember函数，然后有一个指向虚拟表的指针
在第一个数据成员之前或在最后一个成员之后。此指针在32位中为4字节
系统和64位系统中的8字节。如果你对biga结构或它的每个成员有疑问，那么你可以用
sizeofoperator做一些测试。sizeofoperator返回的值包括对象末尾的任何未使用的字节。

如果数据成员相对于结构或类的开头的偏移量小于128，则用于访问数据成员的代码更紧凑
，因为偏移量可以是
表示为8位有符号数字。如果相对于structureor类开头的偏移量是128字节或更多，则偏移量
必须表示为32位数字（指令集在8位和32位偏移量之间没有任何值）。示例：

```
//示例7.40
S2类{
   公众：
   int a[100]；//400字节。第一个字节at0，最后一个字节at399
   int b；//4字节。第一字节at400，最后字节at403
   int ReadB（）{return b；}}
；
```

这里是bis400的偏移量。访问b通过指针或成员的任何代码
函数，如readb需要编码偏移as一个32位数。如果a和b是
交换后，可以使用编码为8位signed号的偏移量来访问两者，或者根本没有偏移量。这使得代
码更加紧凑，从而可以更有效地使用代码缓存。因此，建议大数组和其他大对象出现激光结
构或类声明，最常用的数据服务器首先出现。如果不可能在前128个字节内包含所有的数据
成员，那么将最常用的成员放在前128个字节中。

7.21类成员函数（方法）

每次声明或创建类的新对象时，将生成数据成员的新实例。但是每个成员函数只有一个实
例。不复制函数代码，因为相同的代码可以应用于该类的所有实例。

调用内存函数与使用指针或引用结构调用简单函数一样快。例如：

```
// Example7.41
类S3{
   平民
   inta;
   int b;
   int Sum1（）{返回一个+b；}
};
int Sum2（S3*p）{返回p->a+p->b；}
int Sum3（S3&r）{返回r.a+r.b；}
```

sum1，sum2，sum3和它们完全在做同样的事情
同样有效。如果您查看由该计算器生成的代码，您将会注意到，一些计算器将为这三个函数
生成完全相同的代码。Sum1有一个隐式
“这个”指针，它和p和r inSum2和Sum3做相同的事情。无论您是想让函数成为类的成员，还
是给它一个指向类的指针或引用还是
结构只是一个节目风格的问题。一些编译器通过在寄存器中传输“这个”，使32位窗口中的
Sum1比Sum2和Sum3稍微高一些
比在堆栈上。

静态成员函数不能访问任何非静态数据服务器或非静态数据服务器
成员函数。一个统计的标准成员函数比另一个静态的成员函数更快，因为它不需要“这个”指针。如果它们不需要任何非静态访问，您可以使它们更快地运行。

7.22虚拟成员功能

虚拟函数用于实现多态类。a的每个实例
多态类有一个指向指向虚拟函数的不同版本的指针表的指针。这个所谓的虚拟表用于查找虚拟表的有效性
函数在运行时。多态性是面向对象程序比非面向对象程序更有效的主要原因之一。如果你可以避免虚拟函数
然后，您就可以在不支付性能成本的情况下获得面向对象编程的大部分优势。

调用虚拟成员函数所需的时间比调用虚拟成员函数所需的时间多，只要函数调用语句总是调用虚拟函数的相同版本的时钟周期。如果版本改变了，那么您可能会得到一个10-20个时钟周期的错误预测惩罚。虚拟函数调用的预测和错误预测方法与开关语句相同，如第页所述44.

当调用虚拟函数时，可以绕过调度机制
已知类型的对象，但您不能总是依赖于编译器来绕过分派机制，即使它很明显要这样做。请参见第页76.

只有当在编译时不知道多态成员函数的哪个版本时，才需要运行时多态性。如果在程序的关键部分中使用了虚拟函数，那么您可以考虑是否有可能获得所需的功能
没有多态性或有编译时多态性。

有时可以用模板代替虚函数来获得所需的多态性效果。templateparameter应该是一个包含以下函数的类
有多个版本。这种方法更快，因为模板参数总是在编译时而不是在运行时解析。示例第7.47页图59示出了
如何做到这一点的例子。不幸的是，语法是如此混乱，它可能不值得努力。

7.23运行时类型识别(RTTI)

运行时typeidentification向所有类对象添加了额外的信息，并且效率很低。如果编译器有一个RTTI选项，那么关闭它并使用替代实现。

7.24继承

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

代码缓存和数据缓存可能会出现退化，原因如下：

子类包括父类的所有数据成员，即使它不需要它们。

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的数据成员的代码

字节稍微那么紧凑。请参见第页54.

父和子的函数可能存储在不同的模块中。这可能会导致很多人的跳跃和较低效率的编解码器。这个
问题可以通过确保相互调用的函数也相互存储来解决。请参见第页92的细节。

从同一代的多父类继承可能会导致并发症
使用成员指针和虚拟函数，或在通过指向其中一个基类的指针访问派生类的对象时。您可以通过在派生类中创建对象来避免多重继承：

```
// Example7.42a.多重继承
B1类：B2类；
D类：公共B1、公共B2{
平民
   intc;
};
```

替换为：

```
//示例7.42b.可替代多重继承
B1类：B2类；
类D：公共B1{
平民
   B2b2;
   intc;
};
```

一般来说，如果有利于程序的逻辑结构，就应该使用继承。

7.25构造函数和析构函数

构造函数作为内部作为函数返回对对象的引用。为新对象的内存分配不一定是由
构造函数本身。因此，构造函数与任何其他成员函数都一样有效。这适用于默认的构造函数、复制构造函数和任何其他构造函数。

一个类不需要一个构造函数。如果对象不需要初始化，则不需要一个默认的构造函数。复制构造函数不能通过复制对象而简单地复制。一个简单的构造函数可以排队进行改进
表演

当赋值复制对象时，可以调用复制构造函数
函数参数，或作为一个函数返回值。复制构造函数可以是一个时间
消费者，如果它涉及到内存或其他资源的分配。有各种方法可以避免这种浪费的备忘录单块复制，例如：

使用指向该对象的引用或指针，而不是复制它

使用“移动结构器”来转移记忆块的所有权。这需要++0x或以后。

使一个成员函数或朋友函数或操作符，以转移的所有权
从一个对象到另一个对象的内存块。失去内存块所有权的对象的指针应该设置为NULL。当然应该有一个
破坏对象所拥有的任何内存块的析构函数。

析构函数和内存函数一样有效。如果它不是析构函数，则不要创建它
必要的虚拟析构函数是一个有效的虚拟成员函数。请参见第页55.

7.26工会

单位是一种数据成员共享相同内存空间的结构。一个联盟可以是用于存储内存空间，因为它允许两个从未在同一时间使用过的数据成员共享同一段内存。请参见第页以94为例。

Aunion还可以用来以不同的方式访问相同的数据。样例

```
// Example7.43
union{
    浮动f;
    inti;
}x;
x.f=2.0f;
x.i|=0x80000000; //设置f的符号位
cout<<x.f; //将给-2.0
```

在本例中，fis的符号位是使用按位OR运算符设置的，该运算符只能是应用于整数。

7.27位字段

位字段可能有助于使数据更加紧凑。访问位字段的成员比访问结构的成员效率低。额外的时间可能是合理的
大型数组的情况，如果它可以节省缓存空间或使文件更小。

通过使用<<和操作来编写位字段比单独编写成员更快。示例：

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

假设a，Band Care的值太小而不会导致溢出，则可以通过以下方式改进此代码：

```
//示例7.44 B
联合位字段{
    结构{
        int a: 4;
        int b: 2;
        int c: 2;
    };
    字符abc;
};
Bitfieldx;
int A、B、C;
x.abc=a(B<<4)(C<<6);
```

或者，如果需要防止过流：

```
//示例7.44 c
```

```
x.abc=(a&0x0F)|((B&3)<<4)|((C&3)<<6);
```

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重载运算符

重载运算符相当于一个函数。使用重载运算符和使用做同样事情的函数一样有效。

具有多个重载运算符的表达式将导致为中间结果创建临时对象，这可能是不希望的。
示例：

```
//示例7.45 a
类向量{                                                    //2维向量
公众：
    浮动x，y；                                          //x，y坐标
    vector(){}                                          //默认构造函数
    向量（浮点a，浮点b）{x=a；y=b；}                    //构造函数
    向量运算符+(向量常数A){                             //和运算符
        返回向量（x+a.x，y+a.y）；}                     //添加元素
};

载体A、b、c、d；
A=b+c+d；//使(b+c)的中间对象
```

通过加入以下操作，可以避免为中间结果（b+c）创建一个临时对象：

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

幸运的是，大多数编译器将在简单操作中自动进行此优化。

7.30模板

模板与宏类似，即在编译之前模板参数被它们的值替换。下面的示例说明了函数参数和模板参数之间的区别：

```
// Example7.46
int乘(intx、intm)

template< int m>
int MultiplyBy ( int x){
    返回

int a，b；
a=乘法（10,8）；
b= MultiplyBy<8>(10);
```

a和b都将得到阀门e10 * 8 =80。不同之处在于m转换到函数的方式。在简单函数中，在运行时从调用器转移到称为函数。但是在模板函数中，m在编译时被它的值所取代，因此它看到的是常量8而不是变量m。使用模板参数而不是函数参数的优点是参数传输的开销是

避开缺点是编译器需要删除一个新的实例
模板函数的每个不同值的模板参数。如果这个例子被许多不同的因素调用作模板参数，那么代码就可以
变得非常大。

在上面的示例中，模板函数比简单函数更快，因为
编译器知道它可以通过使用移位操作将幂2相乘。x*8是
替换为x<<3，速度更快。在简单函数的情况下，编译器不知道mand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，并将80简单地放入a和b中。但在更复杂的情况下，它可能无法这样做）。

模板参数也可以是类型。页面上的示例图38显示了如何用相同的模板制作不同类型的数组。

模板是高效的，因为模板参数总是在编译时解析。模板会使源代码更加复杂，但不会使编译后的代码更加复杂。在
一般来说，使用模板在执行速度方面没有成本。

如果模板参数为
一丝不差如果模板参数不同，那么您将为每个模板参数集获得一个实例。具有许多实例的模板使编译的代码占用大量的空间。

过度使用模板会使代码难以阅读。Ifa模板只有一个
实例，您可以使用#定义、常量或类型，而不是模板参数。

模板可用于元编程，如第页所述163.

使用模板进行多态性

模板类可以用于实现宏堆时间多态性，它比使用虚拟成员函数获得的运行时多态性更有效。下面的示例首先显示了runtime多态性：

```
// Example7.47a.具有虚拟函数的运行时多态性
类CHello{
平民
   void非多态（）；//非多态函数在这里
   虚拟空间Disp（）；//虚拟函数
   空白你好（）{
      考特<<"你好"；
      Disp（）；//调用到虚拟函数
   }}
；

类C1：公共CHello{公共：
   虚拟空白Disp（）{ cout<<
      1；
   }}
；

类C2：公共CHello{公共：
   虚拟空白Disp（）{ cout<<
      2；
   }}
；
```

```
空测试（）{
    C1 Object1;C2 Object2;
    CHello*p;
    p=& Object1;
    p-> NotPolymorphic();    //直接调用//写道
    p->你好（）；             "Hello1"
    p=& Object2;
    p->你好（）               //写道"Hello2"
; }
```

如果编译器不知道对象p指向的类别，那么分配到光盘（）或C2：：光盘（）是在运行时完成的(见页76).最好的编译器可能能够优化远离p和在线调用对象1。你好，（），比如这样的话。

如果在编译时已知该对象是否属于类c1orc2，那么我们可以
避免低效的虚拟函数分派过程。这可以通过在活动模板库（ATL）和Windows模板库（WTL）中使用的一个特殊技巧来实现：

```
//示例  7.47b.使用模板的编译时间多态性

//胎盘-多形态的功能：
类CGrandParent{
平民
    空白无多态性（）；
};

//任何需要调用多形形式化函数的函数都位于//父类中。子类作为模板参数提供：
template<typenameMyChild>
类是公共的祖父母{
平民
    空白你好（）{
        考特<<"你好"；
        //调用多态子函数：
         （stitc_cast<MyChild*>（这个））->Disp（）；
    }}
；

//子类实现的函数有多个函数
//版本：
公共类<CChild1>{公共类：
        空白显示（）
         {cout<<1；
    }}
；

Chald2>{公共：
        空白显示（）
         {cout<<2；
    }}
；

空测试（）{
    CChild1 Object1; CChild2 Object2;
    CChild1*p1;
    p1=& Object1;
    p1->Hello（）；//写"你好1"
    CChild2*p2;
    p2=&对象2；
    p2->Hello()；//写"Hello2"
```

```
}
```

这里CParentis是一个模板类，它通过
模板参数。它可以调用其子类的多态成员，方法是将其“this”指针转换为其子类的指针。只有当它有正确的子类名作为模板参数时，这才是安全的。换句话说，您必须确保声明

```
class CChild1: public CParent<CChild1>{
```

子类名和模板参数的名称相同。

继承顺序如下。第一代类（CGrandParent）
包含任何非多态成员函数。第二代班
（CParent<>）包含需要调用非态函数的任何成员函数。第三代类包含多态函数的不同版本。这个
第二代类通过图模板参数获取关于第三代类的信息。

在向已知对象类的虚拟成员函数分派上不浪费时间。这个信息包含在具有不同类型的p1和p2中。CParent：：Hello（）有多个占用高速空间的实例。

联联蛋白示例7.47b诚然是非常混乱的。我们通过避免虚拟函数调度机制而保存的少数时钟周期很少足以证明这种难以理解的复制代码，因此很难立即实现。如果
编译器能够进行异常化(请参见页面76)自动地说，那么依赖编译器优化肯定比兜售这个复杂的模板更方便
方法

7.31线程

线程用于同时或同时执行两个或多个任务。现代cpu拥有多个核，使运行多个线程成为可能同时地每个线程将得到通常30 ms的前景作业和
当线程多于CPU内核时，请使用10个ms作为后台作业。上下文
每个时间片之后的开关都是相当昂贵的，因为所有的机器机器都必须适应新的上下文。可以通过延长更长的时间来减少上下文切换的数量
部分这将使应用程序运行得更快，但代价是用户输入的响应时间会延长。

线程对于为不同的任务分配不同的优先级很有用。例如，在文字处理器中，用户期望立即响应按下一个键或移动它们。这项任务应具有较高的优先级。其他的任务，如拼写检查和重分页，都在其他优先级较低的线程中运行。如果不同的任务没有被分为以下两种情况：
具有不同优先级的线程，那么当程序忙于进行拼写检查时，用户可能会经历对键盘和鼠标输入的长时间响应。

任何需要很长时间的任务，比如重数学计算，都应该是
如果应用程序有一个图形化的用户界面，则在一个单独的线程中进行调度。否则，该程序将无法快速响应键盘或鼠标输入。

可以在应用程序中进行类似线程的调度，而无需调用操作系统线程调度程序的开销。这可以通过在一个函数中分段进行重背景计算来完成，该函数从图形用户界面的消息循环中调用（WindowsMFC中的OnIdle）。这个方法可能会更快
而不是制作一个具有少量cpucore的单独线程系统，但它要求背景工作可以划分为一个适当持续时间的小部分。

充分利用具有多个CPU共读的系统来将作业划分为多读的最佳方法。然后，每个线程都可以在它自己的CPUcore上运行。

在优化多线程应用程序时，我们必须考虑四种多线程的成本：

启动和停止线程的成本。如果持续时间与启动和停止线程的时间相比较短，则不要将任务放入单独的线程中。

任务切换的成本。如果相同的线程数量不超过cpucore的数量，这个成本将最小化。

在线程之间进行同步和通信的成本。的overh
信号量互相互音等。是相当大的。如果两个线程经常相互等待以获得对相同资源的访问，那么最好将它们连接到一个线程中。在多个线程之间共享的变量必须声明为不稳定的变量。这可以防止编译器存储在线程之间不共享的多个变量。

不同的线程需要分离存储。没有被使用的函数或类
多个线程应该使用静态变量或全局变量。（请参见线程本地存储器。28)这些线程去掉了各自的堆栈。如果线程共享同一缓存，这可能会导致缓存存在争议。

多线程程序必须使用对线程进行安全保护的函数。一个线程安全的函数不应该使用静态变量。

请参阅章节10页110次，以便进一步讨论多线程技术。

7.32例外情况和错误处理

运行时错误会导致可以通过陷阱或软件的形式检测到的异常
打断这些例外可以通过尝试捕获。程序将崩溃，一个错误消息，并且没有尝试模块。

异常处理是用于检测很少发生的错误和恢复
以一种优雅的方式设置错误条件。您可能认为异常处理不需要额外的费用
时间，只要错误不发生，但幸运的是，这并不总是短暂的。这个
程序可能需要做大量的簿记，以知道恢复的事件
的一个例外。这种记账的成本在很大程度上取决于编译器。一些
编译器具有高效的基于表的方法，具有很少的开销，而其他
编译器可以使用低效的基于代码的方法或需要运行时类型标识（RTTI），这会影响代码的其他部分。看ISO/IECTR18015C++技术报告性能，以便进一步解释。

下面的例子说明了为什么需要进行簿记：

```
// Example7.48
类C1{
    平民
    ...
    ~C1();
};

空白F1（）{
    C1x;
    ...
```

```
}

voidF0(){ t
   ry{
      F1();}
   捕获（...）{
   ...
   }}
```

函数F1应该在返回时调用对象x的析构函数。但是如果在inF1的某个地方发生了异常呢？然后我们就冲出1而不返回。F1被阻止清理，因为它被中断了。现在是
exceptionhandler负责调用x的析构函数。只有当1保存了有关析构函数调用或任何其他可能的清理的所有信息时，这才是可能的
必须的。如果F1调用另一个函数，该函数又调用另一个函数，等等，并且如果
异常发生在最里面的函数中，那么异常处理程序需要有关函数调用链的所有信息，dit需要通过
函数调用来检查所有必要的清理工作待办事项。这称为堆叠展开。

所有函数都必须为异常处理程序保存一些信息，即使没有发生异常。这就是为什么异常处理在某些情况下可能非常昂贵的原因
编译器。如果您的应用程序不需要异常处理，那么您应该
禁用它以使代码更小、更高效。您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。您可以通过向函数原型添加Throw()来禁用单个函数的exceptionhandling：

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

这允许编译器假设F1永远不会抛出任何异常，因此它不必为functionF1保存恢复信息。然而，如果F1调用了另一个可能引发异常的函数F2，那么F1必须检查F2行中的异常，并在F2实际引发异常的情况下调用STD：：unexpected()函数。因此，只有当所有函数都被F1调用时，您才应该将emptythrow()规范应用于F1
还有一个emptythrow()规范。emptythrow()规范对库函数很有用。

编译器可以区分了叶函数和框架函数。一个框架
函数是至少调用一个函数的函数。前叶函数是一个函数
不调用任何其他函数。叶函数比框架函数更简单，因为堆栈展开信息可以排除异常，但可以排除或存在异常
没有例外情况需要清理的。通过排列所有函数进行调用，可以将帧函数转换为叶函数。
最好的性能是一个程序的最内层循环不包含呼叫帧函数。

虽然在某些情况下，（）语句可以在某些情况下的优化，但之前没有理由添加像抛出（A、B、C）这样的语句来显式地告诉什么类型的异常
函数可以抛出。事实上，编译器实际上可以添加外代码来检查抛出的异常确实是指定的类型(参见萨特：实用程序查看异常规范，　　多布斯博士杂志，2002年)。

在某些情况下，即使是在a的最关键的部分，也最好使用异常处理
程序这就是案例中，替代实现的效率较低，您希望能够从错误中恢复。以下实例说明了这些酶：

```
// Example7.49
//门户网站注意：这个例子是特定于微软编译器的。
//，它在其他编译器中看起来会有所不同。
# include< excpt.h>
# include< float.h>
# include< math.h>
#定义EXCEPTION_ FLT_ OVERFLOW0 xC0000091L

空白数学循环（）{
    常量int参数大小=1000；无符号int虚拟人；
    双a，b，c；

    //启用浮点溢出的异常：
    _ controlfp_s（&虚拟，0，_EM_溢出）；
    //_控制文件（0，_EM_溢出）；//如果上面的行不工作

    inti=0；//在两个循环外部初始化循环计数器
    //当循环的目的是在异常后恢复：
    而（i< arraysize）{
        //捕获此块中的异常：
        __try{
            //计算的主环：
            （；我<array大小；我++）{

                //溢出可能发生在乘法在这里：
                一个[i]=日志（b[i]*c[i]）；}
        }
        //捕获流上的浮点，但没有其他：
        __除了(获取异常代码（）==EXCEPTION_FLT_溢出
        ?EXCEPTION_ EXECUTE_ HANDLER： EXCEPTION_ CONTINUE_
            SEARCH){//发生浮点溢出。
            //重置浮点数状态：
            _ fpreset();
            _ controlfp_s（&虚拟，0，_EM_溢出）；
            //_控制文件（0，_EM_溢出）；//如果以上不工作

            //以一种避免溢出的方式重新做计算：
            [i]=日志（b[i]）+日志（c[i]）；

            //增量循环计数器，并返回到for循环：
            i++;}
    }
}
```

假设数字b和c太大，溢出可能发生在
乘法b[i]*c[i]，尽管这种情况很少发生。上面的代码将捕获一个
异常在溢出的情况下，以需要更多时间的方式重做计算，但是
避免溢出。取每个因子的对数而不是乘积可以确保不会发生溢出，但计算时间会增加一倍。

支持异常处理所需的时间是微不足道的，因为在关键的innermostloop中没有tryblock或函数调用（除了log）。逻辑a
我们优化的库函数。无论如何，我们都不能改变它可能的异常处理支持。异常发生时代价高昂，但这不是问题，因为我们假设这种情况很少发生。

测试循环内部的溢出条件在这里不需要任何成本，因为我们
依赖于微处理器硬件在溢出情况下引发异常。操作系统捕获异常，并将其重定向到中的异常处理程序
该程序有一个tryblock。

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。

让我们看看这个例子中异常处理的可能替代方案。我们可以通过在将b[i]和c[i]相乘之前检查它们是否太大来检查溢出。这会
需要两个浮点比较，这相对昂贵，因为它们必须
在innermostloop里面。另一种可能性是总是使用安全的公式A[i]=
log（b[i]）+log（c[i]）；。这将使记录的调用次数增加一倍，并且
对数需要很长时间来计算。如果不需要检查循环外的溢出，而不检查所有的数组元素，那么这可能是一个更好的解决方案。如果所有因素都由相同的低参数生成，则有可能进行这样的检查。或者可能在结果循环后进行检查
结合一些公式，得到一个罪恶的结果。

异常和向量代码

向量指令对于不并行地进行多个计算非常有用。这在第章中描述12下。异常处理不能很好地处理向量代码，因为一个向量元素可能导致异常，而其他向量元素可能导致异常
不你甚至可能在另一个分支中产生一个例外
所有的分支都是通过向量代码来实现的。如果代码可以从向量指令中获益
然后最好禁用例外的捕获和继续NA N和英芬斯特德的传播。请参阅章节 7.34以下。这将在文档中进一步讨论
万维网。阿格纳。org/optimize/nan_propagation。pdf.

避免异常处理的成本

当没有尝试从错误中恢复时，不需要异常处理。如果你只是想让程序发出一个错误消息，并在出错时停止程序，那么就没有理由使用try、catch和throw。定义您的
自己的错误处理函数，它简单地打印一个适当的错误消息，然后调用exit。

如果存在需要清理的已分配资源，则调用exit可能不安全，
如下所述。还有其他可能的方法可以在不使用
例外。检测到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来恢复或发出错误消息。

建议使用系统且经过深思熟虑的错误处理方法。您必须区分可恢复和不可恢复的错误；确保出现错误，清除分配的资源，并向用户发出适当的错误消息。

制定异常安全代码

假设一个函数打开一个排他模式，并且一个错误条件在文件关闭之前终止程序。在程序终止后，该文件将继续被锁定，并且在计算机重新启动之前，用户将无法访问该文件。你必须使你的程序变得异常安全。换句话说，程序必须清理一切，以防出现异常或其他错误情况。可能的事情
需要进行清理，其中包括：

用新成员分配的记忆。

处理窗口，图形笔刷等。

沙痂锁着的短信。

打开数据库连接。

打开文件和网络连接。

需要被删除的临时文件。

需要进行扩展的用户工作。

任何其他已分配的资源。

处理清洁++的方法。一个可以读取或
写入文件可以包装成一个具有构造器的类，使文件关闭。相同的方法可以用于任何其他资源，例如动态分配
内存、窗口、交互项、数据库连接等。

C++异常处理系统确保本地对象的所有析构函数都是
呼叫程序异常安全如果有析构函数的包装类
负责所有已分配资源的清理工作。如果析构函数导致另一个异常，则系统很可能会失败。

如果您使用自己的错误处理系统而不是使用异常处理，那么您就不能确保所有析构函数都被调用和资源清理。如果一个n个错误处理程序调用退出（），中止（），_endthread（）等。那么就不能保证所有的东西了
析构函数被称为。处理不可恢复错误的安全方法
从函数返回的异常。如果可能，函数可以返回一个错误代码，或者错误代码可以存储在globalobject中。然后调用函数必须检查错误代码。如果后一个函数也有一些东西需要向上倾斜，那么它必须尽快返回到它的调用者。

7.33叠层退绕的其他情况

前一段描述了一种称为堆叠展开的机制，该机制由
exceptionhandlers，用于在不使用正常returnroute的异常情况下，在跳出函数后清理和调用y必要的析构函数。这种机制也用于另外两种情况：

当线被终止时可以使用堆叠退绕机构。目的是检测线程中声明的任何对象是否有需要调用的析构函数。建议在终止线程之前从需要清理的函数返回。您不能确定对_endthread()的调用是否会清理堆栈。这种行为是
依赖于实现。

当函数longjmp被使用时，堆栈展开机制被启动
跳出一个函数。尽可能避免使用长臂键。不要依赖于longjmp内部关键代码。

7.34NAN和INF的传播

浮点错误将传播到一系列计算的最终结果。这是一个非常有效的替代例外和错误捕获
。

浮点溢出和零分裂得到无穷大。如果你添加或使用无限
结果你会得到无限的东西。INF代码可能会以此方式传播到最终结果。然而，并不是所有带有hINF输入的操作都会影响INF。如果你把a

你得到零。特殊案例和INF给NAN(不-

一个数字)。当您将0除以零时，也会出现特殊的codeNAN，比如assqrt（-1）和log（-1）
。

大多数带有NAN输入的操作将给出aNAN输出，因此NAN将传播到最终结果。这是一种简单的检测浮动点点的方法。几乎所有的浮点错误都将传播到最终结果，它们出现为IN F orNAN
。如果
你打印结果出来，你将看到答案或答案而不是编号。没有额外的代码
需要跟踪错误，并且没有提取INF和NAN的传播。

ANAN可以包含带外带的有效载荷。函数库可以将错误的错误代码放入这个有效加载中，这个有效负载将传播到最终结果。

当参数为INF或NAN时，函数有限的（）将返回为假，如果它是非正交的浮点数，则为真。这可以用于在浮点数之前检测错误
数字被转换为一个整数，在其他情况下，我们想检查祖先。

INF和NAN传播的细节在文档“NAN
浮点代码中的传播与故障捕获
www.agner.org/optimize/nan_propagation.pdf。本文还讨论了INF和NAN传播失败的情况，以及影响这些代码传播的编译器优化选项。

7.35预处理指令

预处理指令（所有以#开头的指令）在程序性能方面是没有代价的，因为它们在程序编译之前就被解析了。

#if指令对于使用相同源代码支持多个平台或多个配置非常有用。#if比if更有效，因为#if在编译时解析
而如果是在运行时解决的。

当用于定义常量时，#definedirectives等同于constdefinitions。例如，#define abc123和const int ABC=123；因为在大多数情况下，优化编译器可以用其价值然而，在某些情况下，常内部声明可能会占用内存空间
其中，#定义直接变量占用内存空间。一个浮动点总是占用记忆空间，即使没有相同的时间。

#定义指令在用作宏时，有时比函数更有效。请参见第页48fora讨论。

7.36 Namespaces

使用名称空间没有关于执行速度的相关术语。

8.在编译器中的优化

8.1编译器如何操作

现代的编译器可以对代码进行大量的修改，以提高性能。让程序员知道编译器能做什么和它不能做什么是很有用的。下面的部分描述了与之相关的一些编译器优化
程序员知道。

函数内衬

编译器可以用被调用函数的主体替换函数调用。样例

```
// Example8.1a
浮动方块（浮动a){
    返回a*a}

浮点抛物线（浮点x）
    返回平方（x）+1.0 f；}
```

编译器可以用Square内部的代码替换对Square的调用：

```
//示例8.1 b
浮点抛物线(float x){return
    x*x+1.0f;}
```

functioninlining的优点是：

- •消除了调用、返回和参数传递的开销。
- •代码缓存会更好，因为代码变得连续。
- ·如果只有一次对inlinedfunction的调用，则代码会变小。
- ·函数内联可以打开其他优化的可能性，如下所述。

函数内联的缺点是，如果对内联函数的调用不止一次，并且函数是大的，代码就会变大。如果函数很小，或者只从一个或几个地方调用，编译器更有可能内联函数。

恒定折叠和恒定传播

仅包含常量的表达式或子表达式将被calculatedresult替换。示例：

```
//示例8.2 a
双a，b;
a= b+2.0/3.0;
```

编译器将把它替换为

```
//示例8.2b
a= b+0.66666666666666666667;
```

这实际上很方便。write2.0/3.0than更容易计算值，并写入许多小数。建议在这样的a周围加一个括号
子表达式，以确保编译器将其识别为一个子表达式。Forexample，b*2.0/3.0将被计算为（b*2.0）/3.0rather而不是b*（2.0/3.0），除非你在常量表达式周围加上一个括号。

一个常量可以通过一系列的计算来传播：

```
// Example8.3a
浮动抛物线（浮动x){返回
    x*x+1.0f; }

浮点a、b;
=抛物线（2.0f）;
b= a+1.0f;
```

编译器可以用

```
//示例8.3b
```

```
a=5.0f;
b=6.0f;
```

如果表达式包含不能内联或不能在压缩时计算的混合物，则不可能进行恒定折叠和恒定传播。例如：

```
// Example8.4
双=sin（0.8）；
```

正弦函数是在一个单独的函数库中定义的，您不能期望
编译器能够排列这个函数并在编译时进行计算。一些编译器能够在编译时计算出最常见的数学函数，如sqrt和pow，但不能计算出更复杂的函数，如sin。

消除指针

如果目标点已知，则可以消除指针或引用。样例

```
// Example8.5a
voidPlus2(int*p){
    *p=*p+2;}

inta;
Plus2(&a);
```

编译器可以用

```
//示例8.5b
a+=2;
```

常见的子表达消除

如果相同的子表达式发生了不止一次，那么编译器可以只计算一次。样例

```
// Example8.6a
inta,b,c;
b=(a+1)*(a+1);
c=(a+1)/4;
```

编译器可以用

```
//示例8.6b
inta、b、c、温度;
temp=a+1;
b=温度*温度;
c= temp/4;
```

寄存器变量

最常用的变量存储在寄存器中(见页面 27).

