C++性能优化指南

“根据我的经验,对于简单的查找和排序任务而言,最优算法的准备时间很少,即使在小型 数据集上使用它们也能改善性能”

C++代码优化策略总结

1. 用好的编译器并用好编译器
2. 使用更好的算法
3. 使用更好的库
4. 减少内存分配和复制
5. 移除计算
6. 使用更好的数据结构
7. 提高并发性
8. 优化内存管理

影响优化的计算机行为

内存很慢

计算机的主内存相对于它内部的逻辑门和寄存器来说非常慢。将电子从微处理器芯片中注入 相对广阔的一块铜制电路板上的电路,然后将其沿着电路推到几厘米外的内存芯片中,这个 过程所花费的时间为电子穿越微处理器内各个独立的微距晶体管所需时间的数千倍。主内存 太慢,所以桌面级处理器在从主内存中读取一个数据字的时间内,可以执行数百条指令。 优化的根据在于处理器访问内存的开销远比其他开销大,包括执行指令的开销。” 计算机的主内存相对于其内部的逻辑门和寄存器来说非常慢。将电子从芯片中注入相对非法的铜制电路板上的电路，然后将其沿电路推到几厘米外的内存芯片中，这个过程所耗费的时间为电子穿越内部各个独立的微站晶体管所需时间的数千倍。主内存太慢，所以桌面级处理器在从主内存中读取一个数据字的周期，可以执行数百条指令。优化的根据处理器访问内存的开销远比其他开销大，包括执行指令的开销。

内存访问并非以字节为单位

当 C++ 获取一个多字节类型的数据,比如一个 int、double 或者指针时,构成数据的字**可能跨越了两个物理内存字。**这种访问被称为非对齐的内存访问(unaligned memory access)。

C++ 编译器会帮助我们对齐结构体,使每个字段的起 始字节地址都是该字段的大小的倍数。但是这样也会带来相应的问题:结构体的“洞”中 包含了无用的数据。在定义结构体时,对各个数据字段的大小和顺序稍加注意,可以在保 持对齐的前提下使结构体更加紧凑。

处理器乱序多发射

“每一次赋值、函数参数的初始化和函数返回值都会调用一次构造函数,这个函数可能隐 藏了大量的未知代码。 • 有些语句隐藏了大量的计算。从语句的外表上看不出语句的性能开销会有多大。 • 当并发线程共享数据时,同步代码降低了并发量。”

1. 每一次赋值、函数参数的初始化和函数返回值都会调用一次构造函数,这个函数可能隐 藏了大量的未知代码。
2. 有些语句隐藏了大量的计算。从语句的外表上看不出语句的性能开销会有多大。

3. 当并发线程共享数据时,同步代码降低了并发量。”

测量性能

评估代码开销

“访问内存的时间开销远比执行其他指令的开销大。在烤箱和咖 啡机所使用的简单微处理器中,执行一条指令所花费的时间大致包含从内存中读取指令的 每个字节所需要的时间,加上读取指令的输入数据所需的时间,再加上写指令结果的时 间。相比之下,隐藏于内存访问时间之下的解码和执行指令的时间就显得微不足道了。 在桌面级微处理器上,情况就更加复杂了。许多处于不同阶段的指令会被同时执行。读取 指令流的开销可以忽略。不过,访问指令所操作的数据的开销则无法忽略。正是由于这个 原因,读写数据的开销可以近似地看作所有级别的微处理器上的执行指令的相对开销。”

评估循环的开销

1. 评估嵌套循环中的循环次数
2. 评估循环次数为变量的循环的开销
3. 识别隐式循环
4. 识别假循环

优化字符串

字符串是动态分配的

“但字符串内部的字符缓冲区的大小仍然是固定的。任何会使字符串变长的操 作,如在字符串后面再添加一个字符或是字符串,都可能会使字符串的长度超出它内部的 缓冲区的大小。当发生这种情况时,操作会从内存管理器中获取一块新的缓冲区,并将字 符串复制到新的缓冲区中。”

字符串就是值

“由于字符串就是值,因此字符串表达式的结果也是值。如果你使用 s1 = s2 + s3 + s4; 这 条语句连接字符串,那么 s2 + s3 的结果会被保存在一个新分配的临时字符串中。**连接 s4 后的结果则会被保存在另一个临时字符串中。这个值将会取代 s1 之前的值。接着,为第一个临时字符串和 s1 之前的值动态分配的内存将会被释放。这会导致多次调用内存管理器。”**

字符串会进行大量复制

“在 C++11 及之后的版本中,随着“右值引用”和“移动语义”(详见 6.6 节)的出现,使用 它们可以在某种程度上减轻复制的负担。如果一个函数使用“右值引用”作为参数,那么 当实参是一个右值表达式时,字符串可以进行轻量级的指针复制,从而节省一次复制操作”

