文本, 应用程序, 电子邮件

描述已自动生成

最终应用程序的效率取决于编程语言的实现方式。当代码被编译并作为二进制可执行代码分发时，可以获得最高的效率。大多数C++、Pascal和Fortran的实现都基于编译器。其他几种编程语言都是用解释来实现的。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，因为循环的主体在循环的每次迭代中都要被一次又一次地解释。

编程语言及其实现的历史揭示了一个曲折的过程，反映了效率，平台独立性和易于开发的相互冲突的考虑。

C++的一个重要缺点是安全性。不检查数组边界冲突、整数溢出和无效指针。

2.8克服C++语言的缺点

**Portability**

C++语言最严重的问题是安全性。标准C++实现没有检查数组边界冲突和无效指针。这是C++程序中常见的错误来源，也是黑客可能的攻击点。有必要遵守某些编程原则，以防止在安全问题的程序中出现此类错误。

每次创建或修改字符串时，字符串类都会分配一个新的内存块。这可能导致内存碎片化，并涉及堆管理和垃圾收集的高开销成本。一个不影响安全性的更有效的解决方案是将所有字符串存储在一个内存池中。

3、寻找最大的时间消费者

我们可能会决定做一些测量，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 ms

3.14上下文切换

上下文切换是多任务环境中不同任务之间的切换，多线程程序中不同线程之间的切换，或大程序不同部分之间的切换。频繁的上下文切换会降低性能，因为数据高速缓存、代码高速缓存、分支目标缓冲区、分支模式历史等的内容可能必须更新。如果分配给每个任务或线程的时间片较小，则上下文切换更频繁。时间片的长度由操作系统决定，而不是由应用程序决定。在具有多个CPU或具有多个内核的CPU的计算机中，上下文切换的数量较少。

4性能和可用性

5选择最佳算法

我必须警告不要过度杀戮。如果一个简单的算法可以足够快地完成这项工作，就不要使用高级和复杂的算法。例如，一些程序员甚至对最小的数据列表使用哈希表。对于非常大的数据库，哈希表可以显着缩短搜索时间，但对于太小以至于二分搜索甚至线性搜索都足够快的列表，没有理由使用它。哈希表增加了程序的大小以及数据文件的大小。如果瓶颈是文件访问或缓存访问而不是CPU时间，这实际上会降低速度。复杂算法的另一个缺点是，它使程序开发更昂贵，更容易出错。

7.1不同类型的变量存储

如果数据在内存中随机分布，则数据缓存效果很差。因此，了解变量是如何存储的非常重要。简单变量、数组和对象的存储原则是相同的。

我们可以从中学到的教训是，所有变量和对象最好在它们被使用的函数内部声明。

通过在{}括号内部声明变量，可以减小变量的作用域。然而，大多数编译器即使可以在退出声明变量的{}括号时释放内存，也不会在函数返回前释放变量的内存。如果变量存储在寄存器中（见下文），那么它可能会在函数返回前被释放。

全局或静态存储

全局或静态存储 在函数外部声明的变量称为全局变量。它们可以从任何函数中访问。全局变量存储在内存的静态部分。静态内存也用于声明为静态关键字的变量，浮点常量，字符串常量，数组初始化列表，switch语句跳转表和虚函数表。

静态数据区域通常分为三部分：一部分用于程序永远不会修改的常量，一部分用于程序可能修改的已初始化变量，以及一部分用于程序可能修改的未初始化变量。

字符串常量和浮点常量通常存储在静态内存中

但是，更好的解决方案可能是将访问保存变量的函数作为同一个类的成员，并将共享变量存储在类内部。

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。存储在寄存器中的变量的访问速度非常快。所有优化编译器都将自动选择函数中最常用的变量进行寄存器存储。同一寄存器可用于多个变量，只要它们的使用（有效范围）不重叠。局部变量特别适合寄存器存储。这也是首选局部变量的另一个原因。

Volatile关键字

**防止编译器优化**

编译器在优化代码时，通常会假设变量的值不会被程序之外的因素改变。因此，它可能会将变量的值缓存到寄存器中，以减少内存访问的开销。然而，如果变量的值可能被外部因素（如硬件、其他线程等）修改，这种优化就会导致程序行为错误。