整数寄存器变量的最大数量在32位系统中约为6个，在64位系统中约为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，在64位模式下启用AVX512指令集时为32个。有些编译器很难在32位系统中创建浮点寄存器变量
除非启用了SSE（或更高版本）指令集。

编译器将选择最常用于寄存器变量的变量。典型的
寄存器变量的候选变量是局部变量、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占用，即如果有一个指向它的指针或引用，它就不能存储在寄存器中。因此，您应该避免对可能受益于寄存器存储的变量进行任何指针或引用。

Liverange分析

变量的范围是该变量使用的可待因的范围。优化编译器可以为多个变量使用同一个寄存器，如果它们的有效范围不使用同一个寄存器
重叠或者它们是否确定具有相同的值。当可用寄存器的数量有限时，这很有用。示例
：

```
//示例8.7
int some function（int a，int x[]）
   {int b，c；
   x[0]=A；
   b=a+1；
   x[1]=b；
   c=b+1；
   返回c；}
```

在这个例子中，A、b和C可以共享同一个寄存器，因为它们的有效范围不重叠。如果c=b+1更改为c=a+2，则A和b不能使用sameregister，因为它们的活动范围现在重叠。

编译器通常不对存储在内存中的对象使用此原则。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。见
页图93是如何使不同对象共享同一内存区域的示例。

连接相同分支hes

通过连接相同的代码片段，可以使代码更加紧凑。示例：

```
//示例8.8 a
双x，y，z；布尔b；
如果(b){
   y=sin(x)；
   z=y+1.；}
否则{
   y=cos(x)；
   z=y+1.；}
```

编译器可以用

```
//示例8.8 B
双x，y；布尔b；
如果(b){
   y=sin(x)；}
否则{
   y=cos(x)；}
z=y+1.；
```

消除跳跃

跳转可以通过复制跳转到的代码来避免。示例：

```
//示例8.9 a
int SomeFunction（int a，bool b）{if
    （b）{
       a=a*2；}
    否则{
       a=a*3；}
    返回a+1；}
```

这段代码从a=a*2跳转；ret urna+1；。编译器可以通过复制returnstatement来消除这种跳跃：

```
//示例8.9 B
int SomeFunction（int a，bool b）{if
    （b）{
       a=a*2;
       返回a+1；}
    否则{
       a=a*3;
       返回a+1；}
}
```

如果条件可以简化为AlwaysTrue或alwaysfalse，则可以消除Abranch：

```
//示例8.10 a
if(true){
   a=b;
}
否则{
   A=c;
}
```

可以归结为：

```
//示例8.10 B
a=b;
```

如果从前一个分支已知该条件，也可以消除A分支。示例：

```
//示例8.11 a
int SomeFunction（int a，bool b）{if
    （b）{
       a=a*2；}
    否则{
       a=a*3；}
    如果(b){
       返回a+1；}
    否则{
       返回a-1；}
}
```

编译器可以将其简化为：

```
//示例8.11 b
```

```
int SomeFunction（int a，bool b）{if
    （b）{
       a=a*2;
       返回a+1；}
    否则{
       a=a*3;
       返回a-1；}
}
```

循环展开

如果需要高度优化，编译器将展开循环。参见第页45.如果循环体非常小，或者如果它打开了进一步优化的可能性，这可能是有利的。重复计数非常低的循环可以完全取消，以避免循环开销。示例：

```
//示例8.12 a
int i，A[2];
对于（i=0；i<2；i++）a[i]=i+1;
```

编译器可以将其简化为：

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

不幸的是，有些编译器滚动太多。过度循环展开不是最佳的
因为它增加了代码缓存、微操作缓存和循环缓冲区的负载，这些都是关键资源。

循环不变码字离子

如果计算独立于循环计数器，则可以将其移出循环。示例：

```
//示例8.13 a
int i，A[100]，b;
对于（i=0；i<100；i++）{
   a[i]=b*b+1;
}
```

编译器可能会将其更改为：

```
//示例8.13 B
int i，a[100]，b，温度;
温度=b*b+1;
对于（i=0；i<100；i++）{a[i]=温
   度;
}
```

矢量化

编译器可以使用向量寄存器同时处理多个数据，例如：

```
//示例8.14。自动矢量化
const int size=1024;
浮点A【大小】、b【大小】、c【大小】;
//...
for（int i=0；i<size；i++）
   {a[i]=b[i]+c[i];
}
```

SSE2指令集提供128位向量寄存器。一个128位寄存器c保存四个32位浮点值。编译器可以在示例中展开循环使用128位向量寄存器，在一次操作中进行8.14乘以4和dofour加法。

如果AVX2指令集可用，则可以一次进行八次加法，
使用256位向量寄存器。如果AVX512指令集可用，则可以使用512位向量寄存器一次进行16次加法。

有必要向编译器指定要使用哪个指令集。微处理器支持的最高指令集可获得最佳性能。

第页进一步解释了自动矢量化118.

感应变量

一个表达式是一个计数器的线性函数，可以通过将一个常数加到前一个值来计算。示例：

```
//示例8.15 a
int i，A[100]；
对于（i=0；i<100；i++）{
    A[i]=i*9+3；
}
```

编译器可以通过将其更改为：

```
//示例8.15 B
int i，A[100]，温度；
温度=3；
对于（i=0；i<100；i++）{a[i]=温
    度；
    温度+=9；}
```

归纳变量通常用于计算数组元素的地址。示例：

```
//示例8.16 a
结构S1{双A；双b；}；
S1列表[100]；int i；
对于（i=0；i<100；i++）{list[i]
    。a=1.0；
    列表[i]。b=2.0；}
```

为了访问列表中的元素，编译器必须计算它的地址。The
列表[i]的地址等于列表p lusi*sizeof（S1）的开头的地址。这是i的线性函数，可以用一个归纳变量来计算。编译器可以使用相同的归纳变量来访问List[i].a和List[i].b。当最终值
感应变量可以提前计算。这将代码简化为：

```
//示例8.16 B
结构S1{双A；双b；}；
S1列表[100]；
S1*温度；
for(temp=&list[0];temp<&list[100];temp++){temp-
    >a=1.0；
    温度->b=2.0；}
```

factorsizeof(S1)=16实际上隐藏在C++语法融合蛋白示例背后8.16 b.&list[100]的整数表示为(int)(&list[100])=

（int）（&list[0]）+100*16，andtemp++实际上是将16加到temp的整数值上。

编译器不需要归纳变量来计算数组的地址
简单类型的元素，因为CPU硬件支持计算
数组元素的地址，如果该地址可以表示为基地址加上
常数加上指数乘以1、2、4或8的因子，但不是任何其他因子。如果一个and bin示例8.16 a是float而不是double，那么（S1）的大小将是8，并且不需要感应变量，因为CPU有硬件支持将索引乘以8。

编译器很少为浮点表达式或更多的表达式创建归纳变量
complexinteger表达式，因为这会生成循环携带的依赖链，可能会限制整体性能。参见第页83关于如何使用浮点归纳变量来计算多项式的示例。

调度

为了并行执行，编译器可以重新排序指令。示例：

```
//示例8.17
浮点A、b、c、d、e、f、x、y;
x=A+b+c;
y=d+e+f;
```

在这个例子中，编译器可以交错两个公式，这样首先计算a+b，然后计算end+e，然后将cis加到第一个和，然后将fis加到第二个和，然后将第一个结果存储在x中，最后将第二个结果存储在y中。这样做的目的是帮助CPU并行执行多个计算。现代CPU实际上能够重新排序没有编译器帮助的指令（参见第113），但是编译器可以使CPU更容易地重新排序。

代数约简

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

我不认为程序员经常编写类似-（-a）的表达式，但是这样的表达式可能是其他优化的结果，比如函数内联。
可约化表达式也经常作为宏扩展的结果出现。

然而，程序员确实经常编写可以推导的表达式。可以是
因为非简化表达式更好地解释了程序背后的逻辑或
因为程序员没有考虑过代数还原的可能性。例如，程序员可能更喜欢写if（！a&&！b）而不是等价的if（！（ab）），即使后者有一个无运算符。幸运的是，在这种情况下，所有编译器都能够进行缩减。

你不能指望编译器减少复杂的代数表达式。例如，
并非所有编译器都能够将（a*b*c）+（c*b*a）简化为a*b*c*2。在编译器中实现代数的许多规则是相当困难的。有些编译器可以减少某些类型的表达式，其他编译器可以减少其他类型的表达式，但我从未见过任何编译器可以减少所有类型的表达式。在布尔代数的c ase中，有可能
实现一个通用算法（例如Quine-McCluskey或Respresso）可以减少任何表达式，但我测试过的编译器似乎都没有这样做。

编译器更擅长简化整数表达式而不是浮点表达式，尽管代数规则在这两种情况下都是一样的。这是因为代数
对浮点表达式的操作可能会产生不良影响。这种效果可以通过以下示例来说明：

```
//示例8.18
int a、b、c、y;
y=a+b+c;
```

根据代数规则，我们可以写出：

```
y=c+b+A;
```

如果子表达式c+b可以在其他地方重用，这可能是有用的。现在考虑
ais放大负数，b和c放大正数的情况，因此c+的计算溢出。整数溢出将使值被环绕并给出负结果。幸运的是，c+bis的溢出用后续的
添加ais时出现下溢。A+b+c的结果将是相同的asc+b+A，即使
后一种表达式涉及溢出和下溢，而前一种表达式不涉及。这就是为什么对整数表达式使用代数操作是安全的（<、<=、>和>=运算符除外）。

相同的参数不适用于浮点表达式。浮点变量不环绕溢出和下溢。浮点变量的范围如此之大，以至于我们不必太担心溢出和下溢，除非在特殊数学中-
数学应用。但我们确实不得不担心精度的损失。让我们用浮点数重复上面的例子：

```
//示例8.19
浮点a=-1.0E8，b=1.0E8，c=1.23456，y;
y=a+b+c;
```

这里的计算结果是a+b=0，然后0+1.23456=1.23456。但是如果我们改变操作数的顺序并添加b和cfirst.b+c=
100000001.23456.float类型的精度大约为7个有效数字，因此b+c的值将四舍五入到100000000。当我们把这个数字加起来时，我们得到0而不是1.23456。

这个论点的结论是浮点操作数的顺序不能是
在没有失去精度风险的情况下进行更改。编译器不会这样做，除非您指定了一个允许不太精确的浮点计算的选项。即使有放松的精确度，
编译器不会做像0*a=0这样明显的缩减，因为如果它是无穷大或nan（不是一个数），这将是无效的。不同的编译器行为不同，因为对于哪些不精确应该被允许，哪些不应该被允许有不同的看法。

您不能依赖编译器对浮点代码执行任何代数归约，除非您仔细指定了相关的浮点选项，以实现归约
允许。参见第页174 fora编译器选项列表。

手动进行代数还原更安全。我在不同的编译器上测试了减少各种代数表达式的能力。结果列于表见下文8.1。

去虚拟化

如果知道需要哪个版本的虚函数，优化编译器可以绕过虚函数调用的虚表查找。示例：

```
//Example8.20.去虚拟化
C0类{
    公众：
    虚空f()；
};

classC1：
   publicC0{public：
    虚空f()；
};

voidg(){C1
    obj1；
    C0*p=&obj1；
    p->f()；//虚拟调用C1：：f
}
```

如果没有优化，编译器需要在虚拟表中查找callp->f()是否转到C1：：f的C0：：。但是优化编译器会看到p总是指向classC1的对象，因此可以直接调用C1：：F而不使用virtualtable。

8.2不同编译器的比较

我在不同的C++编译器上做了一系列实验，看看它们是否能够进行不同类型的优化。所有的编译器都是多年来开发的，它们的能力也提高了几倍。你可能会好起来的
通过使用最新版本的编译器来提高性能。一些效率较低的编译器已经从市场上消失或不再维护。

我的测试结果总结在表中8.1.下表显示了不同的
编译器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。该表可以给出一些您可以进行哪些优化的指示
期望一个特定的编译器做什么，以及你必须手动做哪些优化。

必须强调的是，编译器在不同的测试示例上可能表现不同。你不能指望编译器总是按照表行事。

优化方法	Gnu	铿锵	微软	英特尔经典	英特尔基于LLVM
常规优化：					
函数内联	x	x	x	x	x
恒定折叠	x	x	x	x	x
恒定传播	x	x	x	x	x
通过环路的恒定传播	x	x	x	x	x
指针消除	x	x	x	x	x
常见亚表达消除	x	x	x	x	x
寄存器变量	x	x	x	x	x
Liverange分析	x	x	x	x	x
连接相同分支	x	x	x	x	x
消除跳跃	x	x	x	x	x
尾部呼叫	x	x	x	x	x
删除始终为假的分支	x	x	x	x	x

循环展开，数组循环	x	过度的	x	x	过度的
循环展开，结构	x	x	x	x	x
循环不变码运动	x	x	x	x	x
数组元素的归纳变量	x	x	x	x	x
其他整数表达式的归纳变量	_	x	_	x	x
浮点表达式的归纳变量	_	_	_	_	_
多个累加器，整数	_	x	x	x	_
多个累加器，浮点	_	x	x	x	_
去虚拟化	x	x	_	_	x
轮廓引导优化	x	x	x	x	x
整个程序优化	x	x	x	x	x
整数代数约简：					
a+b=b+a，a*b=b*a（可交换）	x	x	x	x	x
(a+b)+c=a+(b+c),(a*b)*c=a*(b*c)(关联)	_	x	x	仅限mul	x
A*b+A*c=A*(b+c)(分配)	x	x	x	x	x
A+b+c+d=(A+b)+(c+d)(改进并行性)	_	x	_	x	_
a*b*c*d=(a*b)*(c*d)(改进并行性)	_	x	_	x	x
x*x*x*x*x*x*x*x=((x2)2)2	x	x	_	x	x
A+A+A+A=A*4	x	x	x	x	x
a*x*x*x+b*x*x+c*x+d=((a*x+b)*x+c)*x+d	x	x	x	x	x
-(-A)=a	x	x	x	x	x
A-(-b)=A+b	x	x	x	x	x
A–A=0	x	x	x	x	x
A+0=a	x	x	x	x	x
A*0=0	x	x	x	x	x
A*1=a	x	x	x	x	x
(-A)*(-b)=A*b	x	x	x	x	x
a/a=1	x	x	x	_	x
A/1=a	x	x	x	x	x
0/a=0	x	x	x	_	x
乘以常数=移位和相加	x	x	x	x	x
Divideby常数=乘法和移位	x	x	x	x	x
除以2的幂=移位	x	x	x	x	x
(-a==-b)=(a==b)	x	x	x	_	x
(a+c==b+c)=(a==b)	x	x	x	x	x
！(a<b)=(a>=b)	x	x	x	x	x
(A<b&&b<c&&A<c)=(A<b&&b<c)	x	_	_	_	x
浮点代数还原：					
a+b=b+a，a*b=b*a（可交换）	x	x	x	x	x
A+b+c=A+(b+c)(关联)	x	x	_	x	x
a*b*c=a*(b*c)(关联)	x	x	_	_	x
A*b+A*c=A*(b+c)(分配)	x	x	x	x	x
a+b+c+d=(a+b)+(c+d)，a*b*c*d=(a*b)*(c*d)	x	x	x	_	x
a*x*x*x+b*x*x+c*x+d=((a*x+b)*x+c)*x+d	x	x	x	x	x
x*x*x*x*x*x*x*x=((x2)2)2	x	x	_	_	x
A+A+A+A=A*4	x	x	x	_	x
-(-A)=a	x	x	x	x	x
A-(-b)=A+b	x	x	x	x	x
A–A=0	x	x	x	x	x
A+0=a	x	x	x	x	x
A*0=0	x	x	x	x	x

A*1=a	x	x	x	x	x
(-A)*(-b)=A*b	x	x	x	x	x

a/a=1	x	x	_	_	x
A/1=a	x	x	x	x	x
0/a=0	x	x	x	_	x
(-a==-b)=(a==b)	x	x	x	_	x
(-a>-b)=(a<b)	x	x	x	x	x
除以常数=乘以倒数	x	x	x	x	x
布尔代数约简：					
没有分支的布尔运算	x	x	_	很少	x
A&&b=b&&A，a b=b A（可交换）	x	x	_	x	x
A&&b&&c=A&&(b&&c)（关联）	_	_	_	x	_
(A&&b)(A&&c)=A&&(bc)(分配)	x	x	_	_	x
(ab)&&(ac)=A(b&&c)(分配)	x	x	_	_	x
！一个&&！b=！（ab）（德摩根）	x	x	_	_	x
！(！a)=a	x	x	x	x	x
一个&&！A=假，A！a=tr ue	x	x	x	x	x
a&&true=a，a fals e=a	x	x	x	x	x
a&&false=false，a true=t rue	x	x	x	x	x
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	x
（A&&b）（！A&&c）（b&&c）=A？b：c	_	_	x	x	_
(A&&b)(A&&b&&c)=A&&b	x	x	x	x	x
(A&&！b)(！A&&b)=A异或b	x	x	_	_	x
按位运算符integerreductions：					
A&b=b&A，a b=b A（可交换）	x	x	x	x	x
A&b&c=A&(b&c)（关联）	x	x	x	x	x
(A&b)(A&c)=A&(bc)(分配)	x	x	x	x	x
(ab)&(ac)=A(b&c)(分配)	x	x	x	x	x
~A&~b=~(ab)(德摩根)	x	x	x	x	x
~(~a)=a	x	x	x	x	x
a&~a=0，a~a=-1	x	x	x	x	x
a&-1=a，a0=a	x	x	x	x	x
A&0=0	x	x	x	x	x
a-1=-1	x	x	x	x	x
A&A=A，a a=a	x	x	x	x	x
(A&b)(A&~b)=a	x	x	x	x	x
（A&b）（~A&c）（b&c）=（A&b）（~A&c）=A？b：c	-	-	-	-	-
(A&b)(A&b&c)=A&b	x	x	x	x	x
（A和 (~a&b)=a^b ！b）	x	x	x	x	x
~a^~b=a^b	x	x	x	x	x
A&b&c&d=(A&b)&(c&d)	_	x	_	x	x
A<<b<<c=A<<(b+c)	x	x	x	x	x
按位运算符整数向量约简：					
A&b=b&A，a b=b A（可交换）	x	x	_	_	x
A&b&c=A&(b&c)（关联）	x	x	_	_	x
(A&b)(A&c)=A&(bc)(分配)	x	x	_	_	x
(ab)&(ac)=A(b&c)(分配)	x	x	_	_	x
~A&~b=~(ab)(德摩根)	x	x	_	_	x
~(~a)=a	x	x	_	_	x

a&~a=0，a~a=-1	x	x	_	_	x
A&-1=a	x	x	_	_	x
a0=a	x	x	_	x	x
A&0=0	x	x	_	x	x

a-1=-1	x	x	_	_	x
A&A=A，a a=a	x	x	_	x	x
(A&b)(A&~b)=a	x	x	_	_	x
（A&b）（~A&c）（b&c）=（A&b）（~A&c）=A？b：c	-	-	-	-	-
(A&b)(A&b&c)=A&b	x	x	_	_	x
（A和 (~a&b)=a^b ！b）	x	x	-	-	x
~a^~b=a^b	x	x	_	_	x
A&b&c&d=(A&b)&(c&d)	x	x	_	_	x
三元算子构造	_	_	x	_	_
A<<b<<c=A<<(b+c)	_	_	_	_	x
整数向量代数约简：					
a+b=b+a，a*b=b*a（可交换）	x	x	x	_	x
(a+b)+c=a+(b+c),(a*b)*c=a*(b*c)(关联)	x	x	_	_	x
A*b+A*c=A*(b+c)(分配)	x	x	_	_	x
A+b+c+d=(A+b)+(c+d)	x	x	_	_	x
x*x*x*x*x*x*x*x*x=((x2)2)2	x	x	_	_	x
A+A+A+A=A*4	x	x	_	_	x
a*x*x*x+b*x*x+c*x+d=((a*x+b)*x+c)*x+d	x	x	_	_	x
-(-A)=a	x	x	_	_	x
A-(-b)=A+b	x	x	_	_	x
A–A=0	x	x	_	x	x
A+0=a	x	x	x	_	x
A*0=0	x	x	x	x	x
A*1=a	x	x	_	_	x
(-A)*(-b)=A*b	x	x	_	_	x
乘以2的幂=移位	x	x	_	_	x
(-a==-b)=(a==b)	_	x	_	_	x
(a+c==b+c)=(a==b)	_	x	_	_	x
！(a<b)=(a>=b)	x	x	_	_	_
(A<b&&b<c&&A<c)=(A<b&&b<c)	_	_	_	_	_
浮点向量代数约简：					
a+b=b+a，a*b=b*a（可交换）	x	x	x	x	x
(a+b)+c=a+(b+c),(a*b)*c=a*(b*c)(关联)	x	x	_	_	_
A*b+A*c=A*(b+c)(分配)	x	x	_	_	_
A+b+c+d=(A+b)+(c+d)	x	x	_	_	_
x*x*x*x*x*x*x*x*x=((x2)2)2	x	x	_	_	_
A+A+A+A=4*a	x	x	_	_	_
a*x*x*x+b*x*x+c*x+d=((a*x+b)*x+c)*x+d	x	x	_	x	_
-(-A)=a	x	x	_	_	x
A-(-b)=A+b	_	x	_	_	x
A–A=0	x	x	_	x	_
A+0=a	x	x	x	x	_
A*0=0	x	x	x	x	_
A*1=a	x	x	_	x	x
(-A)*(-b)=A*b	_	x	_	_	x
a/a=1	x	x	_	_	_
A/1=a	x	x	_	x	x
0/a=0	x	x	x	x	_
除以常数=乘以倒数	x	x	_	_	_
(-a==-b)=(a==b)	_	_	_	_	_
！(a<b)=(a>=b)	_	_	_	_	_

一般向量优化：					
循环的自动矢量化	x	x	x	x	x

带分支的自动矢量化	x	x	x	x	x
使用g掩码的条件指令	x	x	-	x	x
将条件零合并到屏蔽指令中	x	x	_	x	x
将广播合并到指令中	x	x	_	_	x
合并blendinto掩码d指令	x	x	_	x	x
将布尔和合并到屏蔽比较中	x	x	_	x	x
eliminatemask all true	x	x	x	x	x
eliminatemask all false	x	x	_	x	x
数学库函数的矢量化	x	x	x	x	x
优化跨载体pe突变	_	x	_	_	x

表8.1。不同C++编译器中优化的比较

这些测试是通过编译64位Linux或Windows的测试代码来执行的，所有相关的优化选项都打开了，包括放松的浮点精度。测试了以下编译器版本：

Gnu C++版本12.1.0Clang
C++版本12.0.0

Microsoft C++编译器Visual Studio版本17.2.5,2022英特尔
C++编译器经典版本2021.6.0
基于oneAPILLVM的英特尔C++编译器版本2022.1.0

Clang和Gnu编译器是优化最好的编译器，而微软编译器的性能较差。Microsoft编译器通常使用比
其他编译器也在做同样的事情。微软编译器对于要求不高的目的来说仍然足够好。

英特尔编译器有两个版本，一个是名为“Classic”的旧版本，另一个是使用开源LLVMproject的Clang编译器的新版本。基于VM的Intel编译器与Clang编译器非常相似，但却是Intel自己的
高度优化的函数库。不推荐使用经典版本，因为基于VM的英特尔编译器更好，而且经典版本生成的代码在AMD处理器上性能较差。

Microsoft Visual Studio提供了使用aClang或英特尔编译器作为插件的选项。

Clang编译器倾向于过多地展开循环。过度的循环展开不是最佳的，因为它会增加代码缓存、微操作缓存和循环缓冲区的负载，而这些都是关键资源。除此之外，Clangis是一个优秀的编译器，有时会做一些巧妙的事情来优化代码。您可以优化大小(-Os)以限制循环
展开。

8.3编译器优化的障碍

有几个因素会阻止编译器进行我们想要的优化。对于程序员来说，意识到这些障碍并知道如何避开它们是很重要的。下面讨论优化的一些重要障碍。

无法跨模块优化

除了它正在编译的模块之外，编译器没有关于其他模块中函数的信息。这阻止了它跨函数调用进行优化。示例：

```
//示例8.21
模块1。cpp
int Func1(int
    x){return x*x+1
    ;
}

模块2。cpp
```

```
int Func2(){
    int a=Func1（2）;
    ...
}
```

如果Func1和func2在同一个模块中，那么编译器将能够执行functioninlining和constantpropagation，并减少到constant5。但是编译器在编译module2.cpp时没有关于utfunc1的必要信息。

解决这个问题最简单的方法是将多个。cpp模块通过#includedirectives合并为一个。这肯定适用于所有编译器。大多数编译器都有一个名为整个程序优化的功能，它可以实现跨模块的优化（参见85).

指针别名

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

```
//示例8.22 a
voidFunc1(inta[], int p[]){inti
    对于（i=0; i<100; i++）
        {a[i]=p[0]+2;
    }
}

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

这里，有必要重新加载p[0]并计算p[0]+2a一百次，因为
指向toby p[0]的值与ina[]中的一个元素相同，该元素将在循环过程中发生变化。不允许假设p[0]+2是一个循环不变码，它可以是
移出了循环。示例8.22 a确实是一个非常人为的例子，但关键是编译器不能排除这种人为例子的理论可能性
存在。因此，编译器不能假设p[0]+2是一个可以移出循环的循环不变表达式。

如果编译器支持，可以通过使用关键字restrictor restrict来告诉编译器特定指针不别名任何内容：

```
//示例8.22 B
void Func1(int*限制A, int*限制p){inti;
    对于（i=0; i<100; i++）
        {a[i]=p[0]+2;
    }
}
```

一些编译器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。这要求您仔细分析代码中的所有指针和引用，以确保没有变量
或者在代码的同一部分以多种方式访问对象。

我们永远无法确定编译器是否接受了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去做。在示例中 8.22 B，你

可以计算p[0]+2并将其存储在循环外的临时变量中，如果您确定指针不会对数组中的任何元素进行别名。

```
//示例8.22 c
void Func1(int a[], int
   p[]){inti
   int p02=p[0]+2;
   对于(i=0; i<100; i++){A[i]=p02
      ;
   }
}
```

这种方法要求您能够预测优化的障碍在哪里。

指针别名也是自动矢量化的障碍。编译器不能
向量化循环，例如示例中的循环8.22 a如果不能排除指针混叠。

动态内存分配

任何动态分配的数组或对象（使用newor malloc）都必须
通过指针访问。对于程序员来说，指向不同动态分配的对象的指针不重叠或别名可能是显而易见的，但编译器通常会注意到这一点。它还可能阻止编译器最佳地对齐数据，或者阻止编译器
知道对象是对齐的。在需要它们的函数中声明对象和固定大小的数组更有效。本地声明还可以改善数据缓存，因为当函数返回并调用新函数时，分配给本地数组和对象的内存空间会被回收。

纯函数

纯函数是一个没有副作用的函数，其返回值只取决于其参数的值。这与函数的数学概念密切相关。

使用相同参数对纯函数的多次调用肯定会产生相同的
结果。编译器可以消除包含纯函数调用的公共子表达式，并且可以将包含纯函数调用的循环不变代码移出。不幸的是，如果函数定义在不同的模块或函数库中，编译器就无法知道函数是纯函数。

因此，当涉及到pure时，有必要手动进行公共子表达式消除、常数传播和循环不变代码运动等优化
函数调用。

一些编译器有一个属性，可以应用于函数原型，告诉编译器这是一个纯函数。示例：

```
//示例8.23
#ifdef GNUC
#define pure_function属性((const))
#否则
#definepure_function
#endif

double Func1(double)pure_function;

double Func2(double x){
   返回Func1(x)*Func1(x)+1.; }
```

在这里，Gnu编译器只会调用Func1一次，而其他编译器会调用两次。

编译器通常知道sqrt、pow和log等标准库函数是纯函数，但不幸的是，没有标准化的方法来告诉编译器用户-
定义的函数是纯的。

虚函数和函数指针

编译器并不总是能够确定
虚拟函数将被调用，或者一个函数指针指向什么。因此，它不能在线虚拟函数或优化跨函数，确切的类不是
大家知道的

代数约简

大多数编译器都可以做简单的代数约简，如-（-a）=a，但它们绝不也不能做更复杂的约简。代数约简是一个在辅助计算器中难以实现的复杂过程。

许多代数约简是不允许的，因为数学纯度和推理
准确率在许多情况下，可能构造模糊的例子，其中简化会导致溢出或失去精度，特别是在浮点表达式(见页75).编译器不能排除特定还原在特定情况下无效的可能性。因此，必须在许多情况下按标准地进行代数约简。

由于页面上解释的原因，整数表达式不太容易出现溢出和精度损失的问题75.因此，编译器可以做更多的事情
整数表达式比浮动点表达式减少。大部分减少
对合整数的加、减和乘法在所有情况下都是允许的，而任何简化涉及除法和关系算子(例如。'>')是不允许的，作为数学纯度的原因。例如，编译器可能无法减少
整数表达式->-b到<b，因为溢出的可能性。

各种编译器选项可用于告诉编译器它可能使用哪种类型的代数缩减。解释了这些选项
下面的页面86.

表8.1(页80)显示了编译器能够做的哪些减少，至少在某些情况下，并且启用了相关的选项。编译器无法做的所有缩减
必须由程序员手动完成。

浮动点感应变量

编译器很少为浮动点表达式使用浮动点转换变量
因为循环携带的依赖链可能会降低性能，而且因为
累积的舍入误差可能会降低精度。然而，如果循环计数器的函数可以更有效地计算出来，浮动点感应变量可能有用
从以前的值比从循环计数器。任何是循环计数器的第n个度项式的表达式都可以通过加法和函数来计算。下面的例子显示了第二个r德多项式的原理：

```
// Example8.24a.循环制作多项式表
常量双A=1.5，B=2.25，C=3.5；//多项式系数
双表[100]；//表
intx；//循环计数器
对于（x=0；x<100；x++）{
    表[x]=A*x*x+B*x+C；//计算多项式
}
```

该多项式的计算可以通过使用两个归纳变量进行两次加法：

```
//示例8.24 b.用归纳变量计算多项式
常量双A=1.5，B=2.25，C=3.5；//多项式系数
双桌[100]；                                    //表
intx;                                          //循环计数器
常量双A2=A+A；双Y=C；                          //=2*A
双Z=A+B；                                      //=a*x*x+B*x+C//=Del
对于（x=0；x<100；x++）                        ta Y
    {Table[x]=Y；
    Y+=Z；                                     //存储结果
    Z+=A2；}                                   //更新感应变量//更新感应变量 和
                                               Z
```

请注意，如果归纳变量具有无法准确表达的值，则累积舍入误差会降低精度。

示例中的循环8.24 BHAS有两个循环携带的依赖链，即两个
归纳变量Y和Z。每个依赖链都有一个延迟，它与浮点加法的延迟相同。这可能小到足以证明方法的合理性
循环还包含其他耗时的操作。lon gerloop携带的依赖项
链会使归纳变量方法变得不利，除非该值是从两个或两个以上的值计算出来的。

归纳变量的方法也可以是矢量化的，如果你考虑到每个值都是从r放回序列中的值计算出来的，其中r是
向量或循环展开因子中的元素数。在每种情况下找到正确的公式都需要一点数学运算。

内联函数具有非内联副本

函数内联有一个复杂的问题，即同一个函数可以从另一个函数调用
模块。编译器必须制作内联函数的非内联副本，以便该函数也可以从另一个模块调用。如果没有其他模块调用该函数，则此非内联副本为死代码。代码的这种碎片化使得
缓存效率较低。

有各种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。这告诉编译器不能从任何其他模块调用该函数。静态声明使它
编译器更容易评估内联函数是否是最佳的，并且它防止编译器制作内联函数的未使用的副本。static关键字还使得各种其他优化成为可能，因为编译器不必服从
无法从其他模块访问的函数的任何特定调用约定。您可以将statickeyword添加到所有本地非成员函数中。

不幸的是，此方法不适用于类成员函数，因为static关键字对于成员函数具有不同的含义。您可以通过在类定义中声明functionbody来强制成员函数内联。这将防止
编译器不会制作函数的非内联副本，但它的缺点是，即使这样做不是最佳的（即，如果成员函数很大并且从任何不同的地方调用），函数也总是内联的。

大多数编译器都有函数级链接选项（Windows：/Gy，Linux：
-ffunction-sections），它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。长的
代码中的依赖链防止CPU执行无序执行，如
在第页解释21.避免长依赖链，尤其是具有长延迟的循环承载依赖链。

8.5编译器优化选项

所有C++编译器都有各种可以打开和关闭的优化选项。研究您正在使用的编译器的可用选项并打开所有相关选项是很重要的。最重要的选项解释如下。

许多优化选项与调试不兼容。调试器可以执行
一次一行代码并显示所有变量的值。显然，当代码的一部分被重新排序、内联或超时时，这是不可能的。这是常见的
使程序可执行的两个版本：一个是在程序开发期间使用的具有完全调试支持的调试版本，另一个是具有所有相关功能的发布版本
优化选项已打开。MostIDE（集成开发环境）有
用于制作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分析支持。

大多数编译器提供了在优化大小和优化速度之间的选择。
当代码无论如何都很快并且您希望可执行文件尽可能小时，优化大小是相关的。在某些情况下，优化时可能会获得更好的性能
如果CPU中的代码缓存、微操作缓存或循环缓冲区是一个有限的资源，则大小比速度更重要。

当CPU访问和内存访问是关键时间时，优化速度是相关的
消费者。最高优化选项通常是最好的，但在某些情况下，最高优化选项实际上是在做特定情况下不是最佳的事情。

一些编译器提供配置文件引导的优化。这是通过以下方式工作的。首先，在分析支持下编译程序。然后用profiler进行测试
确定程序流以及每个函数和分支的次数
处决。然后，编译器可以使用这些信息来针对热点进行优化，并将不同的分支和函数按最佳顺序排列。

大多数编译器都支持整个程序优化。这是通过编译两个步骤来完成的。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
编译完成。寄存器分配和函数inliningis在第二个
步。中间文件格式不标准化。它甚至不兼容同一编译器的不同版本。因此，不可能在
这种格式。

其他编译器提供了编译多个的可能性。cpp文件到一个单一的leobject文件。这使得编译器能够在过程间进行跨模块优化
优化已启用。一种更原始但非常有效的完成整个程序的方法
优化是通过#includedirectives将所有源文件连接到一个源文件中，并声明所有非成员函数为静态或内联函数。这将使编译器能够对整个程序进行过程间优化。

当不需要与旧cpu兼容时，您可以选择新可用的指令集。或者，您也可以为代码中最关键的部分创建多个版本，以支持不同的cpu。该方法在第页上有解释135.

当没有例外的处理时，代码将变得更有效。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，并且您希望代码能够从异常中恢复。请参见第页62.

建议取消对运行时类型标识（RTTI）的支持。请参见页面55.

建议启用快速浮点计算或关闭严格浮点计算的要求，除非严格要求。请参见第页75讨论。

如果可用，请打开“功能级别链接”选项。请参见第页84来解释这个选项。

如果您确定代码没有指针别名，请使用“假设没有指针别名”选项。参见第页81寻求解释。

许多编译器都有“标准堆栈帧”或“帧指针”的选项。standardstack框架用于调试和异常处理。省略标准堆栈
框架使函数调用更快，并使一个额外的寄存器可用于其他目的。这是有利的，因为注册是一种稀缺资源。不要使用堆叠框架
除非您的程序依赖于异常处理。

参见第页174 fora编译器选项的详细列表。

优化浮点代码的选项

优化浮点代码的一些障碍可以通过使用特定的编译器选项来克服。

通过将编译器optimizationlevel设置为2(-O2)或更高，可以实现整数表达式的大多数优化。浮点表达式的优化更为复杂
因为许多优化可能会损害精度或标准合规性。The
程序员需要考虑几个编译器选项来调整浮点代码的优化。以下讨论适用于基于Gnu、Clang和LLVM的英特尔
编译器。

选项-ffp-model=fast或-funsafe-math-optimizations将启用
需要操作数重新排序的优化。计算a+b+c和c+b+a不一定会给出相同的结果，如示例中所述8.19.宽容的
浮点模型对于涉及操作数重新排序或
否则给出稍微不同的精度。这些选项还能够将乘法和加法组合到已使用的乘加指令中。

当变量a可以是无穷大或无穷大时，诸如asa*0=0和a-a=0的优化是无效的。选项-ffinite-math-only告诉编译器允许这样的优化，并假设数字是有限的。

-fno-trapping-math。此选项告诉编译器不要在捕获时执行任何操作
具有try/catchblocks或全局浮点状态的浮点异常
登记。此选项对于包含分支的循环的矢量化是必需的。The
没有此选项，编译器无法在向量代码中创建分支，因为未采取的分支中的异常可能会引发陷阱。

-fno-math-errno.sqrt和log等数学函数在出错时设置一个名为errno的变量。这种检测错误的方法效率低下，应该
避免了。选项-fno-math-errno对于包含数学函数的循环的矢量化是必需的。下一节将讨论数学函数的进一步考虑。

Microsoft编译器的选项较少。通过选项/fp：fast可以获得浮点代码的最大优化。不要启用浮点异常。

参见第页174 fora编译器选项的详细列表。

参见第页131用于向量化数学函数的特征。

8.6优化指令

一些汇编器有大量的关键字和指令，用于提供具体的
优化代码中特定位置的指令。这些指令中有许多都是如此
特定于编译器。您不能期望Windows编译器的指令在Linux编译器上工作，反之亦然。但大多数微软指令工作于其他编译器
而Gnu指令则可以在Linux的其他编译器上工作。

可以在所有C-++编译器上工作的关键字

关键字挥发物确保一个变量永远不会存储在寄存器中，甚至不是暂时的。这将用于在多个线程之间共享的变量，但它也可以用于为测试目的关闭一个变量的所有优化。

常数关键字告诉我们，一个变量从未改变过。这将允许该计算器在许多情况下优化去除变量。例如：