优化方法

1. 使用复合赋值操作避免临时字符串
2. 通过预留存储空间减少内存的重新分配
3. 消除对参数字符串的复制
4. 使用迭代器消除指针解引用
5. 消除对返回字符串的复制
6. 用字符数组代替字符串

7. 如果有可能，考虑更好的算法

代码清单 4-10 remove_ctrl_erase()：不创建新的字符串，而是修改参数字符串的值作为结果返回

```
std::string remove_ctrl_erase(std::string s) {
    for (size_t i = 0; i < s.length();)
        if (s[i] < 0x20)
            s.erase(i,1);
        else ++i;
    return s;
}
```

这种算法的优势在于，由于 s 在不断地变短，除了返回值时会发生内存分配外，其他情况下都不会再发生内存分配。修改后的函数性能非常棒，测试结果是每次调用耗时 0.81 毫秒，比初始版本的 remove_ctrl() 快了 30 倍。如果在第一次优化中取得了这个优异的结果，开发人员可能认为自己胜利了，然后退出优化战场，不会考虑如何进一步优化。有时候，选择一种不同的算法后程序会变得更快；即使没有变快，也可能会变得比原来更容易优化。

8. 使用更好的编译器
9. 使用更好的字符串库
10. 使用更好的内存分配器

优化动态分配内存的变量

1. 使用智能指针管理动态分配的内存
2. 使用静态数据结构
    1. 用std::array替代std::vector
    2. 在栈上创建大块缓冲区
    3. 静态地创建链式数据结构
    4. 在数组中创建二叉树
    5. 用环形缓冲区代替双端队列
3. 不要随便共享所有权(如shared_ptr)
4. 减少动态变量的重新分配
    1. 预分配动态变量防止重新分配
    2. 在循环外创建动态变量
5. 移除无谓的复制

开发人员在寻找一段热点代码中的优化机会时，必须特别注意赋值和声明，因为在这些地方可能会发生昂贵的复制。实际上，复制可能会发生于以下任何一种情况下：

- 初始化（调用构造函数）
- 赋值（调用赋值运算符）
- 函数参数（每个参数表达式都会被移动构造函数或复制构造函数复制到形参中）
- 函数返回（调用移动构造函数或复制构造函数，甚至可能会调用两次）
- 插入一个元素到标准库容器中（会调用移动构造函数或复制构造函数复制元素）
- 插入一个元素到 vector 中（如果需要重新为 vector 分配内存，那么所有的元素都会通过移动构造函数或复制构造函数复制到新的 vector 中）

1. 在类定义中禁止不希望发生的复制
2. 移除函数调用上的复制
3. 移除函数返回上的复制

   有一种方法可以移除函数内部的类实例的构造以及从函数返回时发生的两次复制构造（或是等价于复制构造函数的赋值运算符）。这需要开发人员手动编码实现，所以其结果肯定比寄希望于编译器在给定的情况下进行 RVO 要好。这种方法就是不用 `return` 语句返回值，而是在函数内更新引用参数，然后通过**输出参数**返回该引用参数：

   ```
   void scalar_product(
       std::vector<int> const& v,
       int c,
       vector<int>& result) {
       result.clear();
       result.reserve(v.size());
       for (auto val : v)
           result.push_back(val * c);
   }
   ```

4. 免复制库
5. 切割数据结构

   切割（slice）是一种编程惯用法，它指的是一个变量指向另外一个变量的一部分。例如，C++17 中推荐的 `string_view` 类型就指向一个字符串的子字符串，它包含了一个指向子字符串开始位置的 `char*` 指针以及到子字符串的长度。

   被切割的对象通常都是小的、容易复制的对象，将其内容复制至子数组或子字符串中而分

   配存储空间的开销不大。如果被分割的数据结构为被共享的指针所有，那么切割是完全安全的。但是经验告诉我，被切割的对象的生命是短暂的。它们在短暂地实现了存在的意义后，就会在被切割的数据结构能够被销毁前超出它们的作用域。例如，`string_view` 使用一个无主指针指向字符串。