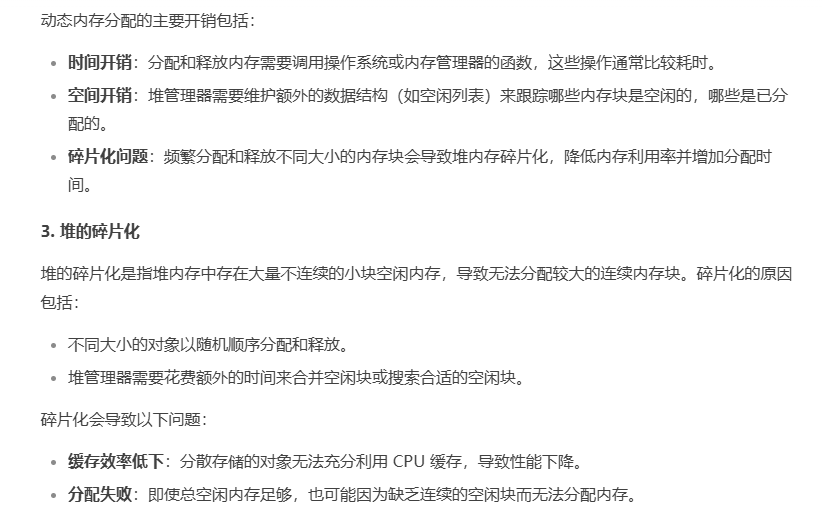
volatile 的作用是告诉编译器：

* **不要缓存该变量的值**：每次访问该变量时，都必须从内存中读取或写入。
* **不要对该变量进行优化**：编译器不能假设该变量的值在两次访问之间保持不变。

volatile 关键字的作用是确保变量存储在内存中而不是寄存器中，并防止对该变量的所有优化。这在测试情况下非常有用，可以避免某些表达式被优化掉。

需要注意的是，volatile 并不意味着原子性。它不会阻止两个线程同时尝试写入变量。在上面的示例中，如果代码尝试将 seconds 设置为零的同时，另一个线程正在递增 seconds，代码可能会失败。更安全的实现是只读取 seconds 的值，并等待该值变化五次

动态内存



文本

描述已自动生成

在类内部声明的变量

除了最简单的情况，对象不能存储在寄存器中，但它的数据成员可以复制到寄存器中。

带有static修饰符的类成员变量将存储在静态内存中，并且将有且只有一个实例。

Integer operators

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

乘法和除法需要更长的时间。在 Pentium 4 处理器上，整数乘法需要 11 个时钟周期，而在大多数其他微处理器上需要 3-4 个时钟周期。整数除法需要 40-80 个时钟周期，具体取决于微处理器。在 AMD 处理器上，整数大小越小，除法越快，但在 Intel 处理器上则不然。

浮点变量和运算符

文本

描述已自动生成

7.6指针和引用

在一个结构良好的 C++ 程序中，大多数变量实际上都是以某种方式通过指针访问的。因此，微处理器必须设计得使指针高效，而事实也确实如此。

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

函数内部声明的所有非静态变量和对象都存储在栈上，实际上是通过相对于栈指针的地址来访问的。同样，类中声明的所有非静态变量和对象都是通过 C++ 中称为 this 的隐式指针访问的。

文本

描述已自动生成

智能指针

智能指针在程序的逻辑结构规定对象必须由一个函数动态创建，然后由另一个函数删除，并且这两个函数彼此无关（不是同一个类的成员）的情况下很有用。如果同一个函数或类负责创建和删除对象，则不需要智能指针。

如果一个程序使用许多小的动态分配对象，每个对象都有智能指针，那么你可以考虑这个解决方案的成本是否太高。将所有对象集中到一个容器中可能会更有效，最好使用连续的内存。

将浮点数转换为整数需要很长时间，除非启用SSE 2或更高版本的指令集。通常，转换需要50 - 100个时钟周期。

7.12分支和switch语句

现代微处理器的高速度是通过使用流水线来实现的，在流水线中，指令在执行之前在几个阶段中被提取和解码。然而，管道结构有一个大问题。每当代码具有分支（例如，if-else结构）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。如果错误的分支被馈送到流水线中，则直到10 - 20个时钟周期之后才检测到错误，并且在这段时间期间通过提取、解码以及可能推测性地执行指令而完成的工作已经被浪费。其结果是，每当微处理器将分支馈送到流水线中时，并且随后发现它选择了错误的分支时，微处理器浪费了几个时钟周期。

**1. 循环的效率**

* 循环的效率取决于 CPU 对循环控制分支的预测能力。
* 对于固定且较小的循环次数，CPU 可以完美预测循环控制分支。
* 对于嵌套循环，某些处理器可以预测所有层次的循环，而其他处理器只能预测最内层循环。
* 对于高重复次数的循环，只有在循环退出时才会发生分支预测错误，因此预测错误的惩罚可以忽略不计。

图形用户界面, 文本

描述已自动生成

函数

文本

描述已自动生成

也就是说，如果可能的话，对于短小，调用频繁的函数尽可能内联，在最内层的循环中，避免调用框架函数，所谓框架函数是指，我调用这个函数后，这个函数还会调用其他函数，而叶函数则不会调用其他函数，对吗

7.16函数返回类型

函数的返回类型最好是简单类型、指针、引用或空。返回复合类型的对象更加复杂，而且通常效率低下。

除了最简单的情况外，复合对象的返回是通过将其复制到调用者通过一个隐藏指针指定的位置来实现的。在复制过程中，通常会调用拷贝构造函数；当原始对象被销毁时，会调用析构函数。在简单情况下，编译器可能能够通过在对象的最终位置构造对象来避免调用拷贝构造函数和析构函数，但不要对此抱有太大期望。