```
// Example8.25.整数常数
常量ArraySize=1000;
int列表[ArraySize];
...
为（int i=0；i<array大小；i++）列表[i]++;
```

这里，编译器可以用值1000替换ArraySize的所有出现。示例中的循环如果loopcount（ArraySize）的值是常量并且在编译时编译器已知，则8.25可以以更有效的方式实现。没有记忆会
为整数常量分配，除非它的地址（&ArraySize）被取走。

constpointer或constreference不能改变它指向的内容。常量成员
函数不能修改数据成员。建议使用const关键字
在适当的情况下，向编译器提供关于变量、指针或成员函数的附加信息，因为这可以提高优化的可能性。例如，编译器可以安全地假设AcClass数据成员的值在对属于同一类的constfunction的调用中是不变的。

静态关键字根据上下文有几种含义。关键字
静态，当应用到另一个成员函数时，意味着该函数不被任何其他模块访问。这使得内联更有效，并能够实现过程间操作
优化。请参见第页84.

关键字静态，当应用于全局变量时，它不会被任何其他模块访问。这将支持过程间的优化。

关键字静态，当应用于函数内的局部变量时，表示
变量将在函数返回时被保留，并在下次返回时保持不变
函数被调用。这可能效率低下，因为一些计算器将插入额外的代码，以防止变量防止从多个线程同时访问。即使变量是const，这也可能适用。

然而，可能有一个理由使局部变量静态，并确保它只在第一次调用函数时初始化。样例

```
// Example8.26
空白函数（）{
    静态常量双log2= log（2.0）；
    ...
}
```

在这里，log（2.0）只在第一次执行函数时进行计算。如果没有静态的，每次执行函数时都会重新计算该对数。这是不利的，函数必须检查是否之前被调用。这比再次计算的速度要快，但再计算的速度会更快
用计算出的值替换它。

关键字静态，当应用到一个类成员函数意味着它不能
访问任何非静态数据成员或成员函数。静态成员函数是
调用的速度是非静态成员函数快，因为它不需要一个“这个”指针。建议在适当的情况下调整其静态功能。

编译器特定的关键字

快速的函数调用。fastcallor attribute((fastcall)).“快速呼叫”
修改器可以使32位模式下使函数调用更快。前两个整数参数是转移寄存器，而不是在堆栈上。快速呼叫函数并不兼容
跨编译器。在64位模式下，其中的参数是不需要快速呼叫的
无论如何转移寄存器。类似的关键字是矢量调用。这将改进在64位Windows中使用浮点参数或向量参数的函数调用。请参见第页49.

纯函数。attrib软件（（const））（仅限Linux）。指定一个要纯的函数。这允许常见的子表达式消除和循环不变的协降级。请参见第页82.

假设有指针别名。限流器（名称）或实用优化（“a”，开启）。指定不发生指针别名。看页81的解释。请注意，这些指令并不总是有效的。

数据对齐方式。declspec (align(16))or attribute((aligned (16))).指定阵列和结构的对齐方式。在C++11中被标识（16）所取代。对矢量操作很有用，请参见第页115.

8.7检查编译器的功能

研究编译器生成的代码以查看它的效果如何可能是非常有用的
优化代码。有时编译器会做一些非常巧妙的事情来使编码高效，有时它会以难以置信的愚蠢的方式编码事情。看着
编译器输出通常可以显示一些可以通过修改源代码来改进的东西，如下面的示例所示。

检查编译器生成的使用编译器选项的代码的最佳方法
汇编语言输出。在大多数编译器上，您可以通过调用编译器来实现这一点
从命令行与所有相关的优化选项和选项-S或/Fafor程序集输出。如果编译器没有程序集输出，则o参照
程序集输出使用了一种令人混淆的格式，然后在对象文件上使用反汇编程序。这个objconv反汇编程序在这里可能很有用。

Intel编译器可以在程序集输出中选择源代码注释（/FAs或-源代码-asm）。此选项使组件输出更具可读性，但不幸的是，它阻止了某些优化。如果要看到完全优化的结果，请不要使用源注释选项。

它完全可以在a的反汇编窗口中看到编译器生成的代码
调试器但是，您在调试器中看到的代码并不是优化版本
您在调试模式下进行编译。调试器可能无法完全设置断点
优化了发布模式代码，因为它没有行号信息。仍然有可能在调试器中单步通过优化的码器。
另一个解决方案是
插入一个固定的代码的中断点。这个
代码是asm int3；或asm（“int3”）；或删除gbreak（）；。如果你用的是 _
调试器中的优化代码（发布版本）将在中断处中断3
断点并显示反汇编，可能没有关于函数名和变量名的信息。记得再次删除Interrupt 3断点。

下面的示例显示了编译器的程序集输出是什么样子的，以及如何使用它来改进代码。

```
//示例8.27 a
void Func（int a[]，int&r）
    {inti
    对于（i=0；i<100；i++）
        {a[i]=r+i/2；
    }
}
```

英特尔编译器根据示例生成以下汇编代码8.27 a（32位模式）：

```
;示例8.27 a编译为汇编：
ALIGN4;对齐4
公众？Func@@YAXQAHAAH@Z；损坏的函数名
？Func@@YAXQAHAAH@Z PROC NEAR；功能启动
;参数1：8+esp；a
;参数2：12+esp；r
$B1$1：;未使用的标签
        推ebx；在堆栈上保存ebx
        mov ecx，DWORD PTR[ esp+8]；ecx= a
        xoreax，eax；eax= i=0
        mov edx，DWORD PTR[ esp+12]；edx= r
$B1$2：；环的顶部
        在ebx中计算i/2
        shr ebx，31；i的向下移动符号位
        添加ebx，eax；i+符号(i)
        sar ebx，1；右移=除以2
        添加ebx，DWORD PTR[ edx]；添加r所指向的内容
        mov DWORD PTR[ ecx+ eax*4]，ebx；以数组存储结果
        addeax,1;i++
        cmpeax，100；检查i<100
        如果为真，则重复循环
$B1$3：未使用的标签
        弹出ebx；从堆栈中恢复ebx
        ret;返回
        ALIGN4;对齐
？Func@@YAXQAHAAH@Z ENDP标记程序结束
```

编译器生成的大多数注释都被mycomments，ingreen所取代。习惯阅读和理解编译器生成的需要一些经验
汇编代码。让我详细解释一下上面的代码。有趣的名字
？Func@@YAXQAHAAH@zis func的名称，并添加了许多有关
函数类型及其参数。这叫做名字混乱。角色的？、@和
程序集名称中允许使用“$”。有关manglingare名称的详细信息，请参见

手册5：“不同C++编译器和操作系统的调用约定”。The
参数a和rare在堆栈上的地址sp+8和sp+12处传输并加载

分别转换为ECX和edx。（在64位模式下，参数将被传输到
寄存器，而不是在堆栈上)。ecxnow包含数组的第一个元素的寻址，而edx包含它所指向的变量的地址。一个参考是
与汇编代码中的指针相同。寄存器ebx在被使用之前会被推送到堆栈上
并在函数返回之前从堆栈中弹出。这是因为登记了我们的年龄
约定说，不允许一个函数更改ebx的值。只有寄存器eax、ecx和exx可以自由更改。循环计数器i被存储为一个寄存器
变量在eax。三循环初始化i=0；已被翻译成指令
xoreax，eax.这是一种常见的设置为零的方法，比moveax效率更高。循环车身从标签B1美元2：。这只是一个仲裁词
编译器已经选择了这个标签。它使用ebx asa临时寄存器进行计算
i/2+r.指令mov ebx、eax/shrebx、31将i的符号位复制到
ebx的最低有效位。接下来的两条指令addebx，eax/sarebx，1将this添加到i，并向右移动一位，以将i除以2。指令添加
ebx，DWORD PTR[edx]向ebx添加的不是edx，而是地址在edx中的变量。方括号表示将EDXA中的值用作内存指针。这是变量
r指向。现在ebx包含si/2+r。下一个指令mov DWORD PTR
[ecx+eax*4]，EBX将该结果存储在[i]中。注计算的效率如何
数组地址。ECX包含数组开头的地址。eaxholds
索引，i。为了计算元素号i的地址，这个索引必须乘以每个数组元素的大小（以字节为单位）。t中an的大小是4。所以地址
数组数秒+eax*4。结果的ebxis然后存储在地址上
[ecx+eax*4].这一切都不在一个单一的指令中。cpuu支持这种功能
关于快速访问数组元素的指令。指令addeax，1是循环
incrementi++.$2是i<100的循环条件。它将eax和100进行比较，如果i<100，则跳回$B1$2的标签。pop ebx恢复在开始时保存的ebx的值。从该函数中返回。

汇编列表显示了可以进一步优化的三件事。第一件事是
注意到它用i的号做一些有趣的事情，以便把i除以2。他没有注意到我从来不消极，这样我们就不必关心信号。我们可以通过使ian无符号或类型转换i来解决这个
无符号的int，在除以2之前(见第页150).

我们注意到的第二件事是r指向的值是从内存A中重新加载的
一百次。这是因为我们忘记告诉编译器假设没有指针别名（请参见81).添加编译器选项“假设没有指针别名”（如果可用）可能会改进代码。

第三个可以改进的是TR+i/2可以通过归纳法计算
变量，因为它是循环索引的阶梯函数。整数除法p revents
编译器frommaking aninduction变量，除非循环由2展开。（请参阅72).

结论是我们可以帮助编译器优化示例8.27 a通过将循环滚动两个并创建一个显式感应变量。（这消除了前两个建议的改进的需要）。

```
//示例8.27 B
void Func（int a[]，int&r）
    {inti
    int归纳=r;
    对于（i=0；i<100；i+=2）{A[i]=归纳；
        A[i+1]=诱导；
        感应++；
```

```
    }
}
```

编译器根据示例生成以下汇编代码8.27b:

```
样例8.27 b已编译成装配集：
ALIGN4；对齐4
公共的吗？@Z；混乱的函数名
?在你附近；Func的开始
；参数1：4+特别；a
; parameter2:8+ esp; r
$B1$1：未使用的标签
        mov eax，DWORD PTR[ esp+4]；eax=地址
        mov edx，DWORD PTR[ esp+8]；edx=地址
        mov ecx，DWORD PTR[ edx]；ecx=感应
        lea edx，DWORD PTR[ eax+400]；edx=指向a的结束
$B2$2：；环的顶部
        mov DWORD PTR[ eax]，ecx；一个[i]=感应；
        mov DWORD PTR[ eax+4]，ecx；a[i+1]=感应；
        添加ecx，1；诱导++；
        添加eax，8；指向[i+2]
        cmp edx，eax；与数组末尾比较
        是$B2$2；跳转到循环顶部
$B2$3：;未使用的标签
        ret;从函数返回
        ALIGN4;mark
_end;
? Func2@@YAXQAHAAH@Z ENDP
```

这个解决方案显然更好。循环体现在只包含六条指令，而不是
九，即使它在一次迭代中进行两次迭代。编译器已将i替换为包含currentarray元素地址的secondinduction变量(eax)。而是
而不是比较i和100在循环控制中，它将数组指针eaxto
数组末尾的地址，它已经提前计算并存储在edx中。
此外，该解决方案少使用一个寄存器，因此不必推送和弹出ebx。

9优化内存访问

9.1代码和数据的缓存

计算机中内存的代理。该代理机构的规模更小，也更接近于
CPU比主存更快，因此访问速度要快得多。为了尽可能最快地访问最常用的数据，可能有两级或三级缓存。

CPU的速度比内存的速度要快得多。因此，高效的缓存是非常重要的。

9.2缓存组织

如果您正在制作的程序具有非顺序访问的大数据结构，并且您希望防止欺骗，那么了解组织缓存是很有用的。如果您对更多的启发学指南感到满意，那么您可以跳过本部分。

大多数缓存都是有组织的链接和集合。让我用一个例子来解释一下。我的

例如是一个8 kb大小的缓存，行大小为64字节。每一行包含64个连续字节的内存。1千字节是1024字节，所以我们可以计算出行数是8*1024/64=128。这些行被组织为32组×4种方式。这意味着a

特定的内存地址不能加载到任意的粗线中。只能使用32个集合中的一个，但集合中的4行中的任何一行都可以使用。我们可以通过公式（设置）=计算特定的内存寻址：（设置）=(内存*地址）/（行大小）%（集合的数量）。这里，/表示不带截断的整数分割，%表示模数。例如，如果我们想从内存地址读取一个=10000，那么我们有（设置）=（10000/64）%32=28。这意味着a必须被读成四种方法中的一种*
缓存设置编号28中的行。如果我们使用十六进制，计算就会变得更容易
因为所有的数字都是2的幂。如果使用十六进制数字，我们有一个=0x2710和（设置）=（0x2710/0x40）%0x20=0x1C。读取或写入变量
地址0x2710将导致高速缓存从地址加载整个64或0x40字节
0x2700到0x273F从设置0x1C的四个粗线中的一个。如果程序后在这个范围内读取或写入任何其他地址，那么该值已经在缓存中，不必等待另一个内存访问。

假设程序从地址0x2710进行读取，并从地址进行横向读取
0x2F00、0x3700、0x3F00和0x4700。这些地址都属于设置号0x1C。
每个集合中只有四个高速缓存线。如果缓存总是使用最近使用最少的缓存行，那么当我们从0x4700读取时，覆盖地址范围从0x2700到0x273F的行将被丢弃。从地址0x2710再次读取将导致堵塞。但是，如果程序h从不同的地址读取，那么包含地址范围从0x2700到0x273F的线仍然是存在的。这个
问题是因为地址间隔为0x800。我将称这段距离为关键的一步。在记忆中的距离是很多的变量
关键的步伐将争夺相同的缓存线。该步幅可以计算为（临界步幅）=（集合数）×（行大小）=（总缓存大小）/（方法数）。

ifa程序包含许多分散在内存中的变量和对象，然后有一个风险，即几个变量恰好被多个临界步幅的间隔
并在数据缓存中引起争议。如果在程序内内存中分散了许多函数，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碰巧被关键步幅的倍数间隔，那么这可能会导致代码缓存中的争议。接下来的部分描述了各种各样的内容
避免这些问题的方法。

更多关于缓存工作的细节可以在维基百科的CPU缓存中找到(en。维基百科。org/wiki/L2_缓存)。

关于不同处理器的缓存组织的细节见手册3：“Intel、AMD和Viacpu的主题体系结构”。

9.3一起重用的函数应该一起存储

如果相互使用的函数也是，代码缓存工作效率最高
存储在代码存储器中彼此相邻。函数通常按照它们在源代码中出现的顺序存储。因此，将代码中最关键部分中使用的函数彼此靠近地收集在一起是一个好主意
文件。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。为了
例如，可以方便地将父类的成员函数放在一个
源文件和派生cl分配另一个源文件。如果父级的成员函数
类和派生类是从程序的同一关键部分调用的，这两个模块在程序内存中保持连续是有利的。这是可以做到的
通过控制这些模块连接在一起的顺序。链接的顺序通常是
模块在项目窗口或修改文件中出现的顺序。您可以通过请求amap文件来检查内存中功能的顺序。主题文件告诉

每个函数相对于程序开始时的地址。映射文件包括从静态库（.libor .a）链接的库函数的地址，但不是dyna mic
库（.dll或.so）。没有一种简单的方法来控制dyn友好链接库函数的地址。

9.4一起使用的变量应存储在一起

cachemiss非常昂贵。一个变量可以从缓存中获取一个几时钟周期，但是如果不是缓存周期，它可以从RAM内存中获取变量。

如果一起使用的数据片段存储在附近，缓存的工作效率最高
彼此在记忆中。变量和对象最好是在使用它们的函数中声明的。这些变量和对象将存储在堆栈上，堆栈很可能属于一级缓存中。本文介绍了变量存储的不同类型
页25.尽可能避免使用全局变量和静态变量，并避免使用动态记忆分配（新建和删除）。

面向对象编程可以是一种将数据保存在一起的有效方法。数据
类的成员（也称为属性）总是一起存储在该类的一个对象中。父类和派生类的数据成员一起存储在派生类的一个对象中52).

如果您有较大的数据结构，那么数据存储的顺序可能很重要。为了
例如，如果一个程序有两个数组，a和b，并且元素s在顺序[0]，b[0]，一个[1]，b[1]，...中被访问，那么你可以通过组织数据作为一个序列结构来提高性能：

```
// Example9.1a
int Func（int）
常量int大小=1024；
int a[大小]，b[大小]，i；
...
对于（i=0；i<大小；i++）{
   b[i]= Func（a[i]）；
}
```

如果组织如下，本例中的数据可以在记忆中按顺序访问：

```
//示例9.1b
int Func（int）
常量int大小=1024；
结构结构：
Sabab大小；
inti；
...
对于（i=0；i<大小；i++）{
   ab[i]。b=Func（ab[i].a）；
}
```

在示例中，在程序代码中会有额外的溢出量9.1b.相反，编码变得更简单，因为它只需要计算一个数组的元素地址。

有些编译器将对不同的数组使用不同的内存空间，即使它们不会同时使用。样例

```
// Example9.2a
void F1（int x[]）；
```

```
void F2(浮动x);

空白F3（bool y）{
    如果(y){
        int a[1000];
        F1(a);}
    else{
        浮动b[1000];
        F2(b);}
}
```

在这里，它有可能兜售相同的记忆区域，因为它们的活动范围没有重叠。您可以通过在aunion中加入a和b来节省大量的缓存空间：

```
//示例9.2b
空隙F3(库y)
        int a[1000];
        浮动b[1000];
    };
    如果
        (y){F1(
        a);
    }
    else{
        F2(b);
    }
}
```

当然，使用联合并不是一种安全的编程实践，因为你会得到不
如果使用a和重叠，编译器发出警告。您应该只对占用大量缓存空间的大对象使用此方法。把简单的变量放入联合中则不是
最优的，因为它阻止了使用寄存器变量。

9.5数据的对齐

如果变量存储在可整除的内存地址，则访问最有效
根据变量的大小。例如，一个双节点需要8个字节的存储空间。因此，它最好被存储在由8可整除的一个地址处。大小应该总是a
功率的2。大于16个字节的对象应该存储在一个由16个可整除的地址上。您通常可以假设编译器会自动处理这种对齐方式。

结构和类成员的对齐导致了高速空间的浪费，如示例中所示7.39页53.

您可以选择通过缓存线化来对齐大型对象和数组，这通常是
64字节。这可以确保对象或排列的开始与
显示在一个高速缓存行的开始位置。一些编译器会自动对齐大型静态数组，但你也可以通过写作来明确地指定对齐：

```
alignas (64) int BigArray[1024];
```

请参见第页100和133，用于讨论如何对齐动态分配的内存。

9.6动态内存分配

对象和数组可以动态分配与新的和删除，或malloc和自由。当我在汇编时不知道我所需要的内存量时，这可能很有用。这里可以提到动态内存分配的四个典型用途：

·当数组的大小在编译时未知时，可以动态分配大数组。

·当编译时对象总数未知时，可以动态分配可变数量的对象。

·可以动态地分配可变大小的文本字符串和类似对象。

·对于堆栈来说太大的数组可以动态分配。动态内存分配的优点是：

·在某些情况下给出了更清晰的程序结构。

•不会分配超出所需的空间。这使得数据缓存比为了覆盖最坏情况而将固定大小的数组做得非常大时更有效
最大可能内存需求的情况。

·当无法预先给出所需内存空间量的合理上限时有用。

动态内存分配的缺点是：

内存的动态分配和d分配过程比其他存储需要更多的时间。请参见第页25.

当不同大小的对象被分配并按随机顺序进行重新分配时，堆空间将变得碎片化。这使得数据的抱怨效率低下。

如果分配的数组满了，可能需要调整大小。这可能需要重新分配一个更大的内存块，并复制整个内容
到新的块。任何指向旧块中的数据的指针都是无效的。

当堆空间过于碎片化时，堆管理器将启动垃圾收集。这种垃圾收集可能在不可预测的时间开始，并在用户等待的程序流延迟
答复

程序员的责任是确保所有已分配的东西也被释放。如果不这样做，将导致堆文件失败。这是一个常见的编程错误，被称为我濒死漏。

这是程序员的责任，以确保没有任何对象被访问
在ithas被经销商之后。不这样做也是一个常见的程序错误。

所分配的内存可能无法进行最佳对齐。请参见第页133.关于如何对齐动态分配的内存
。

立即优化使用指针的代码是困难的，因为它不能排除混叠(参见页81).

当编译时不知道行长度时，矩阵或多维数组效率低下，因为计算行地址需要额外的工作来访问。编译器可能无法通过归纳法变量对其进行优化。

在决定是否这样做时，权衡优点和缺点是很重要的
使用动态内存分配。因此，没有理由使用动态内存分配
当数组的大小或编译时对象数已知时，或可以定义重新捕获上限。

当动态内存分配的计数为时，动态内存分配的代价可以忽略不计
有限的动态内存分配可以有利时，程序有
一个或几个大小可变的数组。将数组非常大以覆盖最坏情况的另一种解决方案是浪费缓存空间。一个程序有几个大数组的情况，并且每个数组的大小是关键步幅的倍数(见上图，页91)很可能会在数据缓存中引起争论。

如果一个数组中的元素数量在程序执行过程中增加，那么最好从一开始就分配最终的数组大小，而不是分配更多的空间
一步一步。在大多数系统中，不能增加具有
已经分配了。如果无法预测最终大小或预测结果为
太小，则需要分配一个新的更大的内存块并复制
旧内存块的内容进入新的更大内存块的开头。当然，这是无效的，并且会导致堆空间变得支离破碎。另一种方法是保存多个内存块，可以是带有内存索引的链接listor的形式
街区。具有多个内存块的方法使得对单个arrayelements的访问更加复杂和耗时。

可变数量的对象的集合通常被实现为linkedlist。每个
链表中的元素有自己的内存块和下一个块的指针。链表的效率低于线性数组，原因如下：

每个对象分别进行分配。分配、交易和资产重组需要相当长的时间。

这些对象不会连续存储在主题中。这使得数据的快速计算的效率降低了。

额外的内存空间用于链接指针和堆管理器为每个分配块存储的信息。

浏览链接列表比循环浏览线性阵列要花费更多的时间。在加载上一个链接指针之前，不能加载任何链接指针。这个
创建一个关键的依赖链，以防止无序执行。

通常为所有对象（内存池）分配一个大内存块比为空间空间对象分配一个小内存块更有效。

使用新的和删除的一个鲜为人知的替代方法是分配可变大小的数组
alloca.这是一个在堆栈上而不是在堆上分配内存的函数。当从调用alaca的函数返回时，空间自动释放。当使用分配a时，无需明确地释放空间。这个
新的和免费的优点：

由于微处理器对堆栈有硬件支持，因此微处理器上的分配开销很小。

由于堆栈的最后一个自然属性，内存空间永远不会支离破碎。

取消分配没有成本，因为它会在函数返回时自动运行。不需要进行垃圾收集。

所分配的内存与堆栈中的其他对象相邻，这使得数据缓存非常有效。

下面的示例展示了使用alaca创建一个可变大小的数组：

```
// Example9.3
# include< malloc.h>

空某个函数（int n）{ if（n>0）{
       //生成n个浮点的动态数组：
       float*DynamicArray=(float*)alloca(n*sizeof(float));
       //（有些编译器使用name_alloca）
       for(int i=0;i<n;i++){
          DynamicArray[i]=WhateverFunction(i);
          //...
       }
    }
}
```

显然，函数不应该返回任何指针或对它所拥有的任何东西的引用
用alloca分配，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allo ca的限制，请参见编译器手册。

C99扩展支持可变大小的数组。这个特性是有争议的，只在C中可用，在C++中不可用。您可以使用alloca而不是可变大小数组，因为它
提供相同的功能。

9.7数据结构和容器类

每当使用动态内存分配时，建议将分配的
memoryinto容器类。容器类必须有一个构造函数
所有分配的东西也被取消分配。这是防止记忆的最好方法
与动态内存分配相关的泄漏和其他常见的编程错误。容器类的一个简单替代方案是使用智能指针，如第页上所示37.

容器类还可以方便地添加边界，并为更先进的数据结构进行先进先出或最后进先出访问、排序和搜索工具、二进制树、哈希映射等。

通常是以模板的形式创建容器类，其中它们所包含的对象的类型是作为模板参数提供的。使用模板没有额外的工作条件。

现成的容器类模板可以在标准的C-++库中使用
C++11及以后的标准。使用现成的集装箱的优点是，你不用重新发明轮子。标准的容器是通用的、灵活的、经过良好测试的，并且对许多不同的用途都非常有用。

但是，虽然标准容器的设计具有通用性和灵活性，但效率却被牺牲了。执行速度、内存经济性、缓存效率和代码大小都是低优先级的。特别是内存分配在许多情况下是不必要的浪费

标准容器。许多容器类模板被实现为链表，为容器中的每个对象分配一个或多个新内存块
。

std：：vector将所有对象存储在一个连续的内存块中，但是这个内存
每次填充时，块都会被重新分配，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到astd：：vector中，分别导致7个大小为1、2
、3、4、6、9和13个对象的内存分配（MS Visual Studio 2019）。这种浪费行为
可以通过callingstd：：vector：：reservewith防止在添加第一个对象到向量之前所需的最终
大小的预测或估计。大多数其他容器都没有这样的预先保留内存的特性。

新内存和删除内存的频繁分配和删除分配导致
使内存变得碎片化，从而使缓存非常高效。如上所述，内存管理和垃圾收集有很大的超额成
本。

标准容器的通用性也以代码大小的成本。事实上，
标准库一直因代码膨胀和复杂性而受到批评
(en。维基百科。标准模板库)。按标准存储的对象__
容器允许具有构造函数和析构函数。每次移动一个对象时，每个对象的移动构造函数或复制
构造函数和析构函数都会被调用，这种情况可能经常发生。如果存储的对象本身是容器，则
这是必要的。但是，将一个矩阵作为向量的向量，肯定是非常明显的
效率低下的解决方案。

许多容器使用领英列表。链表是使容器可流行的重要方法，但效率非常低。具有连续内存
的线性数组通常是
比领英名单快得多。

用于容器的、用于访问容器元素的所谓迭代器是
对于多名程序员来说，它们使用起来很麻烦，而且如果你可以使用a
带有简单指数的线性列表。一个好的编译器可以通过优化迭代器的额外标题，但不是所有的
。

幸运的是，在执行速度、内存经济性和小代码大小比代码通用性更优先的情况下，有更有
效的替代方案。最
重要的补救措施是内存池。将许多对象存储在一起更有效
一个大的内存块，而不是将每个对象存储在它自己分配的内存块中。如果没有需要调用的
Move构造函数、copyconstructors或Destructor，则可以通过一次调用memcpy来复制或移动
包含许多对象的largeblock，而不是单独移动每个对象。显示了一些高效容器类的示例页面下
方101.容器类的集合在
万维网。阿格纳。org/optimize/cppexamples。拉链。向量和矩阵的容器类可以在上的向量
类库中找到github.

为特定目的选择容器时，应考虑以下因素：

是否包含一个或多个元素？如果容器只包含一个元素，那么您应该使用本地分配或者一
个智能指针(参见页面37).

编译时间的大小已知吗？如果元素的最大数量在编译时已知，或者可能存在一个不太大
的上限，那么最优解是a
固定大小数组。但是，如果数组或容器对于堆栈太大，动态内存分配可能会受益。

在存储第一个元素之前，是否已知其大小？如果要存储的元素的总数在存储的第一个元素之前就已知了，或者如果有一个合理的估计或
上限可以做出，那么它最好是使用一个容器，如
它允许您预先保留所需的内存量，而不是在内存块出现时进行分段分配或重新分配太小了。

•尺寸是大还是小？即使您需要搜索、添加和
在未排序数组中进行线性搜索肯定不是最有效的算法，但如果数组很小，它可能仍然是最好的解决方案。

•对象是否连续编号？如果对象由连续的索引或有限范围内的键标识，那么简单的数组是最有效的解决方案。

·是否需要多维结构？矩阵或多维数组应
存储在一个连续的备忘录块中。不要为每一行或每列使用一个容器。嵌套容器当然不是最有效的解决方案。如果每行的元素数是编译时已知的常数，则访问速度更快。

对象是否以aFIFO的方式访问？如果以优先输入先出（FIFO）的方式访问对象，则使用队列。实现循环缓冲区的队列要有效得多。参见示例9.6页103.

对象是否以aLIFO的方式访问？如果以最后输入先出（LIFO）的方式访问对象，则使用具有堆栈顶部索引的线性数组。参见示例9.5页102.

对象是否由akey标识？如果键值被限制在一个有限的范围内，则可以使用简单的数组。如果对象的数量很高，那么这是最有效的
解决方案可以是神的树或哈希图。

物品是一种自然的排序吗？如果你需要做一些类似的搜索：“对x最重要的元素是什么？”或者“x安迪之间有多少个元素？”然后你可以使用一个排序列表或一个二叉树。_

•添加所有对象后是否需要搜索？如果搜索设施是
需要，但只有在所有对象都存储在容器中之后，才需要线性
数组将是一个有效的解决方案。添加所有元素后对数组进行排序。然后使用二进制搜索来查找元素。如果搜索对象很少，而搜索很频繁，并且列表不太大，则排序列表可能仍然有用。哈希映射
也可能是一种有效的解决方案。

•在添加所有对象之前是否需要搜索？如果搜索设施是
需要，并且可以随时添加或删除对象，那么解决方案就更加复杂了。如果元素的总数很小，那么排序列表仍然是最有效的解决方案，因为它很简单。但是排序列表可能非常
如果列表很大，则效率低下。在列表中插入新元素会导致所有
在要移动的序列中的后续元素。在这种情况下，二叉树或哈希映射更有效。阴树可能有用元素具有自然元素
在一个特定的时间间隔内，会有对元素的搜索请求。如果元素具有特定的顺序，但由单键标识，则可以使用哈希映射。

物体是混合的类型或大小吗？可以存储不同的对象
在同一个记忆池中具有不同长度的类型或字符串。看
www.agner.org/optimize/cppexamples.zip.如果元素的数量和类型为

在编译时就知道了，那么就不需要使用附件或记忆池了。

沙痂校准吗？一些应用程序要求数据在环形地址处对齐。
特别是内在向量的使用需要对齐到可被16、32或64整除的地址。对数据结构进行对齐，以解决缓存线化（通常是64），在某些情况下可以提高性能。C++98及更高版本
标准保证已分配的内存与指定给新操作符的对象类型正确对齐。

沙痂多线？如果有多个线程扫描同时添加、删除或修改对象，则容器类通常不是线程安全的。这样做更有效率
为每个线程多线程应用程序分离的容器，而不是暂时锁定一个容器为每个线程独家访问。

指向所包含对象的指针吗？创建一个指向对象的指针可能不安全
在容器内部，因为容器可以在需要内存分配时移动对象。容器内的对象应该通过其索引或键入容器来标识，而不是通过指针或引用来标识。但是，可以将一个指针或引用到这样的对象传递到不添加或修改ve的函数
任何对象，如果没有其他线程访问该。

•容器可以回收吗？创建和删除的成本很高
容器。如果程序逻辑允许的话，重用一个容器可能比删除它并创建一个新的容器更有效。

标准C++容器

标准C++容器类模板为组织数据结构的常见问题提供了方便且经过充分测试的解决方案。不幸的是，这些容器相当
复杂且并不总是非常有效。特别是链接列表是低效的，因为数据对象存储在内存的碎片位中，缓存很差，因此-
序列。代码很大，开销很大。一个好的优化编译器可以剥离
大部分的开销，但是代码仍然难以理解和调试
因为过于复杂。这里列出了最常见的C++容器类模板：

•std：：array。这就像一个simplearray。大小在编译时是固定的。
没有界限检查。

·std：：vector。这类似于线性数组。元素存储在一个
连续存储器块。尺寸可以增长。增长是有组织的，当前一个内存块变得太小时，会分配一个新的内存块（大50%）。每次重新分配时，所有数据都必须移动到新的内存块e。在添加第一个元素之前，使用reservefunction对所需的大小进行估计或上限是很重要的。这将减少重新分配的次数。std：：vector是最简单和最有效的标准
具有动态内存分配的容器。

•标准：：甲板。这是双端edqueue。它既可以用作线性数组，也可以用作队列，在队列中可以有效地插入和删除元素
结束。一个队列可能比一个链表分配更少的内存块，因为如果元素很小，每个内存块可以包含多个元素。它处于低点
随机访问这个列表中的任何元素，正好类似于一个数组。一个模型提供了线性数组和领英列表之间有用的妥协。

Subarkstd：：转发列表。这是一个领英名单。一个新的内存块被分配d个元素。这些元素只能按顺序访问。该列表可以被重新绑定，但排序并不是特别有效。

• std::list.这是一个双重领英名单。元素可以添加和删除元素
两头类似于aforward_list的列表，但是列表可以向前和向后历。

std：堆栈。这是最后先出（LIFO）列表。这是一种特例。

• std::set.这是一个订单列表。这些对象是由它们的值和
按其价值排序。对象可以随时以随机的顺序进行添加和删除。搜索对象是有效的。
作为自我平衡实现的平衡
二叉树（红黑树）。如果对象以随机顺序添加并按随机顺序搜索，这是有用的。

•std：：map。这类似于std：：set。与std：：set的区别是一个映射
包含键值对。对象由它们的键标识和排序，而值包含与键相关的其他信息。

·std：：unordered_set。如果对象的顺序不自然，可以用它来代替std：：setif。它被实现为散列映射。可以有效地添加和删除对象。找到一个有效的对象。

·std：：unordered_map。这类似于anunordered_set，但包含键值对。

创建自己的容器类

一个高效的容器类应该分配尽可能少的内存块。最好是，所有数据都应该存储在一个连续的内存块中。只有当在编译时不知道合理的最大大小时，才需要动态内存分配。的类型容器包含的对象可以方便地指定为templateparameter。记得定义一个析构函数来释放所有已分配的内存。

有关容器类模板的更多示例，请参见
万维网。阿格纳。org/optimize/cppexamples。拉链。向量和矩阵的高效容器可以在上的向量类库中找到github.

示例9.4显示了一个线性数组，其中的大小在构造函数中指定。使用[]运算符访问元素。添加了索引超出范围的错误检查。

```
//示例9.4
//带边界检查的简单数组模板
模板<typename T>//T是元素类型
类安全阵列{
受保护：
    T*p=0；//指向数组的指针
    int size=0；//数组大小
公众：
    //构造函数
    SafeArray(int n){//n是所需的大小
        p=新T[n]；//分配内存
        如果(p==0){
            //..此处显示错误消息：内存分配失败}
        大小=n；//保存大小
    }
```

```
    //返回数组的大小int
    getSize()const{
        返回大小；}
    //[]数组索引运算符
    T&operator[](inti){
        如果((unsigned int)i>=(unsigned int)size){//索
            引超出范围
            //..这里的错误消息：索引超出范围返回p[0]；
        }
        //没有错误
        返回p[i];                              //返回对元素的引用
    }
    ~SafeArray(){              //析构函数
        如果（p）删除[]p       //释放内存
    }
};

//...

//使用示例：
SafeArray<float>myArray(100);

for（int i=0；i<100；i++）
   {myArray[i]=0；
}
myArray[2]=9；
```

示例9.5显示了一个后进先出列表（也称为堆栈），其中固定的maximumsize被指定为templateparameter。

```
//示例9.5
//后进先出列表模板
模板<typename T，int maxsize>
类LIFO堆栈{
保护
    intn=0；//列表中的对象数
    T列表，数据缓冲区
平民
    LIFOStack（）{//构建器
        n=0;}
    如果（n>=最大大小）{
            返回false；//如果列表已满，则返回false
        }
        list[n]=x;
        n++；//增量计数
        如果成功，则返回真
    }
    不获取（）{//从列表中获取对象
        如果（n<=0）{
            //错误：列表为空
            //在这里放置错误消息：列表为空
            返回T(0)；}
        n--;
        返回列表                   //减少计数//返回对
                                   象
    }
    int getNum（）常量{返
        回n；                      //告知列表中的对象数量
    }
```

```
};

//...

//使用示例：
LIFOStack< int,100> myQ;
//将数字输入
为（int i=0；i<5；i++）{myQ。把（i
    x10）；
}
//以相反的顺序获取数字
而（ myQ。getNum()>0){
    printf("\n%i", myQ.收到
}
```

样例9.6显示了一个先入先出的列表（也称为队列），其中有一个固定的最大值sizeis被指定为一个模板参数。不需要动态内存和所有的位置。该列表被实现为一个循环缓冲区。

```
// Example9.6
//模板的先入先出列表
template< typename T, int maxsize>
类FIFOQueue{
保护
    指向当前头部和尾部的指针
    列表中的对象数量
    循环缓冲区列表
平民
    FIFO队列（）{//构造函数
        头=尾=列表；//初始化
        n=0;}
    如果（n>=最大大小）{
            返回false；//如果列表已满，则返回false
        }
        n++；//增量计数
        *head=x；//复制x到列表
        头++；//增量头指针
        if(head>=list+maxsize){
            head=列表；//环绕
        }
        返回true；//如果成功则返回true
    }
    T get(){//从列表中获取对象
        如果(n<=0){
            //在此输入错误消息：列表为空
            返回T（0）；                    //如果可能，返回零对象
        }
        n--；                   //递减计数
        T*p=尾部；              //指向对象的指针
        尾++；                  //递增尾部指针
        if(tail>=list+maxsize){
            尾部=列表；//环绕
        }
        返回*p；//返回对象
    }
    int getNum()const{//告诉列表中的对象数
        返回n；
```

```
        }}
;

//...
```

```
//使用示例：
FIFOQueue<int，100>myQ;
//输入数字
for(int
    i=0;i<5;i++){myq.put(i*10);
}
//以相同的顺序输出数字
while(myq.getNum()>0){
    printf("\n%i", myq.get());
}
```

C++标准目前没有定义一种有效的方法来制作维度在运行时确定的矩阵。示例9.7显示了一个可能的解决方案：

```
// Example9.7
请为一个矩阵提供相应的//模板。尺寸标注是在运行时决定的

//这个模板生成了一个任意维数的矩阵
//没有边界检查
template< typename T>
类矩阵{
平民
    具有行数和列数的//构造函数
    矩阵( int行、int列）{
        //保存行数和列数，这是一个->行，=行
        ;
        这个->列=列;
        //分配内存
        p.新的=，列;
        如果（p==0）{
            //..此处出现错误消息：分配失败}
        else{
            //将所有元素初始化为零
            对于（int i=0; i<行*列; i++）{p[i]=0;
            }
        }
    }
    //获取行数
    int nrows() const{
        返回行;
    }
    //获取列数
    int ncols() const{
        returncolumns;
    }
    //解构器
    ~ Matrix(){
        如果(p)删除，则释放内存
    }
    //Operator[]给出一个指向行号r的指针。
    //带有列号的第二个[]给出单个元素
    T*运算符[](intr){
        //无边界检查
        返回p+r*列;
    }
受保护：
    int行; //行数
```

```
    int列; //列数
    T*p; //指向内存的指针
};
```

```
//...

//使用示例：
//制作一个8行10列的双精度矩阵
矩阵<双>垫（8，10）；

//在第3行第5列中输入一个值
mat[3][5]=12.34;

//打印出矩阵
for(int r=0;r<mat.nrows();r++){
    for(int c=0;c<mat.ncols();c++){
        printf（“%5.2 f”，mat[r][c]）；
    }
    printf("\n")
; }
```

我在中提供了其他几个容器类模板的示例
www.agner.org/optimize/cppexamples.zip。这些可以用作
如果不需要标准容器的通用性和灵活性，则使用标准C++容器。您可以编写自己的容器类或修改可用的容器类以满足特定需求。

9.8弦

文本字符串通常具有在编译时未知的可变长度。在string或wstring等类中存储文本字符串是在每次创建或修改字符串时使用new和delete来分配一个新的内存块。如果程序创建或修改许多字符串，这可能是非常低效的。

在大多数情况下，处理字符串的最快方法是带有characterarrays的老式Cstyle。字符串可以用C函数操作，如strcpy、strcat、strlen，
sprintf等。但是请注意，这些函数没有检查数组的溢出。数组溢出会导致不可预测的错误很难诊断。程序员有责任确保数组足够大，以处理包括终止零在内的字符串，并在必要时进行溢出检查。常用字符串函数的快速版本以及高效
asmlib库中提供了字符串搜索和解析函数万维网。阿格纳。
org/optimize/asmlib。拉链。

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，如上所述。本手册附录ix中提供了示例万维网。阿格纳。
org/optimize/cppexamples。拉链。

9.9顺序访问数据

当按顺序访问数据时，缓存的工作效率最高。当反向访问数据时，它的工作效率会低得多，当以随机方式访问数据时，它的工作效率会低得多。这适用于读取和写入数据。

多维数组应该在innermostloop中更改lastindex的情况下访问。这反映了元素在内存中存储的顺序。示例：

```
//示例9.8
const int NUMROWS=100，NUMCOLUMNS=100;
int矩阵[NUMROWS][NUMCOLUMNS];
int行、列;
对于(row=0;row<NUMROWS;row++)
```

```
对于(column=0;column<NUMCOLUMNS;column++)
    矩阵[行][列]=行+列;
```

不要交换两个循环的顺序（除了存储顺序相反的inFortran）。

9.10大型数据结构中的缓存争用

顺序访问多维数组并不总是可能的。一些应用程序（例如非线性代数）需要其他访问模式。这可能会导致严重的损伤
大矩阵中行之间的距离恰好等于criticalstride，如第页所述92.如果矩阵行的大小(以字节为单位)是2的高次方，就会发生这种情况。

下面的示例说明了这一点。我的例子是一个函数，它转换了水二次矩阵，即每个元素矩阵[r][c]与元素交换
matrix[c][r].

```
// Example9.9a
constintSIZE=64; //矩阵中的行/列数

使用函数来转换矩阵//定义一个宏来交换两个数组元素:
    #定义交换（x，y）{temp=x; x=y; y=temp; }

    intrc; 双温度;
    对于（r=1; r<SIZE; r++）{//循环通过行
        用于（c=0; c<r; c++）{//对角线下方的循环列
            交换（a[r][c]，a[c][r]）; //交换元素
        }
    }
}

空测试（）{
    alignas (64)                    //按缓存对齐          管道尺寸
    双矩阵                          //定义矩阵
    转座（矩阵）;                   //调用转座          功能
}
```

转换一个矩阵和在对角线上反射它是一样的。每个元素
对角线下方的矩阵[r][c]与对角线上方镜像位置处的元素矩阵[c][r]交换。示例中的cloop9.9 a从最左边的一列到对角线。对角线上的元素保持不变。

这个代码的问题是，如果对角线下面的元素矩阵[r][c]是逐行访问的，那么对角线上面的镜像元素矩阵[c][r]是
按列访问。

假设我们在奔腾4计算机上运行64 × 64矩阵的代码，其中1级数据缓存为8 kb=8192字节，4路，行大小为64。每个
cacheline可以容纳8个双字节，每个8个字节。临界步幅为8192/4=2048字节=4行。

让我们看看循环内部发生了什么，例如当r=28时。我们从对角线下方的第28行获取元素，并将这些元素与对角线上方的第28列进行交换
对角线。行28中的前八个元素共享相同的高速缓存行。但这八个
元素将进入第28列中的8个不同的缓存行，因为缓存行跟随行，而不是列。这些缓存行的四分之一属于缓存中的同一集合。当我们到达第28列中的元素数16时，该缓存将驱逐该列中元素0所使用的缓存行。17号将驱逐1号。编号为18，将会

Evict编号2等。这意味着我们在对角线上使用的所有缓存线都有
在我们交换第29列和第29行的时候。每个高速缓存线都必须是
重新加载8次，因为在连接之前被驱逐。我有
通过测量转置矩阵所需的时间来证实这一点9.9 aona奔腾4与不同的矩阵大小。我的实验结果如下。这个
每个每个元素的时间为时钟周期。

矩阵大小	总千字节	定时器元件
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

表9.1。不同大小矩阵的换位时间，每个元素的时钟周期。

该表显示，当矩阵的大小是1级缓存大小的倍数时，转置矩阵需要多花40%的时间。这是因为临界步幅是矩阵线大小的倍数。延迟小于重新加载1级所需的时间
从二级缓存中缓存，因为无序执行机制可以预取数据。

当争用发生在二级缓存中时，效果会更加显著。二级缓存为512 kb，8种方式。2级缓存的关键步长为512KB/8=64kb。这个
对应于512 × 512矩阵中的16行。我的实验结果在表中9.1表明，当竞争发生在第2级时，转置矩阵需要六倍的时间
当争用不发生时缓存。这就是为什么这种效应对2级高速缓存争用比对1级高速缓存争用强得多的原因，即2级高速缓存一次不能预取超过一行。

解决该问题的一个简单方法是使矩阵ix中的行比需要的长，以避免临界步长是矩阵行大小的倍数。我试着让
矩阵512 × 520，并保留最后8列未使用。这删除了内容，时间消耗减少到36。

在某些情况下，不可能将未使用的列添加到矩阵中。为了
例如，一个数学函数库应该能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方块，一次处理一个方块。这被称为方形块或平铺。在实例中说明了该技术9.9 b.

```
//示例9.9 B
void转置(double a[SIZE][SIZE]){
    //定义宏以交换两个元素：
    #define swapd（x，y）{temp=x；x=y；y=temp；}
    //检查是否会发生二级缓存争用：
    if(SIZE>256&&SIZE%128==0){
        //预计会有缓存争用。使用方形阻挡：
        int r1、r2、c1、c2；双温；
        //定义正方形的大小：
        常量int TILESIZE=8；//大小必须能被TILESIZE整除
        //为所有正方形循环r1和c1：
```

```
            for(r1=0;r1<SIZE;r1+=TILESIZE){for(c1=0;c1<r
                1;c1+=TILESIZE){
                    //循环r2和c2对于sqaure内部的元素:
                    对于(r2=r1;r2<r1+TILESIZE;r2++){
                        对于(c2=c1;c2<c1+TILESIZE;c2++){
                            swapd(A[r2][c2], A[c2][r2]);
                        }
                    }
                }
                //对角线上只有半个正方形。
                //这个三角形是单独处理的:
                对于(r2=r1+1;r2<r1+TILESIZE;r2++){
                    对于（c2=r1; c2<r2; c2++）{
                        swapd(A[r2][c2], A[c2][r2]);
                    }
                }
            }
        }
        else{
            //没有缓存争论。使用简单的方法。
            //这是示例中的代码9.9a:
            intrc; 双温度;
            对于（r=1; r<SIZE; r++）{//循环通过行
                用于（c=0; c<r; c++）{//对角线下方的循环列
                    交换（一个[r][c], 一个[c][r]）; //swap元素}
            }
        }
    }
```

在我的实验中，这段代码为一个512×512矩阵使用了50个时钟周期元素。

2级缓存的约束非常昂贵，因此对它们进行测量是非常重要的。因此，您应该注意一个矩阵中列的数量是大幂为2的情况。一级缓存的成本较低。使用
像一级缓存的方形阻塞这样的复杂技术可能不值得付出努力。

在《性能》一书中进一步描述了方块块和类似的方法
《数字密集码的优化》，作者：S.Goedecker和A. Hoisie，SIA M2001。

9.11显式缓存控制

具有SSE和更高指令集的微处理器具有某些指令，允许您操作数据缓存。这些指令可通过intrinsic访问
功能。

功能	程序集名称	本征函数名	指令集
预取	预取	_mm_预取	SSE
存储4字节，无需缓存	移动	_mm_stream_si32	SSE2
存储8字节，无需缓存	移动	_mm_stream_si64	SSEs
存储16字节，不带缓存	移动	_mm_stream_ps	SSE
存储16字节，不带缓存	移动	_mm_stream_pd	SSE2
存储16字节，不带缓存	移动	_mm_stream_si128	SSE2

表9.2。高速缓存控制指令。

除了表中提到的指令之外，还有其他高速缓存控制指令9.2，如flush和fence指令，但这些几乎与优化无关。

预取数据

PREFETCH指令可用于获取我们期望在程序流的后面使用的缓存行。然而，在我测试的任何示例中，这都没有提高执行速度。原因是现代处理器自动读取数据
无序的执行和先进的预测机制。现代微处理器能够自动精确地为包含多个具有不同步幅的流的常规访问模式预取数据。因此，如果数据访问，您不能显式地引用数据
可以用固定的步幅按规则的模式排列。

未缓存内存存储

具有缓存错误的写比未缓存的读取更昂贵，因为写使用整个缓存行进行读取和回存。

所谓的非时时写指令（MOVNT）就是为了解决这个问题而设计的。
这些指令可以直接写入内存，而不需要加载高速缓存行。这是
有利的情况下，我们正在写入未缓存的内存，我们不会在缓存行被之前从相同或附近的地址再次读取。不要将非时间写或读写混合到同一内存区域。

例如，非时态写指令不适合使用　　9.9因为我们在同一个地址阅读和写作，所以现金线就会过时。如果我们修改
样例9.9所以它只写，那么非时间写指令的效果就变得明显。下面的示例将转换一个矩阵，并将结果存储在一个不同的矩阵中
大批

```
// Example9.10a
constintSIZE=512；//矩阵中的行数和列数

//函数来转置和复制矩阵
空白转位副本（双a大小，双b大小），c；
    对于（r=0；r<大小；r++）{
        对于（c=0；c<大小；c++）{
            a[c][r]=b[r][c];
        }
    }
}
```

这个函数以列的方式写入mat矩阵，其中关键的串将在第1级和第2级缓存中写入一个新的缓存行。使用
非时间写指令可防止二级缓存加载矩阵a的任何数据线：

```
//示例9.10b.
#include"xmmintrin.h"//用于内在函数的头

//此函数在不加载缓存行的情况下存储一个double：
静态内联void StoreNTD(double*dest,double
    const&source){_mm_stream_pi((m64*)dest,*(m64*)&source);//MOVNTQ
    _mm_empty();//EMMS
}

常量int大小=512；//矩阵中的行数和列数
```

```
//转置和复制矩阵的函数
无效转座复制（双A[大小][大小]，双b[大小][大小]）{intr，c；
    对于(r=0;r<SIZE;r++){
        对于(c=0;c<SIZE;c++){
            StoreNTD(&A[c][r]，b[r][c]);
        }
    }
}
```

在奔腾4计算机上测量不同基质尺寸的每个基质单元的执行时间。测量结果如下：

矩阵尺寸	Timeper元素示例 9.10 A	Timeper元素示例 9.10 B
64×64	14.0	80.8
65×65	13.6	80.9
512×512	378.7	168.5
513×513	58.7	168.3

表9.3。转置和复制不同大小矩阵的时间，每个元素的时钟周期。

如表如图9.3所示，当且仅当可以预期2级高速缓存未命中时，在没有高速缓存的情况下存储数据的方法是有利的。64 × 64的矩阵大小会导致水平失误-
1缓存。这对总执行时间几乎没有任何影响，因为存储操作上的缓存未命中不会延迟后续指令。512 × 512矩阵尺寸
导致二级缓存中的未命中。这对执行时间有非常显著的影响，因为内存业务已经饱和。这可以通过使用非时间来改善
写道。如果可以通过其他方式防止缓存争用，如第1章所述9.10，那么非时间写指令是非最优的。

使用表中列出的说明有一定的限制9.2.所有这些
指令要求单机处理器具有SSE或SSE2的结构集，如表中所示。16字节指令movntp，MOVNTPD和MOVNTDQ要求
操作系统支持XMM寄存器；请参见第页135.

10多线程

CPU的时钟频率受到物理因素的限制。增加其数量的方法
当时钟频率有限时，cpu强度程序的吞吐量是为了同时做多件事。有三种方法：

使用多个cpu或多核心cpu，如本章所述。

使用现代cpu的无序功能，如第章所述11.

使用现代cpu的向量操作，如第章所述12.

大多数现代cpu有两个或两个，而且可以预期的数量
核心技术在未来将会不断增长。有多个cpu或cpucore，我们需要使用
工作到多个线程。这里有两个主要原则：功能分解
和数据分解。这里的功能分解意味着不同的线程在做不同种类的工作。例如，一个线程可以负责用户界面，另一个线程可以负责与远程数据库的通信，第三个线程可以进行数学计算。重要的是用户界面不是相同的

线程是非常耗时的任务，因为这会给出恼人的长且不规则的响应时间。将耗时的任务放在低优先级的单独线程中通常很有用。

然而，在许多情况下，只有一项任务消耗了大部分资源。在这种情况下，我们需要将数据拆分为多个块，以便利用多个块
处理器内核。然后，每个线程应该处理自己的数据块。这是数据分解。

在决定并行执行操作是否有利时，区分粗粒度并行和细粒度并行是很重要的。
并行是指一长串操作可以独立于并行运行的其他任务执行的情况。细粒度并行
一个任务被分成许多小的子任务，但是在需要与其他子任务协调之前，不可能在一个特定的子任务上工作很长时间。

多线程使用粗粒度并行比使用细粒度并行更有效，因为不同线程之间的通信和同步很慢。如果粒度太细，则将任务s分成多个线程是不利的。无序执行（章节11）和向量运算（第12）是利用细粒度并行的更有用的方法。

使用多个CPU内核将工作分成多个线程的方法。的使用
第页讨论了线程61.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调用来确定（例如，Windows中的GetProcessAffinity掩码）。

有几种方法可以在多个CPUCore之间划分工作负载：

·定义多个线程，并为每个线程投入等量的工作。此方法适用于所有编译器。

•使用自动并行化。Gnu和英特尔编译器可以自动检测代码中并行化的机会，并将其分成多个线程，但编译器可能无法找到数据的最佳分解。

•使用OpenMP指令。OpenMP是指定并行处理inC++和FORTRAN的标准。大多数编译器都支持这些指令。见
万维网。开放议员。org和编译器手册了解详细信息。

•使用std：：thread。它的功能比OpenMP少，但是跨平台标准化。

·使用具有内部多线程的函数库，例如。英特尔数学内核库。

多个CPU内核或逻辑处理器通常共享相同的缓存，至少在最后一个缓存级别，在某些情况下甚至共享相同的1级缓存。共享相同高速缓存优点是测试线程之间通信变得更快
可以共享相同的代码和只读数据。缺点是缓存将
如果线程使用不同的内存区域，则会出现缓存争用。

只读数据可以在多个线程之间共享，而
每个线程的修改应该是分开的。有两个或更多线程是不好的
写入同一缓存行，因为线程将验证彼此的缓存并导致较大的延迟。创建线程特定数据的最简单方法是在

线程函数，以便它存储在堆栈上。每个线程都有自己的堆栈。
或者，您可以定义一个结构或类来包含特定于gthread的数据，并且
为每个线程创建一个实例。这个结构或类应该至少按缓存行大小对齐，以避免多个线程写入同一个缓存行。在当代处理器上，缓存行大小通常为64字节。在未来的处理器上，高速缓存行大小可能更大（128或256字节）。

线程之间的通信和同步有多种方法，如信号量、互斥体和消息系统。所有这些方法都很耗时。因此，应组织数据和资源，以便必要的数量
线程之间的通信被最小化。例如，如果多个线程共享同一个队列、列表、数据库或其他数据结构，那么您可以考虑
可以给每个线程自己的数据结构，然后合并多个数据
当所有读取都完成了耗时的数据处理时。

如果线程竞争相同的资源，在只有一个逻辑处理器的系统上运行多个线程并不是一个优势。但这可能是个好主意-
将计算消耗到优先级低于用户界面的单独线程中。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的响应时进行计算。

10.1并发多线程

许多微处理器能够在每个内核中运行两个线程。例如，具有四个内核的处理器可以同时运行八个线程。此处理器有四个物理
处理器但是八个逻辑处理器。

“超线程”是英特尔对并发多线程的术语。在同一个内核中运行的两个线程总是会竞争相同的资源，比如缓存和执行单元。如果任何共享资源是性能的限制因素，那么使用并发多线程就没有优势。相反，每个线程可以以更低的速度运行
由于缓存驱逐和其他资源冲突，速度减半。但如果很大
一部分时间用于缓存未命中、分支错误预测或长依赖链，然后每个线程将以单线程速度的一半以上运行。在这种情况下，使用并发多线程有一个优势，但是性能不会增加一倍。A
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。

通常有必要进行实验，以确定在特定应用中同时使用多线程是否有利。

如果同时多线程不是有利的，那么有必要查询某些
操作系统函数（例如，Windows中的GetLogic ALProcessorInformation）确定处理器是否具有并发多线程。如果是这样，那么你可以避免
仅使用偶数逻辑处理器（0、2、4等。）.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予一个线程高于另一个线程的优先级。
因此，经常会发生低优先级线程从运行在同一内核中的高优先级线程窃取资源的情况。操作系统有责任避免在同一个处理器内核中运行两个优先级相差很大的线程。
不幸的是，当代操作系统不能充分处理这个问题。

11乱序执行

除了一些小型低功耗CPU（英特尔凌动）之外，所有现代x86 CPU都可以无序执行指令或同时执行多件事情。下面的例子
showshow要利用此功能：

```
//示例11.1 a
浮动A、b、c、d、y;
y=a+b+c+d;
```

该表达式计算为((a+b)+c)+d。这是一个依赖链，其中每个添加都必须等待前一个添加的结果。您可以通过以下方式改进这一点：

```
//示例11.1 B
浮动A、b、c、d、y;
y=(a+b)+(c+d);
```

现在两个括号可以独立计算了。CPU将在完成（a+b）的计算之前开始计算（c+d）。这可以节省几个时钟周期。您不能假设优化编译器会更改示例中的代码11.1 ato 11.1 b自动，尽管这似乎是显而易见的事情。其原因
编译器不会对浮点表达式进行这种优化，因为这可能会导致精度损失，如第#页所述75.你必须设置括号
手动。

当依赖链很长时，它们的影响会更强。这通常是循环中的情况。考虑以下示例，它计算100Numbers的总和：

```
//示例11.2 a
常量int大小=100;
浮点列表【大小】，sum=0；int i;
对于（i=0；i<大小；i++）sum+=list[i];
```

这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isloop将需要大约500个时钟周期。您可以提高性能
通过展开循环并将依赖链分成两个：

```
//示例11.2 B
常量int大小=100;
浮点列表【大小】，sum1=0，sum2=0；int i;
对于（i=0；i<size；i+=2）
    {sum1+=list[i];
    sum2+=列表[i+1];
}sum1+=sum2;
```

如果只有一个加法器的微处理器从时间T到T+5对sum1进行加法，那么它可以从时间T+1到T+6对sum2进行另一次加法，整个循环只需要256个时钟周期。

每次迭代都需要前一次迭代的结果的aloop中的计算称为aloop携带的依赖链。这种依赖链可能很长，而且很长-
消费。如果这种依赖链能够被打破，将会有很多收获。这两个
求和变量sum1和sum2称为累加器。当前CPU剃掉一个或
两个浮点加法单元，这些单元是流水线的，如上所述，这样它就可以在前一个加法完成之前开始一个新的加法。

用于浮点加法和乘法的累加器的最佳数量可能是三个或四个，具体取决于CPU。

如果迭代次数不能被展开因子整除，展开循环就会变得更加复杂。例如，如果示例中列表中的元素数11.2 b如果是奇数，那么我们必须在循环之外添加最后一个元素，或者在列表中添加一个额外的虚拟元素，并使这个额外的元素为零。

如果没有循环携带的依赖链，则没有必要展开循环并使用多个累加器。具有无序能力的微处理器可以重叠迭代，并在前一次迭代完成之前开始一次迭代的计算。示例：

```
//示例11.3
常量int大小=100；int i；
浮点A【大小】、b【大小】、c【大小】；
浮点寄存器温度；
对于(i=0;i<size;i++){
    温度=A[i]+b[i]；
    c[i]=温度*温度；}
```

具有乱序能力的微处理器非常聪明。在示例中，它们可以在循环的一次迭代中检测寄存器温度的值　　11.3独立于中的值
上一次迭代。这允许在温度值为
已完成使用上一个值。它通过为Temp分配一个新的物理寄存器来实现这一点，即使出现在机器代码中的逻辑寄存器是相同的。这叫做寄存器重命名。CPU可以保存同一个逻辑寄存器的许多重命名实例。

这个优势是自动出现的。没有理由展开循环
温度1和温度2。现代CPU能够重命名寄存器并执行多个
如果满足某些条件，则并行计算。使CPU可以重叠循环迭代的计算的条件是：

·没有循环携带的依赖链。一次迭代的计算中没有任何东西应该依赖于前一次迭代的结果（除了循环计数器，如果它是整数，它是快速计算的）。

·所有中间结果都应该保存在寄存器中，而不是内存中。重命名机制只适用于寄存器，不适用于内存或缓存中的变量。大多数
编译器将使temp成为示例中的寄存器变量11.3即使没有registerkeyword。

●循环分支应该被预测。如果重复计数较大或恒定，则没有问题。如果循环计数很小并且在变化，那么CPU可能偶尔预测循环退出，而实际上它没有，因此无法进行下一次计算。然而，无序机制允许CPU提前递增循环计数器，使得它可以在错误预测之前检测到错误预测
迟到了。因此，你不应该太担心这种情况。

通常，无序执行机制是自动工作的。然而，程序员可以做一些事情来最大限度地利用无序执行。最重要的是避免长依赖链。你可以做的另一件事是混合不同类型的操作，以便在CPU的不同执行单元之间平均分配工作。只要不需要整数和浮点计算之间的转换，混合整数和浮点计算可能是有利的
点数。将浮点加法与浮点乘法混合，将简单整数与向量整数运算混合，以及将数学计算与存储器访问混合也是有利的。

太长的依赖链会给CPU的无序资源带来压力，即使它们不会进入循环的下一次迭代。现代CPU通常可以处理更多

超过一百个待定操作（参见手册3：“英特尔、AMD和VIA CPU的微架构”）。为了打破超长的依赖链，将一个循环一分为二并存储中间结果可能很有用。

12使用向量运算

今天的微处理器有向量指令，可以同时对向量的所有元素进行运算。这也称为单指令多数据（SIMD）操作。每个向量的总大小可以是64位(MMX)、128位(XMM)、256位(YMM)和512位(ZMM)。

当对大型数据集进行计算时，向量运算是有用的，其中对多个数据元素执行相同的运算，并且程序逻辑允许并行
计算。例子有图像处理、声音处理和数学
向量和矩阵的运算。本质上是串行的算法，例如大多数
排序算法不太适合向量运算。严重依赖表查找或需要大量数据置换的算法，如许多加密算法，可能不太适合向量运算。

向量运算使用一组特殊的向量寄存器。每个的最大大小
如果SSE2指令集可用，向量寄存器为128位(XMM)；如果AVX512指令集可用，向量寄存器为256位(YMM)；如果AVX512指令集可用，向量寄存器为512位
可用。每个向量中的元素数取决于数据元素的大小和类型，如下所示：

元素类型	每个元素的大小，比特	元素数量	向量的总大小，位	指令集
查尔	8	8	64	MMX
短int	16	4	64	MMX
int	32	2	64	MMX
int64_t	64	1	64	MMX
查尔	8	16	128	SSE2
短int	16	8	128	SSE2
int	32	4	128	SSE2
int64_t	64	2	128	SSE2
_浮动16	16	8	128	AVX512FP16
浮动	32	4	128	SSE
double	64	2	128	SSE2
查尔	8	32	256	AVX2
短int	16	16	256	AVX2
int	32	8	256	AVX2
int64_t	64	4	256	AVX2
_浮动16	16	16	256	AVX512FP16
浮动	32	8	256	AVX
double	64	4	256	AVX
查尔	8	64	512	AVX512BW
短int	16	32	512	AVX512BW
int	32	16	512	AVX512F
int64_t	64	8	512	AVX512F
_浮动16	16	32	512	AVX512FP16
浮动	32	16	512	AVX512F
double	64	8	512	AVX512F

表12.1。不同指令集扩展中可用的向量大小

例如，当SSE2指令setis可用时，一个128位的XMM寄存器可以被组织为一个包含8个16位整数或4个浮点数的集合集合。旧的MMX寄存器
是64位宽，应该避免使用，因为它们不能使用x87风格的浮动点码。

这个128位的XMM向量必须由16个来对齐，即存储在一个内存地址上，即为
当编译一个低于AVX的指令时，可被16整除（见下文）。第256位YMM向量最好由32对齐，512位ZMM寄存器由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。

向量操作在较新的处理器上特别快。许多过程扫描计算的向量的速度和标量一样快（标量表示单个元素）。的第一代
支持新向量大小的处理器有时具有执行单元、存储器端口等。只有最大向量的一半大小。这些单位用于处理全尺寸向量两次。

数据元素越小，向量运算的使用越有利。例如，您可以在完成8个浮点加法所需的时间内获得16个浮点加法
双倍加法。如果数据很好地适合向量寄存器，在当代CPU上使用向量运算几乎总是有利的。

12.1 AVX指令集和YMM寄存器

在AVX中，128位XMM寄存器扩展为256位YMM寄存器
指令集。AVX指令集的主要优点是它允许更大的浮点向量。还有其他优点可以提高性能
有点。AVX2指令集还允许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。不支持AVX的旧操作系统是
今天很少使用，所以这不是问题。微软、英特尔、Gnu和Clang的最新编译器都支持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。从AVX代码转到非AVX代码时会产生性能损失
因为YMM寄存器状态改变。应该通过在从AVX代码到非AVX代码的任何转换之前调用intrinsic function_mm256_zeroupper()来避免这种损失。在以下情况下可能需要这样做：

·如果程序的一部分在AVX支持下编译，而程序的另一部分在没有AVX支持下编译，则在离开AVXpart之前调用_mm256_zeroupper()。

•lfa函数在有和没有AVX的情况下使用CPU调度编译成多个版本，然后在离开AVXpart之前调用_mm256_zeroupper()。

如果使用AVX支持编译的代码调用编译器附带的库以外的库中的函数，并且库不支持AVX，那么在调用库函数之前使用_mm256_zeroupper（）。

编译器可能会或也可能不会自动进行insert_mm256_zeroupper（）。编译器的汇编输出将告诉什么。

12.2AVX512指令集和ZMM寄存器

该2个56位的YMM寄存器在AVX512中被扩展到名为ZMM的5个12位的寄存器
指令组在64位模式下从16扩展到32。
在32位模式中只有8个向量寄存器。因此，AVX512代码最好为64位模式编译。

AVX512指令集还添加了一组掩寄存器。这些被用作布载体。几乎任何向量指令都可以被掩码寄存器掩蔽，以便每个
只有当对应的b和掩码寄存器为1时才能计算出向量元素。这使得带有分支的代码的向量化更加有效。

AVX512还有许多额外的扩展。使用AVX512的小处理器有一些这样的扩展，但到目前为止还没有一个处理器有所有的扩展。AVX512的一些已知的和平面的扩展是：

- AVX512F.基金会所有AVX512 p都可以。包括对32位和64位整数的运算，浮点运算和对512位向量的双重运算，包括m个要求
  操作
- AVX512VL.包括对128位和256位向量进行相同的操作，包括掩码操作和32个向量寄存器。
- AVX512BW.对5个12位向量中的8位和16位整数的操作。
- AVX512DQ.使用64位整数的乘法和转换指令。关于浮动和双重操作的各种说明。
- AVX512ER.快速倒数，倒数平方根，指数函数。
  精确的浮动；近似的加倍。仅在few处理器上可用。

• AVX512CD.冲突的检测。在向量中找到重复的元素。
• AVX512PF.使用收集/散点逻辑进行预取指令。
•AVX512VBMI.8位粒度的置换和移位。
•AVX512VBMI2。用8位和16位整数压缩和扩展稀疏向量。有用的文本处理。
•AVX512IFMA.52位整数上的融合乘法和加法。
·AVX512_4VNNIW。16位整数上的迭代点积。
•AVX512_4FMAPS.迭代融合乘法和加法，单精度。
·AVX512_FP16。半精度浮点数的向量。

这使得CPU调度更加复杂。您可以选择以下扩展
对特定任务有用，并为具有此扩展的处理器创建代码分支。

在AVX512代码中，_mm256_zeroupper()的使用不太重要，但仍然
推荐。参见手册2：“用汇编语言优化子程序”第13.2章和手册5：“调用约定”第6.3章。

12.3自动矢量化

在并行性明显的情况下，好的编译器可以自动使用向量运算。有关详细说明，请参见编译器文档。示例：