返回值，其实是，在被调用函数的栈帧，有一个临时对象，然后将这个临时对象，通过赋值构造函数，在调用者的栈帧上创建对象。而返回值优化则是可以直接在调用者的栈帧上进行构建，从而减少了消耗

当使用引用作为函数参数时，在被调用函数的栈帧上不会为参数创建副本，而是传递实参的内存地址。被调用函数可以通过这个地址直接访问和修改实参。

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

7.17函数尾调用

图形用户界面, 文本, 应用程序, Word

描述已自动生成

7.18递归函数

**递归函数的适用场景**

* **分支数据树结构**：递归函数对于处理分支数据树结构非常有用，因为树结构本身具有递归的特性。例如，在遍历二叉树时，递归可以简洁地实现前序、中序和后序遍历。如果树结构比较宽泛（即每个节点的子节点较多）而不是很深，递归的效率相对较高。
* **非分支递归可被循环替代**：非分支递归（即递归调用中没有分支逻辑）总是可以用循环来替代，并且循环通常效率更高。因为循环不会像递归那样在栈上创建大量的新实例，避免了栈空间的浪费和返回地址预测的问题。

文本

描述已自动生成

表格

描述已自动生成

* **硬件限制**：现代计算机的内存系统通常按照字（word）为单位进行读写操作，不同架构的计算机其字长不同，例如常见的有 32 位（4 字节）和 64 位（8 字节）。CPU 访问内存时，通常是按照特定的地址边界进行的。如果数据存储在符合其自身对齐要求的地址上，CPU 可以在一个时钟周期内完成对该数据的访问；而如果数据没有对齐，CPU 可能需要多个时钟周期才能完成访问。
* **示例**：假设一个 32 位的 CPU 每次从内存读取 4 字节的数据，如果一个 4 字节的 int 类型变量存储在地址为 0x00000000 的位置，CPU 可以在一个时钟周期内将其读取；但如果这个 int 变量存储在地址为 0x00000001 的位置，由于它跨越了两个 4 字节的内存块，CPU 可能需要两个时钟周期才能将其完整读取出来，这就降低了内存访问效率。

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

为优化内存使用和代码紧凑性，建议在结构体或类声明中，将大数组和大对象放在最后，最常用的数据成员放在最前面。若无法将所有成员包含在开头的 128 字节内，应将最常用成员放在该范围内。

**类的数据成员和成员函数实例**

* **数据成员**：每次创建类的新对象时，类的数据成员都会有新的实例。这意味着每个对象都有自己独立的数据成员副本，它们在内存中占据不同的空间，用于存储各自的数据。例如，对于类 S3，每个 S3 对象都有自己的 a 和 b 成员变量。
* **成员函数**：类的成员函数只有一个实例，所有该类的对象共享同一个成员函数代码。这是因为成员函数的代码逻辑是通用的，适用于类的所有对象，不需要为每个对象复制一份函数代码。

**静态成员函数**

* **访问限制**：静态成员函数不能访问非静态数据成员和非静态成员函数。这是因为静态成员函数属于类本身，而不是类的某个对象，它没有 this 指针，无法确定要访问哪个对象的非静态成员。
* **效率优势**：由于静态成员函数不需要 this 指针，减少了参数传递的开销，所以通常比非静态成员函数执行速度更快。如果一个成员函数不需要访问任何非静态成员，将其声明为静态成员函数可以提高效率。例如：

7.31线程

线程用于同时或似乎同时执行两个或多个作业。现代CPU有多个核心，可以同时运行多个线程。当线程数多于CPU内核数时，每个线程将获得前台作业通常为30 ms的时间片，后台作业通常为10 ms的时间片。在每个时间片之后的上下文切换是相当昂贵的，因为所有缓存必须适应新的上下文。

在优化多线程应用程序时，我们必须考虑多线程带来的四种成本：

* **启动和停止线程的成本**：如果一个任务的执行时间与启动和停止线程所需的时间相比很短，就不要将该任务放在单独的线程中执行。
* **任务切换的成本**：如果具有相同优先级的线程数量不超过 CPU 核心数，那么任务切换的成本可以降到最低。
* **线程间同步和通信的成本**：信号量、互斥锁等机制的开销相当大。如果两个线程为了访问同一资源而经常相互等待，那么将它们合并为一个线程可能会更好。在多个线程之间共享的变量必须声明为 volatile。这可以防止编译器将该变量存储在寄存器中，因为寄存器在线程之间是不共享的。
* **不同线程需要独立的存储空间**：任何被多个线程使用的函数或类都不应依赖于静态或全局变量（参见第 28 页的线程局部存储）。每个线程都有自己的栈。如果线程共享同一个缓存，这可能会导致缓存争用。

多线程程序必须使用线程安全的函数。线程安全的函数绝不能使用静态变量