```
//示例12.1 a。自动矢量化
const int size=1024;
int a【大小】，b【大小】；
//...
对于（int i=0；i<size；i++）
   {a[i]=b[i]+2；
}
```

当theSSE2或更高版本时，一个好的编译器将通过使用向量操作来优化thisloop
指令集已指定。代码会将b的4个、8个或16个元素读入a
向量寄存器取决于指令集，并与另一个向量进行加法
寄存器包含（2，2，2，...），并将这四个结果存储在一个。
重复数组大小除以向量的元素数。
s peedis相应地改进了。最好是循环计数能被每个向量的rof元素数整除。您可以在数组末尾添加虚拟元素，使数组大小成为向量大小的倍数。

当通过指针访问数组时存在缺点，例如：

```
//示例12.1 b.带对准问题的矢量化
2（<限制aa，<限制bb）{为（int i=0；i<大小；i++）{
      aa[i]= bb[i]+2;}
}
```

数组的性能由v ectorsize，i.e.16,32或64对齐
分别为XMM、YMM和ZMM寄存器。在指令设置到AVX下，有效的向量操作要求数组被16可整除的地址对齐。在
样例12.1a，编译器可以根据需要对齐数组，但在例子中12.1b
编译器无法确定数组是否正确对齐。循环仍然可以向量化，但是代码将降低效率，因为编译器必须采取外部预防措施来考虑未对齐的数组。当通过指针或引用访问数组时，您可以做很多事情来使代码更有效：

如果使用了Intel编译器，则使用#pragma向量对齐或
__assume_aligneddirective告诉编译器数组是对齐的，并确保它们是。

内联地声明该函数。这可能使编译器能够减少示例12.1 b至12.1a.

尽可能启用向量大小最大的指令集。AVX和横向指令对对齐的限制很少，生成的代码无论数组对齐还是不有效。

如果满足以下条件，自动矢量化效果最好：

1.使用具有良好支持自动向量化的编译器，如asGnu、Clang或Intel。

2.请使用最新版本的编译器。编译器正在变得更好、更好地实现向量化。

3.如果通过指针或引用访问数组或结构，则告知
编译器显式地说，指针不别名，如果合适，使用__限制o r__restrict__keyword。

4.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、-mavx512f等。用于Linux）

5.使用限制较少的floatingpoint选项。对于Gnu和Clan gcompilers，使用选项-O2-fno-trapping-math-fno-math-errno-fno-signed-zeros（-ffast-mathworks也可以，但是像isnan(x)这样的函数在
-快速数学）。

6.对于SSE2，将阵列和大结构按16对齐，对于AVX，优选地按32对齐，并且优选地64用于AVX512。

7.循环计数最好是可被向量中的元素数整除的常数。

8.如果数组是通过指针访问的，因此在您想要矢量化的函数的作用域中对齐不可见，那么请遵循上面给出的建议。

9.在向量元素级别尽量减少分支的使用。

10.避免在向量元素级别进行tablelookup。

您可以查看程序集输出列表，以了解代码是否确实按照预期进行了矢量化（请参见88).

编译器可能无法将代码矢量化，或者由于多种原因使代码变得不必要的复杂。列出了自动矢量化最重要的障碍
这里：

•编译器不能排除数据指针指向重叠或混叠的地址。

•编译器不能排除not-takenbranches会产生异常或其他副作用。

·编译器不知道数组的大小是否是vectorsize的倍数。

·编译器不知道数据结构是否正确对齐。

·数据需要重新排列以适应向量。

•代码过于复杂。

•代码调用在Vector版本中不可用的外部函数。

•代码使用查找表。

如果对一系列连续变量执行相同的操作，编译器也可以使用向量操作，其中没有循环。示例：

```
//示例12.2
结构对齐（16）S1/4个浮动的结构，对齐
    浮动a、b、c、d;
};

空白函数（）{
    S1x,y;
    ...
    x.a=y.a+1.;
    x.b=y.b+2.;
    x.c=y.c+3.;
    x.d=y.d+4.;
};
```

四个浮点子的结构适合一个128位XMM寄存器。示例12.2，优化后的代码将把结构加载到一个向量寄存器中，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是否会进行向量化
是否有利。编译器可能有#实用器或其他指令，可以用来告诉编译器哪个循环向量化。实用必须放置位置
在循环或希望它们应用的一系列语句之前。

如果代码包含分支，则会出现并发症，如示例中12.4abelow.处理分支的方法是计算一个分支的两边，然后合并这两个结果
使一些向量元素从一个分支和其他元素中获得
都是取自另一个分支机构的。编译器将不会在这个案例中向量化代码，如果理论上有可能一个未被占用的分支将为一个异常生成一个陷阱
或有其他副作用。您可以通过指定编译器选项、捕获数学和捕获数学编译器来帮助编译器对代码进行向量化
Gnu和Clang编译器，或类似的其他编译器。它不是
建议使用选择快速数学，因为它将禁用无穷大和NAN的检测(见页66).

使用适合应用程序的最小数据大小是有利的。例如，在考试ple12.3a中，您可以使用短int代替int来提高速度。短的
int是16位宽，而int是32位，所以你可以有8个类型
短in一个向量，而你只能有四个类型int。因此，使用的最小皮肤尺寸大到足以保持数字

存在问题，但不会产生溢出。同样地，如果代码可以向量化，则有利于使用浮点使用，而不是双重使用，因为浮点使用32位，而双重使用
64位。如果您需要大量的提取码来在不同的精度之间进行转换，那么使用较低的精度的优势可能会消失。

SSE2向量指令集不能使用除短int（16位）以外的任何尺寸的多重整数。没有关于整数除法调用器的指令，但是向量类库(参见页面有一个关于整数向量除法的函数。

12.4固有函数

很难预测编译器是否会向量化循环。以下
示例显示了编译器可能或可能不自动矢量化的代码。代码有一个分支，为数组中的每个元素在两个表达式之间进行选择：

```
//示例12.4 a。带分支的循环

//带分支的循环
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]); }
}
```

可以通过使用所谓的内在函数显式地向量化代码。这是
在类似情况下有用示例12.4 a其中当前的编译器并不总是自动向量化代码。它也适用于自动矢量化导致
次优代码。

内在函数在某种程度上类似于汇编编程，因为大多数
内在函数调用被翻译成特定的机器指令。内在函数比汇编编程更容易、更安全，因为编译器负责
寄存器分配、函数调用约定等。另一个优点是编译器可能能够通过重新排序指令、公共子表达式消除等进一步优化代码。Gnu、Clang、Intel和Microsoft都支持固有函数
编译器。使用Gnu和Clang编译器可以获得最佳性能。

我们想把例子中的循环矢量化12.4 a，这样我们就可以在八个16位整数的向量中处理八个元素ATA时间。循环中的分支可以在
各种方式取决于可用指令集。传统的方法是当bb[i]>0时用all1制作比特掩码，当bb[i]>0时用all0制作比特掩码。用这个掩码对cc[i]+2的值进行AND运算，用反向掩码对bb[i]*cc[i]进行AND运算。The
表达式thatis AND'ed与所有1是不变的，而表达式t hatis AND'ed与所有0给出零。这两者的或组合给出所选择的表达。

示例12.4 b显示了如何使用SSE2指令集的固有功能来实现这一点：

```
//示例12.4 b.用SSE2矢量化
#include<emmintrin.h>//定义SSE2固有函数

//从数组加载未对齐整数向量的函数
静态内联m128 i LoadVector(void const*p){
   return_mm_loadu_si128((m128i const*)p);
}

//将未对齐的整数向量存储到arr ay中的函数
静态内联void StoreVector(void*d,m128 i const&x){
```

```
    _mm_storeu_si128((m128i*)d, x); }

//分支/循环函数矢量化:
void SelectAddMul(short int aa[],short int bb[],short int cc[]){

    //制作(0,0,0,0,0,0,0,0,0,0)的向量
    __m128 i zero=_mm_setzero_Si128();

    //生成一个向量为(2、2、2、2、2、2、2、2、2)
    __m128 i两个=_mm_set1_第16(2);

    //推出8个循环，以适应8个元素的向量:
    对于(int i=0; i<256; i+=8){

        //从bb连续加载8个元素到向量器中:
        __m128 i b=负载向量(bb+i);

        //从cc中加载8个连续的元素到向量c中:
        __m128 i c=负载向量(cc+i);

        //向向量c中的每个元素添加2
        __m128ic2=_mm_add_epi16(c, 2);

        //乘以b和c
        __m128 i和=_mm_mullo_epi16(b, c);

        //将b中的每个元素与0进行比较，并生成一个位掩码:
        __m128 i掩码=_mm_cmpgt_epi16(b, 零);

        //和向量c2中的每个元素:
        c2=_ mm_和_si128(c2, 掩模);

        //和向量bc中的每个元素:
        =_mm_andnot_si128(面具, 公元前);

        //或两个和操作的结果:
        __m128 i a=_ mm_或_si128(c2, 公元前);

        //将结果向量存储在aa中的8个连续元素中: 存储向量(aa+i, a);
    }
}
```

生成的代码将提高效率，因为每个时间中有8个元素
因为它避免了循环内部的分支。样例12.4b执行三到s甚至比例子快12.4a，这取决于环状体内部的分支的可预测性。

类型m128i定义了一个包含整数的128位向量。它可以包含其中的任何一个16位整数8位，8个整数16位，4个整数32位，或2个整数64位。类型m128定义了一个128位的变量或四浮点。类型m128d定义了一个两个双倍的128位向量。对应的256位向量寄存器被称为m256i、m256和m256d。512位寄存器被召回了m512i、m512和m512d。_ ——— —

内在向量的函数以_mm开头。这些功能列在来自Intel的编程手册中："Intel64和IA-32架构软件开发人员手册"。它们也可以通过方便的搜索页面找到英特尔内部指南。有成千上万个不同的内在函数，而且很难找到正确的功能
为特定目的而提供的功能。

示例中的与或构造如果theSSE4.1instruction集可用，可以替换12.4b：

```
//示例12.4c.同样的例子，用SSE4.1进行向量化

//函数加载从数组中未对齐的整数向量
静态内联m128 i负载向量（无效常量*p）{
    return_ mm_ loadu_ si128（（m128 i const*）p）；
}

//函数存储未对齐的整数向量到arr ay
静态内联空洞存储向量器（void*d，m128 i const&x）{_ mm_ storeu_ si128
    （（m128i*）d，x）；
}

空白选择AddMul（短小aa[]，短小bb[]，短小cc[]）{

    //生成（0、0、0、0、0、0、0、0、0）的向量
    __m128 i零=_mm_setzero_si128（）；
    //生成一个向量为（2、2、2、2、2、2、2、2、2）
    __m128 i两个=_mm_set1_第16(2)；

    //推出8个循环，以适应8个元素的向量：
    对于（int i=0；i<256；i+=8）{

        //从bb连续加载8个元素到向量器中：
        __m128 i b=负载向量（bb+i）；

        //从cc中加载8个连续的元素到向量c中：
        __m128 i c=负载向量（cc+i）；

        //向向量c中的每个元素添加2
        __m128ic2=_mm_add_epi16（c，2）；

        //乘以b和c
        __m128 i和=_mm_mullo_epi16（b，c）；

        //将b中的每个元素与0进行比较，并生成一个位掩码：
        __m128 i掩码=_mm_cmpgt_epi16（b，零）；

        //使用掩模为每个元素在c2和bc之间进行选择
        __M128IA=_MM_BLENDV_EPI8(bc，c2，mask)；

        //将结果向量存储在aa中的八个连续元素中：
        StoreVector(aa+1，a)；}
}
```

AVX512指令集提供了一种更有效的分支方式，使用ma s kregister和条件指令。可以将计算和分支合二为一

说明：

```
//示例12.4 d.相同示例，用AVX512F矢量化

void SelectAddMul(short int aa[],short int bb[],short int cc[]){

    //制作all0的向量
    __m512 i零=_MM512_Setzero_SI512()；

    //制作all2的向量
    __m512 i two=_MM512_SET1_EPI16(2)；
```

```
//展开循环by32以拟合32元素向量:
for(int i=0;i<256;i+=32){
```

```
            //将32个连续元素从bb加载到向量b中：
            __M5121B=_MM512_LOADU_SI512(bb+1);

            //将32个连续元素从cc加载到向量c中：
            __M512IC=_MM512_LOADU_SI512(cc+1);

            //将b和c相乘
            __M5121BC=_MM512_MULLO_EPI16(b, c);

            //将b中的每个元素与0进行比较，并生成32位掩码//对于b[i]>0，掩码中
            的每个位都是1
            __mmask32 mask=_MM512_CMP_EPI16_MASK(b, 零, 6);

            //有条件地将2添加到向量c中的每个元素。
            //当掩码位=1时选择c+2，当掩码位=0时选择bc
            __M512IR=_MM512_MASK_ADD_EPI16(bc, mask, c, two);

            //将结果向量存储在aa中的32个元素中：
            _MM512_STOREU_EPI16(aa+i, r); }
}
```

您必须为您正在编译的指令集包含适当的头文件。头文件immintrin.hor x86intrin.h涵盖所有指令集。

你必须确保CPU支持相应的指令集。The
如果程序包含CPU不支持的指令，则该程序将崩溃。除了微软以外，所有的公司都允许您在命令行上指定您要编译的指令。

请参见第页135关于如何检查CPU支持哪些指令集。

对齐数据

如果数据对齐到一个可按向量大小（16、32或64字节）整除的地址，那么将数据加载到向量的速度就会更快。这对旧处理器和英特尔Atom处理器有显著影响，但对大多数新处理器却不那么重要。以下内容
这个例子显示了如何排列叙述。

```
// Example12.5.对齐阵列
常量int大小=256; //数组大小
对齐（16）int16_taa[size]; //制作对齐数组
```

矢量表查找

查找表对于优化代码很有用，如页面上所述144.不幸的是，表查找通常是向量化的一个障碍。AVX2和更高版本的指令集都有
收集对表格查找有用的说明。32位或64位整数的向量
将索引提供到内存中的表中。结果是32位或64位向量
整数、浮点或双精度值。收集指令需要几个时钟周期，因为元素被一个接一个地读取。

如果查找表很小，可以放入一个或几个向量寄存器，那么使用Permute指令进行表查找会更有效。AVX指令集允许
256位向量中的32位浮点数的排列。使用16位整数的排列需要AVX512BWin指令集。在超过16位的表中使用8位整数的排列
elements需要AVX512VBMI指令集。

使用内在函数可能相当乏味。代码变得庞大且难以阅读。使用向量类通常更容易，正如下一节所解释的。

12.5使用向量类

以实例的方式用本征函数编程　　12.4b，这确实很乏味。通过将向量包装到C-++类中，并对附加向量等内容使用重载操作符，可以以一种更清晰易懂的方式编写相同的内容。操作符排列，使生成的机器编码与您拥有的一样
使用固有函数。写一个+b比写一个write_mm512_add_epi16（a，b
）更容易。

向量类的优点是：

您可以显式地指定要向量化的代码中的哪些部分，以及如何指定

您可以克服页面上列出的自动矢量化的障碍119

代码通常比自动程序的操作更简单
因为编译器必须处理可能与您的情况无关的特殊情况

该代码比汇编代码或内在函数更简单、更易读，而且效率非常高

目前有各种预定义向量类的库可用，包括一个来自Intel的库和一个来自Mme的库。我的向量类库（VCL）有许多特性，请参见
https://github.com/vectorcl屁股。英特尔向量类库的功能很少，也很少更新。

向量类库	英特尔	VCL（Agner）
可从	英特尔和MicrosoftC++编译器	https://github.com/vectorclass
包含文件	dvec.h	矢量类
支持的编译器	英特尔、微软	英特尔、微软、Gnu、Clang
支持的操作系统	Windows、Linux、Mac	Windows、Linux、Mac、BSD
指令集控制	没有	是的
许可证	编译器价格中包含的许可证	Apache 2.0

表12.2。向量类库

下表列出了可用的向量类。包含适当的头文件将使您能够访问所有这些类。

每个元素的大小，比特	向量中的元素数	元素类型	向量的总大小，位	矢量类，英特尔	矢量类，VCL
8	8	int8_t	64	Is8vec8	
8	8	uint8_t	64	Iu8vec8	
16	4	int16_t	64	Is16vec4	
16	4	uint16_t	64	Iu16vec4	
32	2	int32_t	64	Is32vec2	
32	2	uint32_t	64	Iu32vec2	
64	1	int64_t	64	I64vec1	
8	16	int8_t	128	Is8vec16	Vec16c
8	16	uint8_t	128	Iu8vec16	Vec16uc
16	8	int16_t	128	Is16vec8	Vec8s
16	8	uint16_t	128	Iu16vec8	Vec8us
32	4	int32_t	128	Is32vec4	Vec4i
32	4	uint32_t	128	Iu32vec4	Vec4ui
64	2	int64_t	128	I64vec2	Vec2q
64	2	uint64_t	128		Vec2uq
8	32	int8_t	256		Vec32c
8	32	uint8_t	256		Vec32uc
16	16	int16_t	256		Vec16s
16	16	uint16_t	256		Vec16us
32	8	int32_t	256		Vec8i
32	8	uint32_t	256		Vec8ui
64	4	int64_t	256		Vec4q
64	4	uint64_t	256		Vec4uq
8	64	int8_t	512	I8vec64	Vec64c
8	64	uint8_t	512	Iu8vec64	Vec64uc
16	32	int16_t	512	I16vec32	Vec32s
16	32	uint16_t	512	Iu16vec32	Vec32us
32	16	int32_t	512	I32vec16	Vec16i
32	16	uint32_t	512	Iu32vec16	Vec16ui
64	8	int64_t	512	I64vec8	Vec8q
64	8	uint64_t	512	Iu64vec8	Vec8uq
16	8	浮动16	128		Vec8h
32	4	浮动	128	F32vec4	Vec4f
64	2	double	128	F64vec2	Vec2d
16	16	浮动16	256		Vec16h
32	8	浮动	256	F32vec8	Vec8f
64	4	double	256	F64vec4	Vec4d
16	32	浮动16	512		Vec32h
32	16	浮动	512	F32vec16	Vec16f
64	8	double	512	F64vec8	Vec8d

表12.3.在两个库中定义的向量类

不建议使用总大小为64位的向量，因为这些向量是
与浮点代码不兼容。如果删除64位向量，则必须在64位向量操作之后和任何浮点代码之前执行execute_mm_empty()。128位及以上的向量没有这个问题。

只有在CPU和
操作系统（参见第页117).VCL vector类库可以将256位向量仿真为两个128位向量，将512位向量仿真为两个256位向量或四个128位向量，

取决于指令集。只有启用AVX512-FP16指令集时，半精度浮点(_Float16)的向量才有效。

下面的示例显示了与示例相同的代码12.4 b，使用英特尔向量类重写：

```
//示例12.4 d.同样的示例，使用英特尔向量类
#include<dvec.h>//定义向量类

//函数加载从数组中未对齐的整数向量
静态内联m128 i负载向量（空白常量*p）
   {return_mm_loadu_si128（（m128iconst*）p）；）

//函数存储未对齐的整数向量到arr ay
静态内联空洞存储向量（void*d，m128iconst&x）{_mm_storeu_si128（（m128
   i*）d，x）；}

空白选择AddMul（短小aa[]，短小bb[]，短小cc[]）{

   //生成（0、0、0、0、0、0、0、0、0）的向量
   为16vec8零（0、0、0、0、0、0、0、0）；
   //生成一个向量为（2、2、2、2、2、2、2、2、2）
   是16vec82（2、2、2、2、2、2、2、2）；

   //推出循环八，以适应八元素向量：
   对于（int i=0；i<256；i+=8）{
      //从bb连续加载8个元素到向量器中：
      Is16 vec8 b= LoadVector （ bb+i);
      //从cc将8个连续的元素加载到向量c中：
      Is16 vec8 c= LoadVector （ cc+i);
      // result= b>0?c+2: b*c;
      是16vec8a=select_gt（b，零，c+2，b*c）；
      //将结果向量存储在aa中的8个连续元素中：存储向量（aa+i，a）；
   }
}
```

使用VCL向量类的相同例子如下：

```
//示例12.4e.同样的例子，使用VCL
#包括"向量类。h"//定义向量类

void选择AddMul（短，短，短，短，短，短，cc）{//定义向量对象
   Vec16sa、b、c;

   //推出循环8，以适应八元素向量：
   对于（int i=0；i<256；i+=16）{

      //从bb连续加载8个元素到向量器中：
      b. 负载（bb+i）；

      //从cc中加载8个连续的元素到向量c中：
      c. 负载（cc+i）；

      // result= b>0?c+2: b*c;
      a=select (b>0, c+2, b*c);

                                    //将结果向量存储在aa中的8个连续元素中：
      a. 存储（aa+i）
   ；}
}
```

向量类库（VCL）包括许多特征，如向量置换和数学函数。它还包括用于各种目的的附加程序包。请参见详情请访问GitHub网站。

CPU调度与向量类

VCL向量类库可以从相同的源代码中编译不同的指令集。该图书馆有选择最好的预处理指令
为一个授予指令集的实现。

下面的示例显示了要进行选择添加模拟示例（12.4e）与
自动CPU调度。本示例中的代码应该被编译四次，一次
对于SSE2指令集，一个用于SSE4.1，一个用于AVX2，另一个用于AVX512 BW。所有有四个版本被链接到同一个可执行文件中。SSE2是支持的最小值
向量类库的指令集，SSE4.1在选择函数上具有优势，AVX2指令集具有更大的向量寄存器，AVX512具有更大的向量和更有效的分支。

向量类库将使用e512位向量寄存器，或两个256位向量寄存器，或四个128位向量寄存器，这取决于指令集。

预处理宏指令集用于给函数一个不同的名称
each指令集。CPU调度程序在第一次调用函数时设置函数的最佳版本指针。请参见详细信息的矢量类手册。

```
// Example12.6.具有自动CPU调度#的向量类代码包括
"vectorclass.h"//向量类库
#包括<stdio.h>//定义打印格式

//定义函数类型
类型定义无效函数类型（短小aa，短小bb，短小cc）；

每个版本的//函数原型
FuncType SelectAddMul、selectaddmul_sse2、selectaddmul_sse41、
SelectAddmul_AVX2、SelectAddmul_AVX512 BW、SelectAddmul_Dispatch;

//根据指令集定义函数名
#如果INSTRSET==2//SSE2
#define FUNCNAMESelectAddMul_SSE2
#elif INSTRSET==5//SSE4.1
#define funcname selectaddmul_sse41
#elif INSTRSET==8//AVX2
#define FUNCNAME SelectAddMul_AVX2
#elif INSTRSET==10//AVX512 BW
#define funcname selectaddmul_avx512 bw
#endif

//函数的特定版本。每个版本编译一次
void FUNCNAME(short int aa[],short int bb[],short int cc[]){Vec32
   s a,b,c;//定义最大可能的向量对象
   //展开循环by32以适应最大的向量：
   for(int i=0;i<256;i+=32){
      b.负载（bb+i）；
      c.负载（cc+i）；
      a=选择(b>0，c+2，b*c)；
      a.存储(aa+i)；}
}

#如果INSTRSET==2
//使dispatcher仅在最低的编译版本中
```

```
#include“instrset_detect.cpp”//instrset_detect函数
```

```
//函数指针最初指向Dispatcher。
//第一次调用后，它指向所选版本
FuncType*selectaddmul_pointer=&selectaddmul_dispatch;

//调度程序
voidSelectAddMul_dispatch(
    短int aa[]、短int bb[]、短int cc[]){

    //检测支持的指令集
    int iset=instrset_detect();

    //设置函数指针
    如果(iset>=10)selectaddmul_pointer=&selectaddmul_avx512 bw;
    否则如果(iset>=8)selectaddmul_pointer=&selectaddmul_avx2;
    否则如果(iset>=5)selectaddmul_pointer=&selectaddmul_sse41;
    否则如果(iset>=2)selectaddmul_pointer=&selectaddmul_sse2;
    否则{
        //错误：不支持最低指令集
        fprintf(stderr，"\nError：不支持低于SSE2的指令集");
        返回；}
    //在分派版本中继续
    return(*selectaddmul_pointer)(aa，bb，cc)；}

//进入分派函数调用
内联voidSelectAddMul(
    短int aa[]、短int bb[]、短int cc[]){
    //转到调度版本
    return(*selectaddmul_pointer)(aa，bb，cc)；}

#endif//INSTRSET==2
```

12.6变换串行代码进行矢量化

不是所有的代码都有一个并行结构，可以很容易地组织成向量。大量代码是串行的，因为每个计算都依赖于前一个。然而，如果代码是
重复。最简单的情况是一长串数字的和：

```
//example12.7 a.列表的和
浮点A[100]；
浮点和=0；
对于（int i=0；i<100；i++）sum+=a[i]；
```

上面代码是串行的，因为sum的每个值都取决于
和。诀窍是展开loopby n并重新组织代码，以便每个值
取决于它返回的值，其中nis是avector中元素的数量。如果n=4，我们有：

```
//示例12.7 b.列表的总和，滚动4
浮点A[100]；
floats0=0，s1=0，s2=0，s3=0，sum；
对于（int i=0；i<100；i+=4）{s0+=a[i]；
    s1+=A[i+1]；
    s2+=A[i+2]；
    s3+=A[i+3]；
```

```
}
总和=(s0+s1)+(s2+s3);
```

现在S0、s1、s2和S3可以组合成一个128位向量，这样我们就可以做四个
一次操作中的加法。一个好的编译器将转换示例12.7 a至12.7 B
如果我们为fastmath和SSE或更高指令集指定选项，则自动并矢量化代码。

更复杂的情况不能自动向量化。例如，让我们来看看一个泰勒级数的例子。指数函数可以通过以下方法计算：

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

一个C-++实现可能是这样的：

```
// Example12.8a.泰勒系列 泰勒级数
浮动Exp (浮动x) 对于小x的近似exp (x)
    浮动xn=x; //x^n
    浮动和=1.f; //和，初始化为x^0/0!
    浮子nfac=1.f; //n阶乘
    对于（int n=1; n<=16; n++）
        {sum+=xn/nfac;
        xn*=x;
        nfac*=n+1;}
    返还金额;
```

这里，每个值xn根据前面的值x n=x.xn-1计算，每个n！的值根据之前的值n！=n.(n-1)!.如果我们想推出这个循环，我们将必须从四个位回的值中计算每个值。因此，我们将
计算xn为x4.xn-4。没有简单的方法来推广系数的计算，但这是不必要的，因为系数不依赖于x索元存储在预计算表中的值。更好的是：存储互惠系数，这样我们就不必做划分（你知道，分区是缓慢的）。代码现在可以向量化如下（使用向量类库）：

```
//示例12.8b.泰勒级数，向量化
#包括"向量类。h"//vector类库

//对于小的x的近似exp (x)
浮子Exp（浮子x）{
    //表1/n!
    排列（16）常浮动coef[16]={
        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};
    floatx2= x*x;//x^2
    浮动x4=x2*x2; //x^4
    //定义四个浮点数的向量
    Vec4 f xxn (x4,x2*x,x2,x);//x^1,x^2,x^3,x^4
    VEC4F XX4(x4); //x^4
    VEC4F S(0.f, 0.f, 0.f, 1.f); //初始化和
    for(int i=0;i<16;i+=4){//循环by4
        s+=xxn*Vec4f()。负荷(coef+i); //s+=x^n/n!
        xxn*=xx4; //接下来的四个x^n
    }
    返回horizontal_add(s); //将四个和相加
}
```

这个循环计算一个向量中的四个连续项。如果循环很长，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乘法的延迟而不是吞吐量的限制（参见p。113).系数表是在编译时计算的。在运行时计算表可能更方便，只要您确保它只计算一次，而不是每次调用函数。

12.7向量的数学函数

有各种函数库用于计算数学函数，例如
向量中的对数、指数函数、三角函数等。这些函数库对于数学代码的矢量化很有用。

有两种不同的向量数学库：长向量库和短向量库。为了解释这种差异，假设你想在一千个数字上计算同一个函数。有了along向量库，你就可以输入成千上万的
数字作为库函数的参数，函数将一千个结果存储在另一个数组中。使用向量库的缺点是如果你正在做一个
然后，在调用下一步的函数之前，您必须将序列中每一步的中间结果存储在一个临时数组中。用一个短的
向量库，您将数据集划分为适合CPU中向量寄存器大小的子向量。如果向量寄存器可以保持。八个数字，然后你必须打电话给
库函数125次，一次8个数字打包到一个向量寄存器中。The
库函数将在向量寄存器中返回结果，该寄存器可以直接馈送到计算序列中的下一步，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，这可能会更快，因为CPU可以计算，同时预取下一个函数的代码。然而，
如果计算序列形成长依赖链，短向量方法可能是一个缺点。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，并阻止它充分利用它-
订单计算能力。

以下是一些长向量数学库的列表：

- ·英特尔向量数学库（VML，MKL）。适用于所有x86平台。该库在非英特尔CPU上的性能不确定。参见第页143.
- •英特尔性能原语(IPP)。适用于所有x86平台。与
  非英特尔CPU。包括许多统计、信号处理和图像处理功能。
- •是的。开源库。支持X86和ARM平台以及各种编程语言（这仍然有效吗？）.

以下是短载体ma文库的列表：

- ·睡眠库。支持许多不同的平台。开源。www.睡眠。组织
- •英特尔短向量数学库(SVML)。这是英特尔编译器提供的，并通过自动矢量化调用。如果您没有使用英特尔编译器，此库通常可以很好地与非英特尔CPU配合使用。参见第页143.
- ·大多数编译器包括一个向量数学库，用于包含数学函数的循环的自动矢量化。参见下面的讨论。
- ·VCL载体类库。开源。支持所有x86平台。微软、英特尔、Gnu和Clang编译器。代码是内联的，因此不需要与外部库链接。https：//github。com/vectorclass

所有这些库都具有很好的性能和精度。速度比任何非矢量库都快很多倍。

向量数学库中的函数名并不总是有很好的文档记录。The
示例如果您想直接调用库函数，此表可能会有所帮助：

图书馆	4个浮点的exp函数	2 double的exp函数
英特尔SVML v.10.2及更早版本	vmlsExp4	vmldExp2
英特尔SVML v.10.3及更高版本	__svml_expf4	__svml_exp2
英特尔SVML+ia32intrin.h	_mm_exp_ps	_mm_exp_pd
AMD核心数学库	__vrs4_expf	__vrd2_exp
AMD LIBM库	amd_vrs4_expf	amd_vrd2_exp
Gnu libmvec	取决于指令集	
VCL向量类库	经验	经验

使用不同编译器的短向量mat库

包含数学函数的循环只有在有函数时才能矢量化
包含这些函数的向量版本的库，具有向量输入和向量输出。（sqrtfunction不需要外部库）
。

旧版本的Gnu编译器可以选择使用英特尔提供的名为SVML（短向量数学库）的外部库。现在
，Gnu编译器有了自己的向量
名为libmvec的数学库，如果需要，可以自动链接。

Clang编译器目前没有向量数学库。C语言版本12和更高版本有选项fveclib=libmvec，这使得
它使用Gnu向量数学库。

微软编译器（ve rsion17,2022）有自己的向量数学库，需要时会自动链接。

英特尔编译器的所有版本都包含高度优化的SVMLlibrary，它链接到
自动地。SVML库针对Intel处理器进行了优化，但它给出了令人满意的结果
在非英特尔处理器上的性能也是如此，除非它与英特尔编译器的传统版本（名为“经典”）一
起使用。

向量化数学函数的另一种解决方案是使用Vector类库（链接）
描述于第页125.Vector类库在vector中提供inlinemath函数
形式。它还包括英特尔SV MLlibrary的可选接口。这适用于上面提到的所有编译器。

Gnu libmvec库目前只包含最基本的函数，如assin、cos，
功率，日志，经验。其他库还包含反三角函数、双曲函数
函数等。SVML库包含最多的数学函数。SVMLlibrary可以与英特尔以外的其他编译器一起使
用，但是调用接口的文档很少。SVML还包含未记录的函数，例如sinpi。

不同的库使用不同的计算方法，但它们都给出了相当好的准确性。

很难比较不同向量数学库的性能。所有vectormath库在数据位于
合理的范围，但它们在处理极端数据值和特殊情况的效率方面有很大不同。libmvec库在特殊
情况下会退回到标量函数，以确保即使是最罕见的情况也能正确处理。微软和英特尔的VML
图书馆也使用大型分支机构来处理特殊情况。只有Vector类库在主程序流中处理特殊情况
，而不进行分支。这给出了优越的
所有特殊情况下的性能，但代码可能会过早溢出或下溢到极限输入值。

为了确定哪个向量数学库对于特定目的是最有效的，您不仅需要考虑简单数据值的性能，还需要考虑它如何处理数据值，如INF、NAN、次正规数和极值（如果出现这些值）通常在应用程序中。

12.8对齐动态分配的内存

用平台上8或16个dep结尾对齐的newor mallocis分配的内存。当需要16或更多对齐时，这是向量操作的一个问题。使用operatornew时，C++17标准会自动为您提供所需的对齐：

```
//示例12.9
//C++17中的对齐内存分配
int arraysize=100;
__m512*pp=新m512[arraysize];
//pp将通过alignof(m512)对齐，如果使用C++17
```

12.9对齐RGB视频或三维矢量

RGBimage数据每个点有三个值。这不符合g的向量。四彩车。这同样适用于三维几何和其他奇数大小的矢量数据。为了提高效率，数据必须通过向量大小对齐。使用unalignedreads和writes可能会将执行速度减慢到使用向量操作不太有利的程度。您可以选择以下解决方案之一，具体取决于最适合所讨论算法的解决方案：

·Putin未使用的第四个值，以使数据适合向量。这是一个简单的解决方案，但它增加了使用的内存量。如果内存访问是一个瓶颈，您可以避免这种方法。

·将数据组织成四个（或八个）点的组，其中四个R值在onevector中，四个G值在nextvector中，四个B值在lastvector中。

•首先用所有theR值组织数据，然后是所有Gvalues，最后是所有Bvalues。

选择使用哪种方法取决于什么最适合有问题的算法。您可以选择给出最简单代码的方法。

如果点数不能被向量大小整除，则在endin以获得整数个向量。

12.10结论

如果算法允许并行，使用矢量码可以大大提高速度计算。增益取决于每个向量的元素数。最简单的和最干净的解决方案是依靠编译器的自动矢量化。在并行性明显且代码仅包含简单标准操作的简单情况下，编译器将自动将代码矢量化。您所要做的就是启用适当的指令集和相关编译器选项。

然而，在许多情况下，编译器无法将代码矢量化自动地或以次优的方式这样做。这里你必须显式地矢量化代码。有多种方法可以做到这一点：

•使用汇编语言

•使用固有功能

·使用预定义的向量类

显式向量化代码的最简单方法是使用vector类库。你可以
如果您需要向量中未定义的内容，请将其与内在函数结合起来
类库。无论你选择使用内在函数还是向量类只是一个方便的问题——性能通常没有区别。内在函数有很长的名字，看起来笨拙而乏味。当您使用向量类和重载运算符时，代码变得更具可读性。

一个好的编译器通常能够在你将代码向量化后进一步优化它
手动。编译器可以使用优化技术，如函数内联、公共子表达式消除、常量传播、循环优化等。这些技术很少在手工汇编编码中使用，因为它们会使代码变得笨拙、错误
俯卧，几乎不可能维护。因此，手动矢量化与编译器的进一步优化相结合可以在困难的
情况下给出最佳结果。一些实验可能有助于找到最佳解决方案。你可以看看
程序集输出或调试器中的反汇编显示，以检查编译器正在做什么。

矢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其放置在向量中的正确位置。这种数据转换和
排列有时会比实际计算花费更多的时间。这应该是
在决定使用矢量化代码是否有利可图时要考虑到这一点。VCL Vector类库有一些非常有用的排列函数，可以自动找到特定排列模式的最佳实现。

我将通过总结决定矢量化优势的因素来结束本节。

有利于矢量化的因素：

·小型数据类型：char、int16_t、float。
•对largearrays中的所有数据进行类似操作。
·数组大小可被vectorsize整除。
·在两个简单表达式之间进行选择的不可预测分支。
·仅适用于向量操作数的操作：最小、最大、
饱和加法、快速近似倒数、快速近似倒数平方根、RGB色差。
·矢量指令集可用，例如。AVX2、AVX-512
•数学向量函数库。
•使用Gnu或Clangcompiler。

使矢量化不太有利的因素：

•更大的数据类型：int64_t，double。
·未对齐的数据。
·需要额外的数据转换、排列、打包、解包。
●可预测的分支，在未选择时可以跳过大型表达式。
•编译器没有足够的关于指针对齐和别名的信息。
·用于适当类型向量的指令集中缺少的操作，例如SSE4.1之前的32位整数乘法和整数除法。
·执行单元小于向量寄存器rsize的旧CPU。

对于程序员来说，矢量化代码更难编写，因此错误更多
俯卧。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为不同指令集制作多个版本的关键代码

微处理器生产商不断向指令集添加新指令。这些新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1章提到的向量操作12.

如果代码是为特定指令集编译的，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，但与早期CPU不兼容。向后兼容指令集的顺序如下：

指令集	重要功能
80386	32位模式
SSE	128位浮点向量
SSE2	128位整数和双向量
SSE3	横加等。
SSSE3	更多的整数向量指令
SSE4.1	更多向量指令
SSE4.2	字符串搜索指令
AVX	256位浮点和双向量
AVX2	256位整数向量
FMA3	浮点乘加
AVX512F	512 bitinteger和floatin gpoint向量
AVX512BW、DQ、VL	更多512位向量指令
AVX512-FP16	半精度浮点数的向量

表13.1.指令集

有关指令集的更详细说明，请参见手册4：“说明”
将为AVX或更高版本编译的代码与不使用AVX编译的代码混合存在某些限制，如第页所述117.

使用最新指令集的缺点是与旧指令集的兼容性
微处理器丢失了。这个困境可以通过为不同的CPU制作多个版本来解决。这叫做CPUdispatching。例如，您可能希望制作一个利用AVX512指令集的版本，
另一个版本适用于只有AVX2指令集的CPU，以及一个通用版本，与没有这些指令集的旧微处理器兼容。该计划
应该自动检测CPU支持哪个指令集，并为关键的最里面的循环选择适当的子程序版本。

13.1 CPU调度策略

就开发、测试和维护而言，将一段代码制作成多个版本是相当昂贵的，每个版本都针对特定的CPU进行了仔细的优化和微调。对于在多个
应用程序，但并不总是针对特定于应用程序的代码。如果你考虑用CPU调度来制作高度优化的代码，那么如果可能的话，建议把它做成可用库的形式。这也使得测试和维护更易于管理。

我对CPU调度做了大量的研究，发现许多常见的程序使用了不合适的CPU调度方法
。

CPU调度最常见的陷阱是：

·针对当前处理器而非未来处理器进行优化。考虑使用CPUdispatching开发和发布函数库所需的时间。再加上应用程序程序员获得新版本库所需的时间。
再加上开发和销售使用更新函数库的应用程序所需的时间。再加上最终用户获得应用程序最新版本之前的时间。总而言之，往往需要数年时间才能
您的代码正在大多数最终用户的计算机上运行。此时，任何
您优化的处理器可能已经过时。程序员经常低估这个时间滞后。

•根据特定的处理器型号而不是处理器特性来思考。The
程序员通常会想"什么在处理器X上工作得最好？"而不是"什么
在具有此指令集的处理器上工作得最好？"。代码分支到的列表
每个处理器型号的使用时间都很长，而且很难维护。最终用户不太可能拥有该程序的最新版本。C PUdispatcher不应该查看CPUbrand名称和型号，而应该查看什么
指令集和其他功能。

·假设处理器型号形成逻辑序列。如果您知道处理器型号N支持特定的指令集，则不能假设__
modelN+1至少支持相同的指令集。你也不能向我保证N-1型是劣质的。编号较高的型号不一定较新。CPU系列和型号并不总是连续的，您不能
根据未知CPU的系列和型号对其进行任何假设。

•无法正确处理未知处理器。许多CPU调度程序是
设计为仅处理已知处理器。其他品牌或型号
编程时unknown通常会得到genericbranch，它是
性能最差的那个。我们必须记住，许多用户更喜欢在最新的CPUmodel上运行对速度至关重要的程序
很可能是一个在编程时未知的模型。CPU
dispatcher应该给未知品牌或型号的CPU最好的分支，如果它是一个与该分支兼容的指令集。共同的
借口说，"我们不支持过程或x"在这里根本不合适。提出了一种有根本缺陷的
CPU调度方法。

低估了保持CPU调度程序更新的成本。我们需要将代码微调到特定的CPU模型，然后认为您可以在后续的新模型进入市场时进行更新。但是微调，测试的成本，
验证和维护一个新的代码分支是如此之高，以至于它是不现实的，你可以这样做，每次新的处理器进入市场未来几年。
甚至大型软件公司往往无法保持他们的CPU调度程序的最新状态。更现实的目标是做一个新的胸罩，只有当一个新的指令集打开了重要的证明的可能性。

做了太多的分支机构。如果你正在做一些经过微调的分支机构_
特定的cpublads或特定模型，你很快就会得到很多占用代码空间且难以维护的分支。
任何特定的瓶颈或任何瓶颈
特别是在特定CPU模型中处理的指令
可能在一两年内无关。通常，只有两个就足够了
分支：一个适合最新的指令集和一个兼容5到10年的cpu。CPU市场发展得非常快
今天的新品牌将在明年成为主流。

正在忽视虚拟化。当CPUID指令肯定是真正的时候
表示一个已知的CPU模型。虚拟化在今天很不常见。A
虚拟处理器可能会减少核的数量，以便为同一台机器上的其他虚拟处理器保留资源。虚拟处理器可能是
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，我们可能会看到仿真处理器和FPGA软核，它们不对应于任何已知的硬件CPU。这些虚拟处理器可以有任何名称和型号
号码。我们唯一可以依赖的CPUID信息是功能信息，例如支持的指令集和缓存大小。

幸运的是，在大多数情况下，这些问题的解决方案非常简单：CPU
dispatcher应该只有几个分支，调度应该基于CPU支持的指令集，而不是它的品牌、家族和型号。

我见过许多CPU调度不佳的例子。例如，Mathcad（v.15.0）是
使用六年前版本的英特尔数学内核库(MKLv.7.2)。这个库有一个
不能以最佳方式处理当前CPU的CPU调度程序。某些任务的速度
在当前英特尔CPU上，当CPUID人为设置时，可以增加33%以上
换成了老奔腾4。原因是英特尔MKL库中的CPU调度程序依赖于CPU系列号，在旧的奔腾4上ch为15，而所有较新的英特尔
CPU的家族编号为6。当CPUID被操纵以伪造IntelPentium 4时，非IntelCPU的速度提高了一倍多。更糟糕的是，许多
软件产品无法通过处理器识别，因为这个品牌在软件开发时不太受欢迎。

不平等对待不同品牌CPU的CPU调度机制可能会成为严重的法律问题，您可以在我的博客。在这里，您还可以找到更多
不良CPUdispatching的示例。

显然，您应该只将CPU调度应用于程序中最关键的部分——最好是隔离到一个单独的函数库中。只有当指令集相互依赖时，才应该使用使整个程序处于多个版本的激进解决方案
不相容。与调用程序相比，具有定义良好的功能和定义良好的接口的函数库更易于管理，也更容易测试、维护和验证
调度分支分散在源文件中的任何地方的程序。

13.2特定型号调度

在某些情况下，特定的代码实现在
特定的处理器型号。你可以忽略这个问题，并假设下一个
处理器模型将工作得更好。如果问题太重要而不能忽视，那么解决方案是列出该代码版本性能较差的处理器模型的负面列表。列出代码版本的处理器型号并不是一个好主意
表现良好。RE asonis apositive list需要在每次新的
处理器出现在市场上。这样的列表几乎肯定会在
软件的关键时刻。另一方面，消极主义者不需要更新更好的处理器。每当一个处理器有一个特别是弱点或瓶颈，生产者很可能会试图解决问题，并使后续模型更好地工作。

请记住，大多数软件大多在进程上运行
在软件被编码时还很未知。如果该软件包含了哪个处理器模型来运行最先进的代码版本，那么它将运行一个劣质的
在它被编程时是未知的处理器上的版本。提供的
软件包含一个负列表的处理模型，以避免运行高级

然后，它将在编程时未知的所有新模型上运行高级版本。

13.3困难案例

在大多数情况下，可以基于关于支持的指令集、缓存大小等的CP UID信息来选择最佳分支。然而，在少数情况下，做同一件事有不同的方法，而CPUID指令没有给出关于哪种实现是最佳的必要信息。这些情况有时在汇编语言中处理。以下是一些示例：

·strlen函数。string length函数sca ns一个字节字符串来查找第一个字节零字节。一个好的实现方法是使用向量寄存器来测试16字节或更多的ata时间，然后使用BSF（位前向扫描）指令来定位16字节块内的零的第一个字节。一些较旧的CPU速度特别慢
这个位扫描指令的实现。
单独的strlen函数对其性能不满意
在CPU上使用慢位扫描指令，并实现了针对特定CPU模型的分离版本。但是，我们必须考虑到位扫描
每个函数调用只执行一次，因此您必须在性能重要之前调用函数数十亿次，这很少有程序这样做。因此，努力制作一些特殊版本的模板是不值得的
功能的cpu与慢位扫描指令。我的建议是使用位扫描指令，并期望这是未来cpu上的快速解决方案。

一半大小的执行单位。矢量寄存器的大小已经从64-bitMMX增加到128位XMM、256位YMM和5个12位ZMM寄存器。支持128位向量寄存器的第一个处理器实际上只有64位执行的离子单元。每一个
128位操作被分成两个64位操作，因此几乎没有任何操作
在使用更大的矢量大小时，具有更大的速度优势。后来的型号配备了完整的128位执行单位和以后更高的速度。同样地，第一个处理器
支持的256位指令将256位读操作分解为两个128位读取。我们可能会看到类似的进一步扩张。通常，新寄存器大小的全部优势只来自于第二代处理器。在某些情况下，用户实现仅在具有全尺寸执行单元的cpu上是最优的。这个
问题是，CPU调度程序很难知道是否是最大的
矢量寄存器的大小以半速或全速处理。解决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，仅在这样支持AVX2时使用AVX
申请或者，使用对使用新指令集不利的否定主义处理器。

•高精度数学。用于高精度数学的库允许将具有非常多位的整数相加。这通常是在ADC的循环中完成的（add withcarry）
进位必须从一次迭代保存到下一次迭代的指令。The
进位位可以保存在进位标志或寄存器中。如果进位位保持在进位标志中，则循环必须在使用零标志的指令上分支
并且不要修改进位标志（例如DEC、JNZ）。这种解决方案可能会产生很大的
由于所谓的部分标志而导致的延迟在某些处理器上停滞，因为CPU
在较旧的IntelCPU上，将标志寄存器分离为进位和零标志有问题，但在AMD CPU上没有（参见手册3：“Intel、AMD的mic roarchitecture
和通过cpu”)。这是CPUmodel需要调度的少数案例之一。在一个特定品牌的西部CPU上工作得最好的版本很可能是同一品牌未来型号的最佳选择。较新的
处理器支持高精确数学的辅助结构。

内存复制。有几种不同的方式来复制记忆块。
这些方法将在手册2中进行讨论：“优化装配中的子程序”
“语言”，第17.9节：“数据的移动块”，其中也讨论了哪个方法在不同的处理器上最快。在C++程序中，您应该选择一个最新的函数库，作为记忆函数的详细补充。
对于不同的微处理器、不同的对齐和不同大小的数据块，有许多不同的情况，唯一合理的解决方案是
有一个处理CPU调度的标准函数库。这个函数是如此重要和普遍使用，以至于大多数函数库都有这个函数的CPU调度，尽管不是所有的库都有最好和最新的解决方案。
除非有复制构造函数另有规定，否则编译器在复制largeobject时可能会隐式地使用memcpy函数。

在这种困难的情况下，重要的是要记住，你的代码很可能大部分时间都在编程时未知的处理器上运行。因此，它是
重要的是要考虑哪种方法可能在未来的处理器上工作得最好，并为所有支持必要指令集的未知处理器选择这种方法。基于复杂的标准或特定的列表来制作CPU调度程序是很少值得的
CPU模型如果由于微处理器硬件设计的普遍改进，这个问题可能会在未来消失。

最终的解决方案是包括一个性能测试来测量
关键代码的每个版本，以查看哪个解决方案在实际处理器上是最佳的。
然而，这涉及时钟频率可能动态变化以及由于中断和任务切换而测量不稳定的问题，因此有必要
为了做出可靠的决定，交替测试不同的版本几次。

13.4测试和维护

当软件使用CPUdispatching时，有两件事需要测试：

1.使用特定的代码版本可以提高多少速度。

2.检查所有代码版本是否正常工作。

速度测试最好在每个特定代码分支所针对的CPU类型上进行。换句话说，如果你想为几个不同的CPU进行优化，你需要在几个不同的CPU上进行测试。

另一方面，没有必要有许多不同的CPU来验证所有代码
分支工作正常。较低指令集的代码分支仍然可以在具有较高指令集的aCPU上运行。因此，您只需要在中设置了最高指令的aCPU
为了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，允许您覆盖CPU调度并运行任何代码分支。

如果代码实现为函数库或分离模块，那么可以方便地制作一个测试程序，可以单独调用所有代码分支并测试它们
功能这对以后的维护非常有帮助。然而，这并不是一个教科书上的测试理论。关于如何测试一个软件模块的正确性的建议，必须被找到
别处

13.5实施

CPU调度机制可以在不同的地方实现，在不同的时间进行调度决策：

•每次通话都要派单。BRANCH TREE或switch语句导致关键函数的适当版本。每次调用关键函数时都会进行分支。这样做的缺点是分支需要时间。

•第一次呼叫时调度。函数是通过函数指针调用的，函数指针最初指向调度程序。dispatcher更改functionpointer并使其
指向函数的正确版本。这样做的优点是，在函数从未被调用的情况下，它不会花费时间来决定使用哪个版本。在实例中说明了该方法见下文13.1。

•初始化时的Makepointer。程序或库有一个初始化例程，它在第一次调用关键函数之前被调用。初始化例程设置
函数指向函数的正确版本。这样做的优点是函数调用的响应时间是一致的。

•初始化时的Loadlibrary。每个代码版本都是在单独的
动态链接库（*。dl l或*。所以）。该程序有一个初始化例程，用于加载库的适当版本。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，则此方法非常有用。

•在加载时调度。该程序使用程序链接表（PLT），该表在程序加载时初始化。这种方法需要操作系统支持，在较新版本的Linux和Mac OS中可用。参见第页142以下。

•在安装时调度。每个代码版本都在单独的
动态链接库（*。dll或*。所以）。安装程序会创建一个指向库适当版本的符号链接。应用程序加载
通过符号链接的库。

•使用不同的可执行文件。如果指令集互不兼容，则可以使用此方法。您可以为32位和64位系统制作单独的可执行文件。在安装过程中可以选择适当的程序版本
进程或通过可执行文件存根。

如果关键代码的不同版本是用不同的编译器编译的，那么
建议为关键代码调用的任何库函数指定staticlinking，这样您就不必分发所有动态库（*。dll或*。所以）属于应用程序的每个编译器。

各种指令集的可用性可以通过系统调用（例如。
Windows中的IsProcessorFeatuRepresentation）。或者，您可以直接调用CPUID指令，或者使用库中的InstructionSet()
万维网。阿格纳。org/optimize/asmlib。zip或instrset_detect()函数
fromhttps://github.com/vectorclass/version2/blob/master/instrset_detect.cpp.
下面的示例展示了使用指令设置（）函数实现第一次调用方法。另请参见的示例12.6页128用于CPU与向量类库的调度。

```
// Example13.1
//  CPU调度

//头文件的指令设置（）
#包括"asmlib.h"

//定义具有所需参数的函数类型
类型（parm1，parm2）；

//函数原型
```

```
临界功能类型CriticalFunction_Dispatch;

//函数指针作为入口点。
//在第一次调用后，它将指向适当的函数版本批评功能类型*命令命令命令函数
=&CriticalFunction_Dispatch;

//最低版本
int CriticalFunction_386( int parm1, int parm2){...}

// SSE2版本
int CriticalFunction_ SSE2( int parm1, int parm2){...}

// AVX版本
int CriticalFunction_ AVX ( int parm1, int parm2){...}

//调度员。将只被第一次调用
int criticalfunction_dispatch（int parm1, int parm2）
{
   //使用asml ib库获取支持的指令集
   int level=InstructionSet();

   //设置指向适当版本的指针（可能使用表
   //如果有许多分支）：
   如果（水平>=11）
   {//AVX支持
      CriticalFunction=&CriticalFunction_AVX;}
   否则如果（水平>=4）
   {//SSE2支持
      CriticalFunction=&CriticalFunction_SSE2;}
   else
   {//通用版本
      CriticalFunction=&CriticalFunction_386;}

   //现在调用所选版本
   返回（*CriticalFunction）（parm1, parm2）；}

int main()
{
   因塔，b，c;
   ...

   //通过函数指针调用关键函数
   A=（*临界函数）（b，c）；

   ...
   返回0；}
```

函数库中有InstructionSet()函数asmlib，即
不同的编译器有不同的版本。此功能独立于操作系统
检查CPU和操作系统是否支持不同的指令集。示例中CriticalFunction的不同版本如果需要，13.1可以放在单独的模块中，每个模块都为特定的指令集编译。

13.6 Linux中的CPU调度atload time

Linux中引入了一个名为“Gnuindirect function”的特性，并在2010年得到了Gnu实用程序的支持，在2015年得到了Clangin的支持。该特性在程序加载时调用adispatch函数。dispatch函数返回一个指向

函数，并且这个指针被放置在一个普通的过程链接表（PLT）中。ind irectfunction特性需要编译器、链接器和加载器的支持（需要binutils
版本2.20，glibc版本2.11 ifunc分支）。此功能仅适用于使用ELF文件格式的平台，即。Linux和FreeBSD，而不是Windows和MacOS。

请注意，dispatcher函数是在程序开始运行之前和调用任何构造函数之前调用的。因此，dispatcher函数不能依赖其他任何东西
正在初始化，并且它无权访问环境变量。即使从未调用被调度函数，也会调用dispatcher函数。示例13.2展示如何使用Gnuindirect函数特性。

```
//示例13.2。使用Gnu间接函数进行CPU调度

//函数instrset_detect()借用自：
//github。
com/vectorclass/version2/blob/master/instrset_detect。
cpp#包括“INSTRSET_DETECT.CPP”

//将myfunc声明为分派函数
//调度程序的名称用引号给出
int myfunc(int)attribute((ifunc("myfunc_dispatch")));

//通用版本
int myfuncGeneric(int a){
    看跌期权（“通用”）；
    返回A；}

//AVX2版本
int myfuncAVX2(int a){
    看跌期权(“AVX2”);
    返回A；}

//AVX512版本
int myfuncAVX512(int a){
    看跌期权（“AVX512”）；
    返回A；}

//声明函数类型
//（替换参数类型和返回类型以适应您的情况）
typedef int functype(int);

//避免名称混淆
外部“C”{
    //dispatch函数返回指向所选版本的指针
    functype*myfunc_dispatch(){
        //检测支持的指令集int
        instrset=instrset_detect();

        //返回指向所选函数的指针
        如果(instrset>=10){
            返回myfuncAVX512
        }
        else if(instrset>=8){
            返回myfuncAVX2;
        }
        否则{
            returnmyfuncGeneric}
    }
}
```

间接函数特性在Gnu C函数库中用于特别关键的函数。

13.7英特尔编译器中的CPU调度

英特尔编译器有两个版本，一个是名为“经典”的旧版本，另一个是名为“基于LLVM”的新版本，它是Clangcompiler的一个分支。

英特尔编译器“经典”有一个功能，以使多个版本的功能
不同的英特尔的cpu。它使用每个调用方法上的调度。当调用该函数时，将向所需的函数版本进行调度。自动调度可以通过编译一个模块来实现一个模块中的所有合适的功能
/QaxAVX或-axAVX。这将使所有相关函数的多个版本。通过使用该指令，只能对速度关键功能进行调度
__解密规范（cpu_调度（…））。详细信息请参见Intel C++编译器文档。

英特尔编译器目前是唯一可以自动生成CPU调度用户代码的编译器。不幸的是，CPU调度机制是针对英特尔公司进行的
编译器只适用于英特尔的cpu，而不适用于AMD或其他品牌。

由Intel编译器提供的CPU调度机制不如Gnu编译器机制有效，因为它使得对关键函数的每个调用进行调度，
而Gnu机制在过程链接中存储指向所需版本的指针
桌子。如果调度函数用英特尔编译器调用另一个调度函数，则执行后者的调度分支，尽管CPU类型在这里是已知的。这可以通过内联后一个函数来避免，但是像示例中那样显式地执行CPU调度可能会更好第13.1页140.

英特尔函数库具有自动CPU调度功能。对于不同的处理器和指令集，许多英特尔库函数都有多个版本。

不幸的是，Intel Compiler Classic中的CPU检测机制有几个缺陷：

- 只有在英特尔上运行时，才会选择代码的最佳版本
  加工CPU调度程序在检查处理器是否支持哪个指令。代码的一个劣质的“通用”版本是可选择的处理器不是英特尔，即使进程与一个更好的代码版本兼容。这可能会导致AMD和VIA处理器的性能急剧下降。

显式的CPU调度仅适用于智能处理器。Anon-Intel处理器
  通过执行使程序崩溃的非法操作或打印错误消息，使调度程序发出错误信号。

CPU调度程序不允许操作系统支持XMM寄存器。它将在不支持SSE的旧操作系统上崩溃
  。

由Intel发布的几个功能库具有类似的CPU调度机制，其中一些也以次优的方式处理非IntelCPU。

使用英特尔编译器或英特尔函数库构建的软件在其他品牌上表现不佳，这一事实已经成为许多争议和法律诉讼的来源。见我的博客详情。

新的“基于LLVM”的英特尔编译器表现更好。此编译器不检查CPUbrand，只检查支持的指令集。然而，“基于LLVM的”Intelcompiler不支持用户代码的自动CPU调度。只有中的代码函数库已调度。

Intel FunctionLibraries中的CPU调度程序有两个版本，一个版本检查如果CPU不是Intel，则CPUbrand并给出较低版本，并且另一个版本只检查支持哪些指令集。例如，对于这两个版本，标准库中的内部调度程序分别命名为intel_cpu_features_init()和intel_cpu_features_init_x()。其他英特尔函数库也有类似的调度程序，其中它不检查CPU布兰德后缀_x。检查CPUbrand的不公平版本使用了经典版本的英特尔编译器，而处理非英特尔处理器的_xversion使用了基于LLVM的英特尔编译器和非英特尔编译器。注意，这这些信息是基于我自己的实验，可能不适用于所有的情况。

结论是，经典版本的英特尔编译器永远不应该被用于可能运行在非智能处理器上的软件。基于llvm的英特尔公司可能是用于非英特尔处理器，但您可以使用plai n Clang编译器，因为这两个编译器是完全的。

这个基于LLVM的Intel编译器指出，代码与-m或/ arch：选项应该执行在任何兼容的，非英特尔处理器与支持相应的指令集。仅在Intel处理器上，以-x或/Qxwillr开始的选项编译的代码。

英特尔函数库针对英特尔处理器进行了优化，但基于LLVM的英特尔编译器似乎也能在非英特尔处理器上提供良好的性能。然而，这是基于我自己有限的实验。到目前为止还没有明确的英特尔确认其函数库不会降低非英特尔处理器的性能（参见讨论这里）。

我以前发表过关于如何覆盖不公平CPU调度的各种技巧英特尔函数库。这些技巧显然不再需要最新版本的库，只要它们不与经典版本的Intelcompiler结合使用。

14个具体优化主题

14.1使用查找表

从缓存的表中读取值。通常它只需要几个时钟周期来从一个表中读取。如果函数只有一个有限的功能，我们可以通过用表查找替换函数调用来利用这个事实可能输入的数量。

让我们取整数阶乘函数（n！）作为一个例子。唯一允许的输入是从0到12之间的整数。越高的输入给予溢出，负输入s给予无穷大。阶乘函数的一个典型实现如下所示：

```
// Example14.1a
因特（intn）{//n!
   inti, f=1;
   对于（i=2; i<=n; i++）f*=i;
   返回f; }
```

这种计算需要n-1的乘法，这可能需要相当长的时间。使用辅助连接表更有效：

```
//示例14.1b
因特（intn）{//n!
    //因子表：
    事实表[13]={1、1、2、6、24、120、720，
        5040,40320,362880,3628800,39916800,479001600};
    如果（（无符号int）n<13）{//边界检查（参见第页147)
        返回FactorialTable[n]；//表查找}
    否则{
        返回0；//如果超出范围，则返回0
    }
}
```

此实现使用查找表，而不是每次
函数被调用。我在这里添加了大量对n的检查，因为当n是arrayindex时，n超出范围的后果可能比n是loopcount时更严重。边界检查的方法在下面的第页解释147.

该表应该声明为constin，以便启用常量传播和其他优化。您可以内联声明函数。

在可能输入的数量有限且没有缓存问题的情况下，用查找表替换函数是最有利的。如果您希望在每次调用和计算函数所需的时间小于重新加载值所需的时间，那么使用一个连接表是不利的
从内存加上成本到占用一条高速缓存线的程序的其他部分。

表格连接不能用当前的指令集进行向量化。如果这会阻止一个更快的矢量化代码，请不要使用查找表。

在静态内存中存储一些东西可能会导致缓存问题，因为静态数据是
很可能会分散在不同的记忆地址上。如果你感到疼痛是个问题，那么它就会出现
也许将表从静态内存复制到堆栈内存之外的堆栈很有用。这是通过在一个函数内部声明表，但在内部st循环之外，并且没有静态关键字：

```
//示例  14.1 c
void CriticalInnerFunction(){
    //阶乘表：
    const int FactorialTable[13]={1,1,2,6,24,120,720,
        5040,40320,362880,3628800,39916800,479001600};
    ...
    inti，A，b;
    //关键最里面的循环：
    对于（i=0；i<1000；i++）{...
        A=阶乘表[b]；
        ...
    }
}
```

示例中的FactorialTable14.1 c在以下情况下从静态内存复制到堆栈
调用CriticalInnerFunctionis。编译器将把表存储在staticmemory中，并在函数的开头插入一段将表复制到stackmemory的代码。当然，复制表需要额外的时间，但当它超出临界范围时，这是允许的
最里面的循环。循环将使用存储在堆栈内存中的表的副本

与其他局部变量是连续的，因此可能比静态内存更有效地缓存。

如果您不想手工计算表值并在代码中插入这些值
然后你当然可以让程序做计算。只要只做一次，计算表格所需的时间并不重要。有人可能会说，这样做更安全
在程序中计算表格，而不是键入值，因为手写表格中的打字错误可能不会被检测到。

查表原理可用于程序运行的任何情况
在两个或多个常数之间。例如，在两个
常量可以被一个有两个条目的表替换。如果分支的可预测性很差，这可能会提高性能。例如：

```
//示例14.2 a
浮动A；int b；
a=（b==0）？1.0 F：2.5 F；
```

如果我们假设bis总是0或1，并且该值的可预测性很差，那么用表查找替换分支是有利的：

```
//示例14.2 B
浮动A；int b；
常量浮点OneOrTwo5[2]={1.0 F，2.5 F}；
A=1或2[b&1]；
```

在这里，为了安全起见，我用了1。b&1肯定没有其他的
值大于0或1（参见第147).当然，如果
bis值保证为0或1。写a=OneOrTwo5[b！=0]；也会起作用，
尽管效率略低。然而，当b是浮点数或
因为我测试过的所有编译器都实现了OneOrTwo5[b！=0]作为
一个或两个5【（b！=0）？1：0】在这种情况下，我们不能摆脱分支。它可能
当bis浮点时，编译器使用differentimplementation似乎是合乎逻辑的。我想，原因是编译器制造商假设浮点比较器更多
比整数比较更可预测。解A=1.0f+b*1.5f；当bis是浮点时是有效的，但如果bis是整数，则不是，因为整数到浮点的转换比表查找花费更多的时间。

查找表作为交换语句的替代品特别有利，因为交换语句经常受到branch预测差的影响。样例

```
// Example14.3a
intn;
开关(n){
case0:
   printf（“阿尔法”）；打破；
case1:
   Printf（“Beta”）；打破；
case2:
   Printf（“Gamma”）；打破；
case3:
   Printf（“Delta”）；中断；}
```

这可以通过使用一个可查找的方法来改进：

```
//示例14.3b
intn;
char常量*常量希腊[4]={
```

```
    "Alpha"、"Beta"、"伽玛"、"Delta"}
;
如果（（无符号int）n<4){//检查该索引不超出范围printf （希腊语[n]）；
}
```

表的声明有两次常量，因为指针和它们指向的文本都是常量的。

14.2边界检查

在C++中，通常需要检查数组索引是否超出。这通常看起来像：

```
// Example14.4a
常量int大小=16; int i;
浮点列表大小
...
如果（i<0||i>=大小）{
    cout<<"错误：索引超出范围"；}
else{
    list[i]+=1.0f;}
```

<0和>=大小的两个比较可以用单项比较代替：

```
//示例14.4b
如果（（无符号int）i>=（无符号int）大小）{
    cout<<"错误：索引超出范围"；
}
else{
    list[i]+=1.0f;}
```

当i是时，i的一个可能的解释值将作为一个很大的正函数出现
被解释为一个未签名的整数，这将触发错误条件。用一个替换两个比较可以使代码更快
，因为测试一个条件是相对的
代价昂贵，而类型转换不会生成额外的代码。

此方法可以扩展到您想要检查一个整数是否在某个区间内的一般情况：

```
// Example14.5a
常量最小=100，最大=110; int i;
...
如果（i>=min&i<=max）{..。
```

可更改为：

```
//示例14.5b
如果（（无符号int）i-（无符号int）最小<=（无符号int）（最大最小in））{..。
```

如果所需的间隔值的长度为2的功率，则有一种更快的方法来限制整数的范围。样例

```
// Example14.6
浮点列表[16]; int i;
...
list[i&15]+=1.0f;
```

这需要一点解释。i&15的值保证在0到15的间隔内。如果i在这个区间之外，例如i=18，那么&运算符（按位和）将切断ito4位的二进制值，结果将为2。结果与i模16相同。当数组索引超出范围时，该方法对p更新程序错误很有用，如果有炎，我们不需要发送错误信息。值得注意的是
该方法仅适用于2的幂（即2、4、8、16、32、64，...）。我们可以确保一个值小于2n，而不是被2 n-1所否定。按位和操作分离该数字中最不显著的n位，并将所有其他位设置为零
。

14.3一次性检查多个值的易用运算符

位运算符&，|，^，~，<<，>>可以测试或操作一个整数的所有位
一个操作。例如，如果32位整数的每位具有特定含义，那么您可以使用|操作符在单个操作中设置多个位；您可以清除或掩码
使用运算符切换多位，并且可以使用^运算符切换多位。

操作员可以测试单个操作的多种条件。样例

```
// Example14.7a.测试多个条件
枚举工作日{
    星期日、星期一、星期二、星期三、星期四、星期五、星期六};
WeekdaysDay;
如果（==周二||==周三||==周五）{三时间AWeek（）；
}
```

这个例子有三个条件，它们被实现为重新分支。如果周日、周一等常数定义为2，它们可以连接成一个分支：

```
//示例14.7b.使用&测试多种条件
枚举工作日{
    Sunday=1, Monday=2, Tuesday=4, Wednesday=8,
    Thursday=0x10, Friday=0x20, Saturday=0x40};
WeekdaysDay;
如果（天和（星期二|周三|星期五））{
    DoThisThreeTimesAWeek();
}
```

通过给每个常数一个2的值14.7b，我们每天都用每一点来表示其中一个工作日。这种定义的常数的最大数等于一个整数的位数，通常是32。在64位中
系统可以使用64位整数，几乎没有损失效率。

以表达式（星期二|星期三|星期五）为例14.7b被编译器转换为值0x2Cs，使if条件可以由单个&计算
操作，速度非常快。如果周二、周三或周五的任何数据，&操作的结果将为零，重新计算为真实。

请注意布尔操作数rs&&，||，！和相应的位运算符&，|，~之间的区别。布尔运算符只产生一个结果，分别为true (1)或false (0)；和
只在需要时计算第二个操作数。位运算符产生32
当应用于32位整数时，它们总是计算双操作数。
然而，这位运算符的计算速度比布尔运算符要快得多

因为它们不使用分支，只要操作数是整数表达式，而不是布尔表达式。

使用将整数作为布尔值的位向操作器，您可以做很多事情
向量，这些操作非常快。这可以是有用的程序
布尔表达式。常量是否用枚举、const或#定义定义对性能没有影响。

14.4整数乘法

整数乘法比加减法花费的时间更长(3 -10个时钟周期，
这取决于处理器)。优化编译器通常会用一个常数和移位操作来代替整数乘法。乘以一个的幂2比乘以其他常数要快，因为它可以作为一个移位操作来完成。例如，a*16被计算为一个<<4，而a*17被计算为（a<<4）+ a。

你可以利用这个，最好使用2的幂乘以常数。编译器还有乘以3、5和9的快速方法。

在计算阵元的地址时，乘法是隐含的。在某些酶中，当因子为2的幂时，这种乘法会更快。样例

```
// Example14.8
常量int行=10，列=8;
浮点矩阵
inti,j;
int顺序（int x）;
...
为（i=0; i<行; i++）{j=顺序(i);
   matrix[j][0]=i;}
```

这里，矩阵[j][0]的地址在内部计算为
（int）和矩阵[0][0]+j*（列*大小（浮点））。
现在，乘j乘以（列大小（浮动））的因子=8*4=32。这个幂为2，所以编译器可以用j<<5替换j*32。如果列没有，则没有
当2时，复制时间会更长。因此，它可以是
如果行按非顺序访问，有利于使垫矩阵中的列数为2。

这同样适用于结构或类元素的数组。如果对象按顺序访问，则每个对象的大小最好为2。样例

```
// Example14.9
结构S1{
   inta;
   int b;
   intc;
   intUnusedFiller;
};
int顺序（int x）;
常量int大小=100;
S1列表【大小】; int i, j;
...
对于（i=0; i<大小; i++）{j=order
    （i）;
   列表[j].a=列表[j].b+列表[j].c; }
```

这里，我们在结构中插入了UnusedFiller，以确保其大小是
2为了使地址计算更快。

使用2的幂的优点仅适用于在非
顺序。如果示例中的代码14.8和14.9被更改，使它有i而不是j asindex，然后编译器可以看到地址是按顺序访问的
并且可以通过将一个常量添加到后面的一个常量来计算每个地址（参见72).在这种情况下，大小是否是2的幂并不重要。

使用2的幂的建议不适用于非常大的数据结构。相反，如果矩阵太大，缓存变成了一个
问题。如果矩阵中的列数是2的幂，并且矩阵大于缓存，那么您可能会得到非常昂贵的缓存争用，如第#页所述106.

14.5整数除法

整数除法比加法、减法和乘法需要更长的时间（32位整数需要27-80个时钟周期，具体取决于处理器）。

整数除以2的幂可以通过移位操作来完成，这要快得多。

除以常数比除以变量更快，因为优化编译器可以通过适当选择n来计算a/b asa*（2 n/b）>>n。常数（2n/b）为
提前计算，乘法是用扩展的位数完成的。该方法有些复杂，因为必须添加对符号和舍入误差的各种校正。该方法在手册2：“优化
程序集语言中的子例程”。如果股息没有签署，方法就最快。

以下指导方针可用于改进包含智能划分的代码：

整数除法比除法bya变量更快。确保在压缩时已知除数。

如果常数是2的幂，则整数分裂

如果股息是未签名的，则由一个常数进行的整数分割是最好的。例子
：

```
// Example14.10
inta,b,c;
一个=b/c；//这很慢
一个=b/10；//除以一个常数的速度更快
一个=（无符号int）b/10；//如果无符号，那么速度还会更快
如果除数是2的幂，则=值会更快
一个=（无符号int）b/16；//如果无符号，速度还会更快
```

同样的规则也适用于局部计算：

```
// Example14.11
inta,b,c;
a=b%c；//这种速度很慢
=b%10；//Modulo的速度更快
一个=（无符号int）b%10；//如果无符号，速度还会更快
a=b%16；//如果除数是2的幂，则更快
一个=（无符号int）b%16；//如果无符号，速度还会更快
```

你可以利用这些指导，通过使用一个2的常数，并将股息改为无签名，你确保它不会消极的

如果t除数的值不知道，上述方法仍然可以使用
编译时，但是程序与同一个程序重复划分。在这种情况下，你必须进行必要的计算（2n / b）等。在压实时间。函数库www.阿格纳公司。org/ optimize/ asmlib.zip包含了用于这些计算的各种函数。

循环计数器除以一个常数可以避免。样例

```
//示例14.12 a
int列表[300];
核心;
对于(i=0;i<300;i++){list[i]+=i/3
   ;
}
```

这可以替换为：

```
//示例14.12 b
int列表[300];
int i, i_div_3;
对于(i=i_div_3=0;i<300;i+=3,i_div_3++){list[i]+=i_div_3;
   列表[i+1]+=i_div_3;
   列表[i+2]+=i_div_3; }
```

可以使用类似的方法来避免模运算：

```
//示例14.13 a
int列表[300];
核心;
对于（i=0; i<300; i++）
   {list[i]=i%3;
}
```

这可以替换为：

```
//示例14.13 B
int列表[300];
核心;
对于（i=0; i<300; i+=3）{
   列表[i]=0;
   列表[i+1]=1;
   列表[i+2]=2; }
```

示例中的循环展开14.12 b和14.13 b仅当循环计数被展开因子整除时才起作用。如果没有，那么您必须在循环之外执行额外的操作：

```
//示例14.13 c
int列表[301];
核心;
对于（i=0; i<300; i+=3）{
   列表[i]=0;
   列表[i+1]=1;
   list[i+2]=2;}
list[300]=0;
```

14.6浮点划分

浮点除法所花费的时间远远超过加法、减法和乘法(20 - 45个时钟周期）。

浮点除法为一个常数，应乘以倒数：

```
// Example14.14a
双a，b;
a= b/1.2345;
```

将其更改为：

```
//示例14.14b
双a，b;
a= b*(1./1.2345);
```

编译器将在编译时计算（1./1.2345），并将其倒数插入到
代码，所以你永远不会花时间做划分。一些器将在示例中替换代码14.14 a与14.14 b是自动的，但只有当某些选项可以放松时
浮点运算精度（请参见第页75).因此，显式地进行此优化会更安全。

分裂有时可以被完全消除。例如：

```
// Example14.15a
如果（a> b/c）
```

有时能被取代吗

```
//示例14.15b
如果（a* c>b）
```

但是要小心这里的陷阱：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。

多个部门可以合并。例如：

```
//示例14.16 a
双y、a1、a2、b1、b2;
y=a1/b1+a2/b2;
```

在这里，我们可以通过创建一个公分母来消除一个除法：

```
//示例14.16 B
双y、a1、a2、b1、b2;
y=(a1*b2+a2*b1)/(b1*b2);
```

使用公分母的技巧甚至可以用于完全独立的除法。示例：

```
//示例14.17 a
双a1、a2、b1、b2、y1、y2;
y1=a1/b1;
y2=a2/b2;
```

这可以更改为：

```
//示例14.17 B
```

```
双a1，a2，b1，b2，y1，y2，reciprocal_divisor；
reciprocal_divisor=1./(b1*b2)；
y1=a1*b2*reciprocal_divisor；
y2=a2*b1*reciprocal_divisor；
```

14.7不要混合浮点和双精度

浮点计算无论使用单精度还是双精度，都要花费相同的时间，但它不适合混合单精度和双精度
为64位操作系统编译的程序和为指令集SSE2或更高版本编译的程序的精度。样例

```
// Example14.18a
浮点a、b；
一个=b*1.2；//混合浮动和双重是不好的
```

C/C++标准规定，所有的浮点常数都是双精度的
默认值，所以在这个例子中，1.2是一个双精度常数。因此，在乘以双精度之前，必须将b从单精度转换为双精度
精度常数，然后将结果转换为单精度。这些转换需要很多时间。您可以避免转换，并通过使常数单精度或使db双精度使代码达到5倍：

```
//示例14.18b
浮点a、b；
a=b*1.2f；//一切都是浮动的

//示例14.18 c
双A，b；
a=b*1.2；//一切都是双重的
```

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受到惩罚，但最好保留
在所有操作中都具有相同的精度，如果代码随后被移植到另一个平台。

14.8浮点数和整数之间的转换

从floatingpoint到in teger的转换

根据C++语言的标准，所有从浮点
数字到整数使用向零截断，而不是舍入。这是不幸的
因为截断比舍入花费的时间长得多，除非使用了这2个指令集。如果可能，建议启用SSE2指令。SSE2总是
在64位模式下启用。

在没有SSE2的情况下，从浮点到整数的转换通常需要40个时钟周期。如果您无法避免在关键部分从floator double到int的转换
代码，然后您可以通过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。

函数可以实现从floator double到integer的有效转换
lrintf和lrint。不幸的是，这些功能在许多商业产品中是缺失的
编译器由于对C99标准的争议。lrint的实现
示例中给出了函数见下文14.19。该函数将浮点数舍入到最近的整数。如果两个整数相等，则返回eveninteger。有

没有溢出检查。此函数适用于使用微软、英特尔和Gnu编译器的32位Windows和32位Linux。

```
//示例14.19
静态内联int lrint(double const x){//舍入到最接近的整数intn;
#如果已定义(unix)已定义(GNUC)
   //32位Linux，Gnu/AT&T语法：
   __asm("fldl%1\n fistpl%0": "=m"(n): "m"(x): "memory"); #否则
   //32位Windows，英特尔/MASM语法：
   __asm fld qword ptr x;
   __asm fistp dword ptr n;
#endif
   返回n；}
```

此代码仅适用于Intel/x86兼容微处理器。函数也可以在函数库中找到万维网。阿格纳。org/optimize/asmlib。拉链。

以下示例显示如何使用lrintfunction：

```
//示例14.20
双d=1.6;
int a，b;
a=（int）d; //截断很慢。遗嘱的价值是1
b=lrint(d); //舍入很快。b的值为2
```

在64位模式下或启用SSE2指令集时，舍入和截断之间的速度没有差异。缺失的功能可以在64位模式下或启用SSE2指令集时实现，如下所示：

```
//Example14.21.//仅适用于SSE2或x64
#include<emmintrin.h>

静态内联int lrintf(float const
   x){return_mm_cvtss_si32(_mm_load_ss(&x));}

静态内联int lrint(double const
   x){return_mm_cvtsd_si32(_mm_load_sd(&x));}
```

示例中的代码14.21比其他舍入方法快，但既不比启用SSE2指令集时的截断更快也不慢。

从整数到浮点的转换

只有在启用SSE2指令集时，signedintegers到floatingpoint的转换才很快。只有当AVX512指令集启用时，无符号整数到浮点的转换才会更快。参见第页41.

14.9使用整数操作来操作浮动点变量

浮动点数根据IEEE标准754（1985）以图形表示形式存储。这个标准适用于几乎所有的现代微针仪和操作
系统（但不是一些非常老的文档编译器）。

浮点、双和长双的表示反映了±2 eee的浮点值。 1. fffff，其中±是符号，eee是指数，fffff是符号
分数的二进制小数。signis存储为一个单位，其中0为正数，1为负数。指数存储为一个有偏的二进制整数，而分数存储为二进制数字。如果可能，指数总是归一化，使小数点之前的值为1。这个“1”不包括在表示法中，除了在
长双格式。这些格式可以表示如下：

```
结构浮子{
    无符号整数分数：23；//小数部分
    无符号整数指数：8；//指数+0x7F
    无符号int符号：1；//符号位
};

struct Sdouble{
    无符号整数分数：52；//小数部分
    无符号整数指数：11；//指数+0x3 FF
    无符号int符号：1；//符号位
};

struct Slongdouble{
    无符号整数分数：63；//小数部分
    无符号int one：1；//如果非零且正常，则为1
    无符号整数指数：15；//指数+0x3 FFF
    无符号int符号：1；//符号位
};
```

非零floa tingpoint数的值可以计算如下：

floatvalue=（-1）符号。2指数—127.(1+分数.2-23),
双值=（-1）符号。2指数—1023.(1+分数.2-52),
longdoublevalue=（-1）符号。2指数—16383.（one+fraction.2—63）。

如果除符号位之外的所有位都为零，则该值为零。零可以用符号位表示，也可以不用符号位表示。

浮点格式标准化的事实允许我们处理不同的
浮点表示的一部分直接使用ofinteger操作。这可能是一个优势，因为整数运算比浮点运算更快。只有当你确信自己知道自己在做什么时，你才应该使用这种方法。有关一些注意事项，请参见本节末尾。

我们可以简单地通过反转符号位来改变浮点数的符号：

```
//示例14.22
联合{
    浮动f；
    核心；
}u；
u.i^=0x80000000；//u.f的翻转符号位
```

我们可以通过将符号位设置为零来获取绝对值：

```
//示例14.23
联合{
    浮动f；
    核心；
}u；
u.i&=0x7 FFFFFFF//将符号位设置为零
```

我们可以通过测试除符号位之外的所有位来检查浮点数是否为零：

```
//示例14.24
联合{
    浮动f；
    核心；
```

```
}u;
如果(u.i&0x7FFFFFFF){//测试b its0-30//f为非零
}
```

```
否则{
    //f为零}
```

我们可以通过将n加到指数上来将非零浮点数乘以2n：

```
//示例14.25
联合{
    浮动f;
    核心;
}u;
intn;
if(u.i&0x7FFFFFFF){//检查是否没有nzero
    u.i+=n<<23; //将n添加到指数
}
```

示例14.25不检查溢出，仅对正n有效。如果没有下溢的风险，你可以通过从指数中减去n来除以2n。

指数的表示有偏差的事实允许我们简单地通过将两个正浮点数与整数进行比较来比较它们：

```
//示例14.26
联合{
    浮动f;
    核心;
}u，v;
如果(u.i>v.i){
    //u.f>v.f如果两者都为正}
```

示例14.26假设我们知道u。f和V.F都是阳性的。如果两者都为负数，或者一个为0，另一个为-0（带符号位集的零），它将失败。

我们可以移出符号位来比较绝对值：

```
//示例14.27
联合{
    浮动f;
    无符号inti
}u，v;
如果
    (u.i*2>v.i*2){//abs(u.f)
    >abs(v.f)
}
```

示例中乘以214.27将移出符号位，以便剩余的位表示浮点绝对值的单调递增函数号码。

我们可以通过设置fractionbits将区间0<=n<223中的整数转换为区间[1.0,2.0)中的浮点数：

```
//示例14.28
联合{
    浮动f;
    核心;
}u;
intn;
u.i=(n&0x7FFFFF)0x3F800000; //Now1.0<=u.f<2.0
```

此方法对于rand omnumber生成器很有用。

通常，如果将浮点变量存储在
内存，但不是寄存器变量。Theunion强制将变量存储在
记忆，至少暂时如此。因此，如果代码的其他附近部分可以从使用寄存器中受益，则使用上述示例中的方法将是一个缺点
相同的变量。

在这些例子中，我们使用联合而不是指针的类型转换，因为这种方法更安全。指针的类型转换可能不适用于依赖于标准C的strict别名规则的编译器，指定不同类型的指针不能指向相同的对象，除了charpointers。

上面的例子都使用了singleprecision。在32位系统中使用双精度会带来一些额外的复杂性。double用64位表示，但是32位系统没有对64位整数的固有支持。许多32位系统允许您定义64位整数，但它们实际上表示为两个32位整数，这样效率较低。您可以使用double的高32位，它可以访问符号位、指数位，
和分数中最重要的部分。例如，要测试adouble的符号：

```
//示例14.22 B
联合{
    双d;
    int i[2];
}u;
如果(u.i[1]<0){//测试符号位//u.d为负或-0
}
```

不建议通过仅修改一半来修改double，例如，如果
想用u.i[1]^=0x80000000翻转上面例子中的符号bit；因为这可能会在CPU中产生存储转发延迟（参见手册3：“
在64位系统中，通过使用一个64位整数而不是两个32位整数来别名double，可以避免这种情况。

访问64位双精度的32位的另一个问题是，它不能移植到具有大端存储的系统上。示例14.22 b和14.29因此需要修改，如果
在其他具有大端存储的平台上实现。所有x86平台（Windows、Linux、BSD、基于英特尔的Mac OS等。）都有小端存储，但其他系统可能有双端存储（如PowerPC）。

我们可以通过比较32-62位来近似比较双精度。这对于找到矩阵中数字最大的元素以用作Gauss中的枢轴是有用的
淘汰。示例中的方法14.27可以像这样实现ina枢轴搜索：

```
//示例14.29
常量int大小=100;
//100个双精度数组：
union{double d无符号int u[2]}a[size];
无符号int absvalue，largest_abs=0;
int i，largest_index=0;
对于(i=0;i<size;i++){
    //获取[i]的上32位并移出符号位：
    absvalue=a[i].u[1]*2;
    //查找数值最大的元素（大约）：
    if(absvalue>largest_abs){
        largest_abs=absvalue;
        largest_index=i; }
}
```

示例14.29找到数组中数值最大的元素，或者大致如此。它可能无法区分相对无差异大于2-20的元素，但这已经足够了
准确地用来寻找合适的枢轴元件。整数比较很可能比浮点比较要快。在大型的环境系统上，你必须被替换
`u[1]by u[0].`

14.10Mathematical函数

比较常见的数学函数，如对数，指数函数，
三角函数等。是在x86cpu中的硬件中实现的。然而，a
在大多数情况下，当sse2指令可用时，软件实现比硬件实现更快。大多数编译器使用的软件实现，如果
已启用SSE2或侧向指令集。

使用软件实现而不是硬件实现这些功能的优点是，单精度比双精度更高。但在大多数情况下，软件实现比硬件实现更快，即使是双精度。

您可以使用不同的编译器，包括librarylibmmt.liband头文件mathimf.hthat附带的IntelC++编译器。这个
库包含许多有用的数学函数。很多高等数学
函数在英特尔的Math内核库中提供，可从www.intel.com.英特尔函数库针对英特尔处理器进行了优化，但它们通常提供合理的
在AMD和其他处理器上的性能（请参见143).

向量类库包括具有向量输入的优化数学函数
（https：//github.com/vectorclass）。参见第页131进一步讨论向量函数。

14.11静态库与动态库

函数库可以实现为静态链接库（*。lib，*。a）或动态链接库，也称为共享对象（*。d ll，*。所以）。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其复制到可执行文件中。只有可执行文件需要分发给最终用户。

动态链接的工作方式也会有所不同。到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加载到内存中。两者都
可执行文件和所有的动态文件库都需要分发给最终用户。

使用静态链接器而非动态链接器的优点是：

静态链接只包括该库中实际上需要的部分
应用程序，而动态链接使enti库（或至少是很大一部分）加载到内存中，即使只需要一小部分库代码。

当使用静态链接时，所有的代码编码都包含在一个单可执行文件中。动态链接使其在程序启动时必须加载几个文件。

在动态库中调用函数所花费的时间比在静态链接库中所花费的时间更长
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。

•当代码分布在

多个动态库。动态库在可除以内存页大小的圆形内存地址处被加载（4096）。这将使所有动态库都包含

对于相同的缓存行。这使得代码缓存和数据缓存的效率降低。

·由于需要与位置无关的代码，动态库在某些系统中效率较低，见下文。

•安装使用同一动态库的较新版本的第二个应用程序
如果使用动态链接，可以更改第一个应用程序的行为，但如果使用静态链接，则不能更改。

动态链接的优点是：

•同时运行的多个应用程序可以共享相同的动态库
而不需要将库的多个实例加载到内存中。这是
在同时运行许多进程的服务器上非常有用。实际上，只有代码
节和只读数据节可以共享。任何可写数据部分都需要每个进程一个实例。

·动态库可以更新到新版本，而不需要更新调用它的程序。

·可以从不支持静态链接的编程语言调用动态库。

·动态库可用于制作插件，为现有程序添加功能。

权衡每种方法的上述优点，很明显静态链接对于速度关键功能是优选的。许多函数库有静态和动态版本。如果sp eed很重要，建议使用静态版本。

一些系统允许函数调用的延迟绑定。惰性绑定的原理是，当程序加载时，链接函数的地址不会被解析，而是等到第一次调用函数。Lazybinding对于在单个会话中实际上只有少数函数被调用的大型库非常有用。但是lazybinding肯定会降低被调用函数的性能。第一次调用函数时会出现相当大的延迟，因为它需要加载动态链接器。

延迟绑定会导致交互式程序中的可用性问题，因为
例如，对菜单点击的响应时间变得不一致，有时长得不可接受。因此，Lazybinding应该只用于非常大的库。

无法确定加载动态库的内存地址
提前，因为一个固定的地址可能与另一个需要相同地址的动态库发生冲突。有两种常用的方法来处理这个问题：

1.搬迁。如有必要，将修改代码中的所有指针和地址，以适应实际加载地址。搬迁将由客户和装载机完成。

2.位置无关的代码。代码中的所有地址都与当前位置相关。地址计算已在运行时完成。

Windows DLLsuse重新定位。dll由链接器重新定位到特定的负载
地址如果此地址不空，则DLL将再次重新定位（重新定位）到其他地址。从主可执行文件对DLL中的函数的调用会发生
通过动画端口表或一个指针。DLL的变量可以从主访问
通过一个输入的指针，但这个特性很少使用。Itis更常见
通过函数调用交换数据或指向数据指针。对内部数据的内部引用

DLL在32位模式下使用绝对引用，在64位模式下主要使用相对引用
时尚。后者的效率稍高，因为相对引用在加载时不需要重新定位。

默认情况下，类UNIX系统中的共享对象使用与位置无关的代码。这比重新定位效率低，尤其是在32位模式下。下一章描述了这是如何
工作并提出避免职位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-
独立代码。“位置无关代码”这个名字实际上意味着的不仅仅是它所说的。编译为与位置
无关的代码具有以下特征：

- •代码部分不包含需要重新定位的绝对地址，而只包含自身相对地址。因此，代码段可以在任意位置加载
  内存地址和多个进程之间的共享。

- ·数据段不在多个进程之间共享，因为它通常包含可写数据。因此，数据部分可以包含指针或地址
  需要搬迁。

- •在Linux和BSD中，所有公共函数和公共数据都可以被覆盖。如果main executable中的函数与sharedobject中的函数同名，则main中的版本将优先，不仅在从main调用时，而且在从sharedobject调用时也优先。同样，当main中的全局变量与sharedobject中的全局变量同名时，main中的实例将是
  使用，即使从共享对象访问时也是如此。这个所谓的符号
  interpositionis旨在模拟静态库的行为。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），以便实现这种“覆盖”特性。所有对函数和公共变量的访问都要通过PLT和GOT。

允许在Linux和BSD中覆盖公共函数和数据的符号插入特性价格很高，而且在大多数库中从未使用过它。当调用共享对象中的函数时，有必要在过程关联表（PLT）中查找函数地址。每当一个共享对象中的公共变量被访问时，它都是
有必要首先查找全局偏移表（GOT）中的变量的地址。即使从中访问函数或变量，也需要可查找
相同的共享对象。显然，所有这些表查找操作都大大降低了执行速度。更详细的讨论可以在
http://www.macieira.org/blog/2012/01/sorry-state-of-dynamic-libraries-on-linux/

另一个连续的负担是在32位模式下的自相对论的计算。32位x86指令集没有关于数据的自相对寻址的指令。该代码通过以下步骤来访问一个公共数据对象：(1)通过功能调用来获取它所播种的地址。(2)通过自助亲属地址找到目标。(3)在GOT中查找数据对象的寻址，最后(4)通过该地址访问该数据对象。在64位调制解调器中不需要步骤(1)，因为x86-64指令集支持自相对的
称呼

在32-bitLinux和BSD中，缓慢的GOTlookup过程用于所有静态数据，包括不需要“覆盖”特性的本地数据。这包括静态变量，浮动变量
点常量、字符串常量和初始化的数组。我没有解释为什么在不需要的时候使用这个删除过程。

显然，这是避免与位置无关的代码和表的最佳方法
查找isto使用静态链接，如上一章（第158).在无法避免动态链接的情况下，有各种方法可以避免时间-
位置无关代码的消费特征。这些变通方法依赖于系统，如下所述。

32位Linux中的共享对象

共享对象通常使用-fpicaccording选项编译
编译器手册。此选项使代码节与位置无关，为所有函数创建PLT，为所有公共数据和静态数据创建GOT。

可以在没有-fpicoption的情况下编译共享对象。然后我们就摆脱了一切
上面提到的问题。现在代码将运行得更快，因为我们可以访问
内部变量和内部函数在一个步骤中，而不是上面解释的复杂的地址计算和表查找机制。编译的共享对象
没有-fpic会快得多，也许除了一个非常大的共享对象，其中大多数函数永远不会被调用。在32位Linux中不使用-fpic进行编译的缺点是加载程序将有更多的引用需要重新定位，但是这些地址计算是
只做一次，而运行时地址计算必须在每次访问时完成。
在没有-fpic的情况下编译时，代码部分需要每个进程一个实例
因为代码部分中的重定位对于每个进程来说都是不同的。显然，我们失去了重写公共符号的能力，但无论如何很少需要这个特性。

为了移植到64位模式，您最好避免使用全局变量或隐藏它们，如下所述。

64位Linux中的共享对象

在64位模式下，计算自相对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。对特殊位置无关代码的需求较小，因为在64位代码中，默认情况下通常使用相对地址。但是，我们仍然希望getrid的GOT和PLTlookups
本地引用。

如果我们在64位模式下编译没有-fpicin的共享对象，我们会遇到另一个
问题。编译器有时使用32位绝对地址，主要用于静态数组。这在主可执行文件中起作用，因为它肯定是在下面的地址加载的2
GB，但不是在共享对象中，共享对象通常加载在更高的地址，而32位（有符号）地址无法到达该地址。链接器将在此生成一条错误消息
凯斯。最好的解决方案是使用-fpie选项而不是-fpic进行编译。这将
在代码部分生成相对地址，但它不会对内部引用使用GOT和plt。因此，它将比用-fpic编译时运行得更快，并且不会有上面提到的32位情况下的缺点。-fpie选项在32位模式下用处不大，它仍然使用aGOT。

另一种可能是使用-mcmodel=large编译，但这将使用full64位
所有东西的地址，这是相当低效的，它会在代码部分生成重定位，因此不能共享。

64位共享对象中不能有公共变量-fpie
因为当链接器看到对公共的相对引用时，它会发出错误消息
变量，其中它需要一个GOT条目。您可以通过避免任何公共的
变量。所有全局变量（即在anyfunction外部定义的变量）都应该通过使用声明“static”或“attribute((visibility
（“隐藏”））））”。

gnu编译器版本5.1和更高版本有一个选项-fno-semantic-interposition，这使得它避免使用PLT和GOTlook，而只用于同一个文件中的引用。通过使用内联assemblycode为变量指定两个名称，一个全局名称和一个本地名称，并为localreferences使用本地名称，也可以获得相同的效果。

尽管有这些技巧，您可能仍然会得到错误消息：“relocationR_X86_64_PC32
创建sharedobject时不能使用符号“functionname”重新编译
使用-fPIC”，当共享对象由多个模块（源文件）组成并且从一个模块调用另一个模块时。我还没有找到解决这个问题的办法。

BSD中的共享对象

BSD中的共享对象的工作方式与Linux中的相同。

32位Mac OS X

默认情况下，32位Mac OS X MakePosition-独立代码和延迟绑定的编译器，即使不使用共享对象。目前用于计算自我的方法-
32-bitMac代码中的相对地址使用了一种不幸的方法，导致返回地址被错误预测（见手册3：“Intel、AMD和VIAcpu的微架构”）。

所有与共享对象无关的代码都可以通过在编译器中关闭与位置无关的代码fl来显著地加速。因此，请记住，在为32-bitMacOSX编译时，始终指定编译器选项-fno-pice，除非您正在创建一个共享对象。

当您使用选项-fno-picand链接与选项-read_only_relocs抑制进行比较时，可以创建没有与外部位置无关的代码的共享对象。

GOT和PLT表不用于内部引用。

64-bitMac OS X

代码部分是独立的，因为这是这里使用的内存模型最有效的解决方案。这个编译选项显然没有任何效果。

GOT和PLT表不用于内部引用。

在MacOS X中，不需要采取特殊的预防措施来加速64位共享对象。

14.13系统编程

设备驱动程序、中断服务例程、系统内核和高优先级线程是速度特别关键的领域。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函数的执行。

系统代码必须遵守关于registeruse的某些规则，如第三章所述
“手册5中内核代码中的注册用法：”调用不同C++的约定
编译器和操作系统”。因此，您只能使用用于系统代码的编译器和函数库。系统代码应该用C++或
汇编语言。

节约系统代码中的资源使用非常重要。动态内存分配是
特别危险，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。队列应该实现为循环缓冲区
固定大小，而不是链表。不要使用标准C++容器（请参见95).

15元编程

元编程意味着制作代码。例如，在解释的sc riptlanguages中，通常可以制作一段代码来生成一个字符串，然后
将此字符串解释为代码。

元编程可以在编译语言（如C++）中用于执行某些操作
如果计算的所有输入在编译时都可用，则在编译时而不是在运行时进行计算。当然，在解释性语言中，一切都发生在运行时，就没有这样的优势了。

以下技术可以被认为是C++中的元编程：

·预处理器指令。例如，使用#if而不是if。这是删除多余代码的一种非常有效的方法，但是
预处理器可以做到这一点，因为它先于编译器，它只理解最简单的表达式和运算符。

制作一个C++程序，生产其他C++程序（或部分）。这在某些情况下很有用，例如生成最终程序中作为常数数组的数学函数表。当然，这需要您编译第一个程序的输出。

优化编译器可能会尝试在编译时尽可能多地做一些事情。例如，所有好的编译器都会减少int x=2*5；到int x=10；

模板是在编译时进行实例化的。模板实例在编译之前被其实际值替换。这就是为什么使用模板没有虚拟成本的原因。58).理论上，用模板元编程表示任何算法都是可能的，但这种方法非常复杂和笨拙，编译可能需要大量的时间。

编译时间分支数。如果constexpr（布尔表达式）{}。这个
括号内的布尔表达式可以是任何包含在编译时已知的值的表达式。{}中的代码将只包含在决赛中
如果boolis为真。假分支中的代码被删除，但在语法上仍然必须正确。如果出现在模板内部，那么语法检查就不那么严格了。这个特性对于决定代码输出的哪个版本很有用。if
constexpr特性不利于有条件的声明，因为该声明的范围仅限于{}。编译时间分支要求在编译器中激活C++17标准。请参见第页166的细节。

Subargexpr函数。一个constexpr函数可以在编译时进行任意计算，只要使用在编译时已知的参数调用它。请参见第页167年以下。这要求C++14标准或更高版本在抗菌肽中激活。

15.1模板元编程

下面的示例解释了如何使用元编程来加速
当指数是编译时已知的整数时幂函数的计算。

```
//示例15.1 a。计算x的10次方
双xpow10(双x){
    返回功率（x，10）；
}
```

powfunction在一般情况下使用对数，但在这种情况下，它可以认识到
10是整数，因此只能使用乘法来计算结果。当指数为正整数时，幂函数内部使用以下算法：

```
//示例15.1 b.使用循环计算整数幂
double ipow(double x, unsigned int n){
    双y=1.0; //用于乘法
    while(n! =0){//为nn中的每个位循环
        如果（n&1）y*=x; //如果位=1，则乘以
        x*=x; //平方x
        n>>=1; //获取n的下一位
    }
    返回y; //返回y=pow(x,n)
}

双xpow10(双x){
    返回ipow（x，10）; //ipow比pow快
}
```

示例中使用的方法当我们推出循环和重组时，15.1 b更容易理解：

```
//示例15.1 C。计算整数幂，循环展开
双xpow10(双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
    返回    x10;                        //返回x^10
}
```

正如我们所看到的，只需四次乘法就可以计算出pow（x，10）。如何
有没有可能从例子中15.1 b至15.1 C？我们利用n在编译时是已知的这一事实来消除所有只
依赖于n的东西，包括
whileloop、If语句和所有theinteger计算。示例中的代码15.1 C比15.1 b，并且在这种
情况下它也可以更小。

从示例转换15.1 b至15.1 c是由我手动完成的，但是如果我们想生成一个适合任何编译时
常数n的代码，那么我们需要
metaprogramming.只有最好的编译器才能转换示例15.1 a或15.1 b至
15.1cautomatically.元程序混合对于编译器无法减少的情况很有用
自动地

这个示例显示了使用模板元编程实现的这个计算。
如果它看起来太复杂了，就不要惊慌。模板元编程可以非常好
复杂的幸运的是，新的C++17标准化提供了一种更简单的方法，我们将在下一章中看到。

我给这个例子只是为了简化曲折和复杂的模板编程。

```
//示例15.1d.使用模板元编程的整数幂

//为pow（x，N）的模板，其中N是一个正整数常数。
//一般情况下，N不是2的幂：
模板<boolIsPowerOf2，intN>
粉末类
平民
    静态双p（双x）{
    //删除N的二进制表示中最右边的1位：
    #defineN1(N&(N-1))
```

```
        返回powN<（N1&（N1-1））==0，N1>：：p（x）*powN<true，N-N1>：：p（x）
    ；#undef N1
    }}
;

//N A 2幂的部分模板专门化
模板<int N>
类powN<true，N>{
公众：
    静态双p（双x）{
        返回powN<true，N/2>：：p（x）*powN<true，N/2>：：p（x）；
    }}
;

//N=1的完整模板专门化。这将结束递归模板<>
类powN<true,1>{
公众：
    静态double p(double
        x){return x;
    }}
;

//N=0的全模板专门化
//这仅用于避免无限循环，如果powN为
//错误地用IsPowerOf2=false调用，而它应该是true。
模板<>
类功率<true,0>{
公众：
    静态双p（双x）{return 1.0;
    }}
;

//x的N次方函数模板
template< int N>
静态内联双整数功率（双x){
    如果N为2的幂次，则//（N&N-1）==0
    返回powN<（N&N-1）==0，N>：：p(x)；}

//使用模板使x达到10的幂次
双xpow10（双x）{
    返回整数Power<10>(x);
}
```

如果你想知道这是如何工作的，这里有一个解释。如果你不确定你需要它，请提供以下解释。

在C-++模板元编程中，循环是通过无递归模板来实现的。这个powN模板通过调用自己来模拟while循环15.1b.
分支机构是通过（部分的）模板专门化来实现的。这就是示例中的if分支的执行方式15.1 b实现。递归必须始终以非递归结束
模板专门化，而不是使用模板内部的分支。

粉末n模板是一个类模板，而不是一个函数模板，因为它是部分的
模板专门化只允许用于类。将N拆分为其二进制表示的各个位尤其棘手。我用了一个技巧，N1=N&（N-1）给出了N的值，去掉了最右边的1位。如果Nis是2的幂，那么N&（N-1）是0。常数N1可以用宏以外的其他方式定义，但是
这里使用的方法是我尝试过的所有编译器中唯一有效的方法。

好的编译器实际上减少了示例15.1 d至15.1 c如预期的那样，因为它们可以消除常见的子表达式。

为什么模板元编程如此复杂？因为C++模板特性从来就不是为此而设计的。这只是碰巧成为可能。模板元-
编程是如此复杂，我认为除了最简单的程序之外，使用它是不明智的
案件。复杂的代码本身就是一个风险因素，验证、调试和维护这些代码的成本如此之高，以至于很少证明性能的提高是合理的。

15.2使用constexpr分支的元编程

幸运的是，有了C++17标准，元编程变得更加简单，它提供了带有constexprkeyword的编译时分支。

下面的示例显示了与示例相同的算法15.1，使用编译时分支。

```
//示例15.2。用C++17计算整数幂

//递归模板，下面使用
//计算y*pow(x，n)
模板<int n>
inline double ipow_step(double x，double y){if
    constexpr((n&1)==1){
        y*=x；//如果位=1，则乘以
    }
    constexpr int n1=n>>1；//获取n的下一位
    如果constexpr(n1==0){
        返回y；//已完成
    }
    否则{
        //平方x并继续递归
        返回IPOW_STEP<n1>(x*x，y)；}
}

//pow（x，n）的高效计算
模板<int n>
double integerPower(double x){
    如果constexpr(n==0){
        返回1。；//pow(x,0)=1
    }
    否则，如果constexpr（n<0）{
        // x为负
        如果转换（（无符号int）n==0x80000000u）{//- n溢出
            return0.;}
        // pow (x,n)=1/ pow (x,-n)
        return1./ integerPower<-n>(x);}
    //循环通过递归
    返回ipow_步骤<n>（x，1.）;}
```

在这里，我们仍然需要一个递归模板来推出循环，但是它更容易创建分支和结束递归。一个好的编译器将减少示例15.2只有大量的复制指令，没有别的。被捕获的分支将不包括在鳍代码中。

在使用C++17引入编译时分支之前，我们遇到了一个问题
模板在一个未被占用的分支中会被扩展，即使ifit也会被删除。这可能

导致无限递归或未使用的分支数量的指数增长。在C++17之前结束模板递归的唯一方法是使用模板专门化，如
样例15.1 d.如果递归模板包含许多分支，它可能需要很长时间来编译。

编译时分支的效率更高，因为未占用分支中的模板不会被扩展。

15.3使用constexpr函数的元编程

constexprfunction是一个几乎可以在
编译如果参数是编译时间常数。使用C++14标准，您可以在constexprfunction中拥有分支、循环等。

这个例子找到整数中最有效的1位的位置。这与位扫描反向指令相同，但在编译时计算：

```
//示例15.3。使用constexpr函数查找最高有效位
constexpr int bit_scan_reverse(uint64_t const
    n){if(n==0)return-1;
    uint64_t a=n，b=0，j=64，k=0;
    来自{
        j>>=1;
        k=(uint64_t)1<<j;
        如果
            (a>=k){a>>
            =j;
            b+=j；}
    }while(j>0);
    返回int(b)；}
```

一个好的优化编译器将在编译时做简单的计算，如果所有的
输入是已知的常数，但是如果编译器涉及分支、循环、函数调用等，则很少有编译器能够进行更复杂的计算和卫星编译时间。const expr函数可以用来确保某些计算是在编译时完成。

constexpr函数的结果可以在需要编译时间常数的地方使用，例如数组大小或编译时
间分支。

而C++14和C++17则为元编程提供了重要的改进
可能性，仍然有一些事情你不能用C++语言来做。例如，不可能创建一个重复10次的编译时循环来生成十个名为func1的函数，func2，...，func10的函数。这在某些脚本语言和
装配工

16测试速度

测试一个程序的速度是优化工作的一个重要部分。你必须检查你的修改是否真的提高了速度。

有不同的分析器是有用的寻找热点和
测量一个程序的过度性能。分析器不总是准确的，然而，可能很难准确地衡量你想要什么当程序
花费大部分时间等待用户输入或读取磁盘文件。请参见第页16.分析的讨论。

当识别出一个热点问题时，那么仅在代码的这部分上隔离热点和改进条件可能会很有用。这可以通过解决
CPU时钟，通过使用所谓的时间戳计数器。这是一个测量自CPU启动以来时钟脉冲数量的计数器。一个时钟周期的长度是
时钟频率的倒数，如在第页上所解释的15.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，那么您可以得到两个时钟计数之间的差值的确切时间消耗。

可以通过列出的函数ReadTSC获得时间戳计数器的值
下面的例子16.1.此代码仅适用于支持内部函数的编译器。或者，您可以使用头文件
TimingTest.h
万维网。阿格纳。组织/优化/测试。压缩或获取ReadTSC asa库函数万维网
。阿格纳。org/optimize/asmlib。拉链。

```
//示例16.1
#include<intrin.h>//或#include<ia32 intrin.h>等。

long long ReadTSC(){//返回时间戳计数器
    int dummy[4]; //对于未使用的退货
    易失性intDontSkip//易失性以防止操作优化
    长长的钟; //时间
    __cpuid(dummy, 0); //序列化
    DontSkip=dummy[0]; //防止优化离开cpuid
    时钟=rdtsc（）; //读取时间
    returnclock;}
```

可以使用此函数测量关键代码之前和结束后的时钟计数。测试设置可能是这样的：

```
// Example16.2

# include< stdio.h>
# include< asmlib.h>      //使用来自库asmlib..//或示例中的ReadTSC
                          （）16.1
voidCriticalFunction();                          //这是我们要测量的函数

...

计算测试次数=10; //要测试的次数
长时间1，长时间11;
每个测试的时间差
对于（i=0; i<编号的测试次数; i++）{//重复编号的测试次数

    时间1=ReadTSC（）; //测试前的时间

    关键功能（）; //要测试的关键功能

    时间[i]=ReadTSC（）-time1; //（时间后）-（时间前）
}
Printf（"\n结果："）; //打印标题
for(i=0;i<NumberOfTests;i++){//循环打印出结果
   printf("\n%2i%10I64i",i,timediff[i]);
}
```

示例中的代码16.2调用关键函数十次并存储时间
数组中每次运行的消耗。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ReadTSCfunction所需的时间。您可以从计数中减去此值。在示例中，只需删除对CriticalFunction的调用即可测量16.2.

测量的时间按以下方式解释。第一次计数通常高于随后的计数。这是代码和数据未缓存时执行CriticalFunction所需的时间。后续计数给出代码和
数据被尽可能好地缓存。第一个计数和随后的计数代表“最差情况”和“最佳情况”值。这两个值中哪一个最接近事实
这取决于在决赛中是叫一次时间还是叫多次时间
程序和是否有其他代码使用缓存之间的调用
关键功能。如果你的优化努力集中在CPU效率上，那么这是"最好的情况"，你应该看看某个修改是否有利可图。
另一方面，如果您的优化工作是集中于为了提高缓存效率而安排数据，那么您可以使用“最坏情况”计数。在任何情况下，时钟计数应该乘以时钟周期和次数
关键功能是指调用一个典型的应用程序来计算最终用户可能会经历的时间延迟。

偶尔，你测量的时钟计数要高得多。这个
在执行关键功能期间发生任务切换时发生。您可以在受保护的操作系统中删除它，但是您可以通过在测试前增加线程优先级，然后将优先级设置为正常来减少问题。

时钟计数经常波动，可能很难得到重复的结果。这是因为现代cpu可以根据工作负载动态地改变它们的时钟频率。当工作负荷高时，时钟频率增加
当工作负载较低时，以节省电力。有各种各样的方法来获得更多可重复的时间测量：

通过在代码测试之前给它一些繁重的工作来预热CPU。
禁用BIOS设置程序中的省电选项。
onIntelcpu：使用核心锁定循环计数器（见下文）

16.1使用性能监视器计数器

许多CPU都有一个内置的测试功能，称为性能监视器计数器。performancemonitor计数器是CPU内部的一个计数器，它可以被设置为对ce事件进行计数，
例如执行的机器指令数、缓存未命中、分支
错误预测等。这些计数器对于调查性能非常有用
问题。performancemonitor计数器是特定于CPU的，每个CPU型号都有自己的一组performancemonitor环选项。

CPU供应商正在提供适合其CPU的分析工具。英特尔的分析器称为VTune；AMD的Profiler叫做CodeAnalyst。这些分析器对于识别代码中的热点非常有用。

为了我自己的研究，我开发了一个使用性能监视器的测试工具
柜台。我的测试工具支持Intel、AMD和VIAprocessors，可从
www.agner.org/optimize/testp.zip。这个工具不是分析器。它不是为了寻找热点，而是为了在热点被识别后研究一段代码。

我的测试工具有两种用途。第一种方法是插入要测试的代码片段
测试程序本身并重新编译它。我觉得这个很有趣，因为测试单个组件
指令或小代码序列。第二种方法是设置性能
在运行要优化的程序之前监视计数器，并读取
在您想要执行的代码段之前和之后，程序中的性能计数器
测试。您可以使用与示例中相同的原理16.2以上，但阅读一个或多个
性能监视器计数器代替（或补充）时间戳计数器。测试工具可以在所有CPU内核中设置和启用一个或多个性能监视器计数器

并使它们处于启用状态（每个CPU内核中有一组计数器）。计数器将一直亮着，直到您将它们打开，直到计算机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.阿格纳公司。org/ optimize/ testp.zip）。

在英特尔处理器中，一个特别有用的性能监视器计数器被称为核心c锁
周期核心时钟周期计数器是以CPU核心运行的实际时钟频率来计算时钟周期，而不是外部时钟。这就给出了一个几乎独立于时钟频率变化的度量方法。核心时钟周期抵消在测试代码测试的哪个版本时非常有用，因为你可以避免时钟频率上升和下降的问题。

记住，在未测试时插入一个开关程序以关闭计算器的读取。试图读取性能监视器禁用将使程序崩溃。

16.2单位休息的缺陷

在软件开发中，通常会分别测试每个函数或类。这个单元测试对于验证一个优化后的函数的功能是必要的，但幸运的是，单元测试并没有在速度方面提供关于该功能的性能的完整信息。

假设您有两个不同版本的关键函数，并且您想找出答案
这是最快的。测试它的典型方法是制作一个可调用的小测试程序
关键函数多次使用合适的测试数据和测量它需要的时间。在这个单元测试下执行最好的版本可能有一个更大的内存
占用的空间比替代版本要大。对缓存失败的惩罚在这个单元中看不到-
测试，因为测试程序使用的代码和数据内存的总量很可能比大容量更大。

当关键函数插入到最终程序中时，代码缓存、微操作缓存和数据缓存很可能是关键资源。现代cpus速度如此之快，以至于花费在执行指令上的锁周期比内存访问和缓存大小不太可能成为瓶颈。如果是这种情况，那么临界函数的最优版本
也许是在单元测试中时间更长但内存更小的。

例如，如果您想知道它是否有利于推出一个大的循环，那么您就不能在不考虑缓存效应的情况下依赖于单元测试。

你可以通过查看链接映射器来计算记忆函数使用了多少
assemblylisting.对链接器使用“生成映射文件”选项。代码缓存使用和数据缓存使用都可能是关键的。分支目标缓冲区我也有一个可能出现的缓存。因此，函数中的跳转、调用和分支的数量也应该是
经过仔细考虑的

现实性能测试应该只包括一个函数或热点，但还应该包括包含关键功能和热点的最内部循环。测试应该是
使用一组真实的数据来执行，以获得分支的可靠结果
错误的预测。性能测量不包括等待用户输入的任何部分。应该测量文件输入和输出所用的时间
分别地

不幸的是，通过单元测试来衡量性能的谬误非常常见。一些可用的最佳优化函数库使用过多的循环解除滚动，因此内存占用不合理地大。

16.3最坏情况测试

大多数性能测试都是在最佳案例的条件下进行的。所有干扰影响都被删除，所有资源都足够，缓存条件是最优的。最佳案例测试是有用的，因为它提供了更可靠和可重复的结果。如果你想
比较同一算法的两种不同实现的性能，然后需要消除所有干扰影响，以使测量准确
并且尽可能可重复。

然而，在某些情况下，在最坏情况下测试性能更重要。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限制，那么您应该在最坏情况下测试响应时间。

产生流式音频或视频的程序也应该在最坏情况下进行测试，以确保它们始终符合预期的实时性
速度。输出中的延迟或故障是不可接受的。

在测试最坏情况性能时，以下每种方法都可能相关：

- 第一次激活程序的特定部分时，可能会很慢
  由于代码的延迟加载、缓存和分支错误预测。

测试整个软件包，包括所有简单的文库和框架，
  而不是孤立一个单一的功能。在软件包的不同部分之间进行切换，以增加程序代码的某些部分被缓存或甚至被交换到磁盘上的可能性。

依赖于网络资源和服务器的软件应该在一个流量很大的网络和一个全用户的服务器上进行测试，而不是一个专用的测试服务器。

使用数据文件和标签的数据。

使用一台CPU慢，内存不足，很多
  安装了不相关的软件，大量的后台进程，以及一个缓慢和破碎的硬盘。

用不同品牌的cpu、不同类型的图形卡等进行测试。

使用一个防病毒程序，扫描文件的访问。

- 同时运行多个进程或线程。如果微处理器有超线程，那么尝试在同一个处理器内核中运行两个线程。

- 尝试分配比现有RAM更多的RAM，以强制将内存交换到磁盘。

·通过使最内部循环中使用的代码大小或dat大于缓存大小来引发缓存未命中。或者，您可以主动使缓存无效。操作系统可能具有用于此目的的功能，或者您可以使用
  `_mm_clflushintrinsic`函数。

·通过使数据比正常数据更随机来引发分支错误预测。

17嵌入式系统中的优化

小型嵌入式应用中使用的微控制器比标准PC具有更少的计算资源。时钟频率可以低一百倍或甚至一千倍；RAM内存的数量甚至可能比aPC少一百万倍。然而，如果你避开大型图形框架、解释器、即时编译器、系统数据库以及其他通常在大型系统上使用的额外软件层和框架，就有可能制作出在如此小的设备上运行相当快的软件。

系统越小，选择使用资源少的软件框架就越重要。在最小的设备上，您甚至没有操作系统。

通过选择一种可以在aPC上交叉编译，然后作为机器代码传输到设备的编程语言，可以获得最佳性能。任何需要在设备上编译或解释的语言都是对资源的巨大浪费。为了
由于这些原因，首选的语言通常是C-或C-++。关键的设备驱动程序可能需要汇编语言。

C++只比Cif需要更多的资源。您可以根据最适合所期望的程序来选择C或C++
构造

节约内存是很重要的。应该在它们被用在的函数中声明大数组，以便在函数返回时它们被释放。
或者，您也可以对多重元素重用相同的数组。

所有使用新/删除或malloc/免费的动态内存分配应该避免，因为管理内存堆的开销大。
堆管理器有一个数据库收集器，它可能会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，标准的C-++容器类使用动态内存分配
和删除，而且往往过于如此。这些容器绝对应该避免使用
除非你有充足的资源。例如，aFIFO队列应该实现为固定大小的基本缓冲区，以避免动态内存分配。不要使用链接表
（请参见第页97).

字符串类的所有常见实现都使用动态内存分配。您应该避免这些属性，并以老式的C风格处理文本字符串作为特征数组。没有te
C样式的字符串函数没有检查数组的溢出。它是
程序员的责任是确保数组足够大，以处理包括终止零在内的字符串，并在必要时进行溢出检查(见页105).

虚拟函数比非虚拟函数资源中的虚拟函数。尽可能避免使用虚拟功能。

更小的微处理器有本地浮点执行单元。任何浮点
在这样的处理器上的操作需要大量的浮动点库
强烈的因此，您应该避免使用任何浮动点表达式。例如，=b*2.5可以更改为=b*5/2(请注意可能发生的溢出
中间表达式b*5)。只要你有了唱a
十进制点指向你的程序，你将加载整个浮动点库。例如，如果你想用两个小数来计算数，你应该把它乘以100
它可以用一个整数来表示。

整数变量可以是8、16或32位（很少是64）。您可以保存RAMspace，如果
必要的，通过使用不会导致特定的溢出的静态整数大小

申请整数大小并没有在各个平台上进行标准化。有关每个整数类型的大小，请参见
编译器文档。

中断服务例程和设备驱动程序特别重要，因为它们可以阻止其他一切事情的执行。这个正常的程序属于系统编程领域，但没有操作系统的应用程序，这是应用程序程序员的工作。因此，程序员忘记系统代码是关键时的风险更大
没有操作系统，因此系统代码与
应用程序代码。一个中断服务例程应该做尽可能少的工作。通常，它应该在静态缓冲区中保存一个接收数据或从缓冲器发送数据。它不应该响应一个命令或做其他的输入/输出。由中断接收的命令最好以较低的优先级作出响应
级别，通常在主程序中的消息循环。请参见第页162号文件，以便进一步讨论系统代码。

在本章中，我已经描述了一些特别重要的考虑事项
在资源有限的小型设备上。目前手册中其余部分的大部分建议也与小型设备有关，但与小型微控制器的设计也有一些差异：

较小的微控制器没有分支预测离子。43).在这些软件中，不需要考虑到分支预测。

较小的微控制器没有高速缓存。91).不需要组织数据来优化缓存。

较小的微控制器没有故障缺陷。没有必要去分解依赖关系链。21).

18.编译器选项的概述

表18.1中。与优化相关的命令行选项				
	MS编译器窗口	**Gnu和Clang编译器**	**英特尔编译器窗口**	**智力编译器Linux**
优化速度	/O2 or/Ox	-O3或-Ofast	/O3	-O3
程序间优化	/Og			
整个程序优化	/GL	联合收割机 -fwhole-program	/齐坡	-首次公开发售
无异常处理	/EHs-			
没有f.p。异常捕获	/fp：除了-	-fno诱捕数学 -fno-数学-errno		
无堆栈框架	/投票	-fomit-frame-pointer		-fomit-frame-pointer
Noruntime类型标识(RTTI)	/GR-	-fno-rtti	/GR-	-fno-rtti
假设指针别名	/Oa			-fno-别名
非严格浮点	/fp：快速	-快速-数学	/fp：快速 /fp：快速=2	-fp-模型快速，-fp-模型 快速=2
融合乘加		-ffp-contr-act=快速		
除法 常数=mul的倒数		-倒数-数学		
假设没有溢出或NAN		-仅限有限数学		
忽略零号		-无符号零		
简单成员指针	/vms	-fno-complete-member-pointers/vm		
快速呼叫功能	/克			
Vectorcall函数	/全球之声			
函数级链接（删除未引用的功能）	/Gy	-功能-部分	/Gy	- 功能-部分
SSEinstruction set（128位浮点向量）	/arch：SSE	-msse	/arch：SSE	-msse
SSE2指令集合（128个整数或双精度向量）	/拱门：SSE2	-msse2	/拱门：SSE2	-msse2

SSE3指令集		-msse3	/拱门：SSE3	-msse3

补充。SSE3instr. 设置		-mssse3	/拱门：SSSE3	-mssse3
SSE4.1instr .放置		-msse4.1	/arch：SSE4.1	-msse4.1
阿文斯特尔。放置	/arch：AVX	-mAVX	/arch：AVX	-mAVX
AVX2实例。放置	/arch：AVX2	-mAVX2	/arch：AVX	-mAVX2
AVX512F实例。放置		-mavx512f		
AVX512VL/BW/DQ 实例。放置	/arch： AVX512	-mavx512vl -mavx512bw -mavx512dq	/arch：COREAVX512	-mavx512vl -mavx512bw -mavx512dq
自动 向量化	速度快 /fp：除	-O2-fno- 捕获数学错误数学错误=libmvec		
由多线程实现的自动并行分区			/Qparallel	-平行
平行化的 OpenMP指令	/openmp	-fopenmp	/Qopenmp	-openmp
32位代码		-m32		
64位代码		-m64		
静态连接 多线路	/MT	静态的	/MT	静态的
生成 汇编列表	/FA	-S-masm=intel	/FA	-S
生成地图文件	/Fm			
生成 优化报告			/Qopt报告	选择报告

各种浮点选项代表了速度和精度之间的不同妥协。这包括以下内容：

=快速：*b+顺式转换为融合的乘法加法。中间精度（a*b）计算。

-freciprocal-math：除以常数，改为乘以倒数。这提高了速度，对精度的影响最小。

-fassociative-math：(a+b)+c=a+(b+c)。这可能会对
精度。示例：浮点a=1.E10，b=(1+a)-a；这给出了0 with-fno-
联想数学和1 with-fassociative-math。程序员可能更喜欢使用括号来指示所需的计算顺序，而不是使用这个
选项。

-ffinite-math-only：无法传播的优化。例如：x-x可以优化为0。如果x=INF，这将是错误的，因为inf-inf=NAN。警告：
使用此选项可能无法检测NAN和INF。

-fno-trapping-math：自动矢量化所必需的。x=0.0/0.0是减少的tox=NAN，没有陷阱。

-fno-math-errno：notmake为sqrtetc设置errno的陷阱。这对于包含sqrt的分支的自动矢量化是必要的。

-fno-signed-zeros：启用忽略零符号的优化，例如x*0。=0和x+0.=x。

表18.2。与优化相关的编译器指令和关键字

	MS编译器	Gnu和Clang编译器	英特尔Windows编译器	英特尔Linux编译器
对齐16	__declspec(align(16))	__属性((对齐(16)))	__declspec(align(16))	__属性((对齐(16)))
对齐by16(C++11)	alignas（16）	alignas（16）	alignas（16）	alignas（16）
假设指针对齐			#pragma向量比对	#pragma向量比对
假设指针没有别名	#pragma优化（"A"，on） 限制	__限制	__declspec（无别名） __限制 #pragma ivdep	__限制 #pragma ivdep
假设功能是纯粹的		__属性((const))		__属性((const))
假设功能没有投掷例外情况	throw()	throw()	throw()	throw()
假设功能仅从同一模块调用	静态的	静态的	静态的	静态的
假设成员功能仅从同一模块调用		__属性((可见性（"内部"）））		__属性((可见性（"内部"）））
矢量化			#pragma向量总是	#pragma向量总是
优化功能	#pragma使最优化			
快速调用函数	__快速调用	__attribute((fastcall))	__快速调用	
矢量调用函数	__向量调用	__vectorcall (Clangonly)	__向量调用	
非缓存写入			#非时间语用向量	#非时间语用向量

表18.3。预定义宏					
	MS编译器窗口	**格努编译程序**	**克朗编译程序**	**智力编译器窗口**	**智力编译器Linux**
汇编者识别	_MSC_VER和不__英特尔公司	__GNUC而不是__英特尔，COMPI，LER，而不是__叮当地响	__叮当地响	__英特尔COMPILER或__英特尔LLVM编译器	__英特尔COMPILER或__英特尔LLVM编译器
16位平台	不是_WIN32	n.a.	n.a.	n.a.	n.a.
32位平台	不是_WIN64			不是_WIN64	
64位平台	_赢64	_第64页	_第64页	_赢64	_第64页
视窗平台	_赢32			_赢32	
Linux平台	n.a.	__unix_林克斯	__unix_林克斯		__unix_林克斯
x86平台	_混合86			_混合86	
x86-64平台	__MIX86__andWIN64			__MX64	__MX64

19文献

其他手册由Agner Fog提供

本手册是五本手册系列中的第一本。参见第页3论坛标题列表。

代码优化文献

英特尔：“英特尔64和DIA-32架构优化参考手册”。开发商。英特尔。com.
关于IntelCPU的C++和汇编代码优化的许多建议。定期制作新版本。

AMD：《AMD家族15 H处理器软件优化指南》。万维网。amd。com.
关于AMDCPUs C++和汇编代码优化的建议。定期制作新版本。

英特尔：“英特尔®C++编译器文档”。包含在英特尔C++编译器中，可从www.intel.com.
使用英特尔C++编译器优化功能的手册。

维基百科关于编译器优化的文章。先生。维基百科。org/wiki/compiler_optimization。

ISO/IEC TR18015，“C++性能技术报告”。www.open-std.org/jtc1/sc22/wg21/docs/TR18015.pdf。

OpenMP.万维网。开放议员。组织。用于并行处理的OpenMP指令的文档。

ScottMeyers：“有效的C++”。艾迪森-韦斯利。第三版，2005年；和“更有效的C++”。艾迪森-韦斯利，1996年。
这两本书包含了许多关于高级DC++编程的技巧，如何避免难以发现的错误，以及一些关于提高性能的技巧。

Stefan Goedecker和Adolfy Hoisie：“数值密集码的性能优化”，SIAM 2001。
C++和Fortran代码优化高级书籍。主要重点是大型数据集的数学应用。CoversPC、工作站和科学矢量处理器。

小亨利·S·沃伦：《黑客的快乐》。艾迪森-韦斯利，2003年。
包含许多位操作技巧

迈克尔·阿布拉什：“代码优化的禅”，科里奥利集团图书公司1994年。
大多过时了。

RickBooth：“内部循环：快速32位软件开发的原始资料”，Addison-Wesley1997。
大多过时了。

微处理器文档

英特尔：“IA-32英特尔架构软件开发人员手册”，第1卷、第2A卷、第2B卷、第3A卷和第3B卷。开发商。英特尔。com.

AMD：《AMD64架构程序员手册》，第1-5卷。 www.amd.com.

互联网论坛

几个互联网论坛和新闻组包含了关于代码优化的有用的讨论。看针对新闻组的www.agner.org/optimize和FAQ

comp.lang.asm.x86塞默林克斯。

20版权通知