6. 实现移动语义

7. 使用扁平数据结构

当一个数据结构中的元素被存储在连续的存储空间中时，我们称这个数据结构为**扁平的**。相比于通过指针链接在一起的数据结构，扁平数据结构具有显著的性能优势。

- 相比于通过指针链接在一起的数据结构，创建扁平数据结构实例时调用内存管理器的开销更小。有些数据结构（如 `list`、`deque`、`map`、`unordered_map`）会创建许多动态变量，而其他数据结构（如 `vector`）则较少。正如将在第 10 章反复看到的，即使是相似的操作具有相同的大 O 性能开销，`std::vector` 和 `std::array` 等扁平数据结构也有很大的优势。
- `std::array` 和 `std::vector` 等扁平数据结构所需的内存比 `list`、`map`、`unordered_map` 等基于节点的数据结构少，因为在基于节点的数据结构中存在着链接指针的开销。即使所消耗的总字节数没有问题，紧凑的数据结构仍然有助于改善缓存局部性。扁平数据结构在局部缓存性上的优势使得它们更加高效。
- 以前常常需要用到的技巧，诸如用智能指针组成 `vector` 或是 `map` 来存储不可复制的对象，在 C++11 中的移动语义出现后已经不再需要了。移动语义移除了在分配智能指针和它所指向的对象的存储空间时产生的显著的运行时开销。

优化热点语句

从循环中移除代码

1. 缓存循环结束条件值
2. 使用更高效的循环语句

以下是 C++ 中 `for` 循环语句的声明语法：

```
for（初始化表达式 ； 循环条件 ； 继续表达式 ） 语句
```

粗略地讲，`for` 循环会被编译为如下代码：

```
    初始化表达式 ;
L1: if ( ! 循环条件 ) goto L2;
    语句 ;
    继续表达式 ;
    goto L1;
L2:
```

`for` 循环必须执行两次 `jump` 指令：一次是当循环条件为 `false` 时；另一次则是在计算了继续表达式之后。这些 `jump` 指令可能会降低执行速度。C++ 还有一种使用不那么广泛的、称为 `do` 的更简单的循环形式，它的声明语法如下：

```
do 语句 while ( 循环条件 ) ;
```

粗略地讲，`do` 循环会被编译为如下代码：

```
L1: 控制语句
    if ( 循环条件 ) goto L1;
```

3. 用递减代替递增
4. 从循环中移除不变性代码
5. 从循环中移除无谓的函数调用

6. 从循环中移除隐含的函数调用

普通的函数调用很容易识别，它们有函数名，在圆括号中有参数表达式列表。C++ 代码还可能会隐式地调用函数，而没有这种很明显的调用语句。当一个变量是以下类型之一时就可能会发生这种情况：

- 声明一个类实例（调用构造函数）
- 初始化一个类实例（调用构造函数）
- 赋值给一个类实例（调用赋值运算符）
- 涉及类实例的计算表达式（调用运算符成员函数）
- 退出作用域（调用在作用域中声明的类实例的析构函数）
- 函数参数（每个参数表达式都会被复制构造到它的形参中）
- 函数返回一个类的实例（调用复制构造函数，可能是两次）

- 向标准库容器中插入元素（元素会被移动构造或复制构造）
- 向矢量中插入元素（如果矢量重新分配了内存，那么所有的元素都需要被移动构造或是复制构造）

7. 从循环中移除昂贵的、缓慢改变的调用
8. 循环函数变为函数中的循环：减少函数调用次数

函数调用的开销

虚函数的开销

“由于虚函数调用会从多个函数体中选择一个执行,调用虚函数的代码会解引指向类实例的 指针,来获得指向虚函数表的指针。这段代码会为虚函数表加上索引(也就是说,代码会 在虚函数表上加上一段小的整数偏移量并解引该地址)来得到函数的执行地址。因此,实 际上这里会为所有的虚函数调用额外地加载两次非连续的内存,每次都会增加高速缓存未 命中的几率和发生流水线停顿的几率。虚函数的另一个问题是编译器难以内联它们。编译 128 | 第 7 章 器只有在它能同时访问函数体和构造实例的代码(这样编译器才能决定调用虚函数的哪个 函数体)时才能内联它们。”

继承中的成员函数开销

3. 继承中的成员函数调用

当一个类继承另一个类时，继承类的成员函数可能需要进行一些额外的工作。

继承类中定义的虚成员函数

如果继承关系最顶端的基类没有虚成员函数，那么代码必须要给 this 类实例指针加上一个偏移量，来得到继承类的虚函数表，接着会遍历虚函数表来获取函数执行地址。这些代码会包含更多的指令字节，而且这些指令通常都比较慢，因为它们会进行额外的计算。这种开销在小型嵌入式处理器上非常显著，但是在桌面级处理器上，指令级别的并发掩盖了大部分这种额外的开销。

多重继承的继承类中定义的成员函数调用

代码必须向 this 类实例指针中加上一个偏移量来组成指向多重继承类实例的指针。这种开销在小型嵌入式处理器上非常显著，但是在桌面级处理器上，指令级别的并发掩盖了大部分这种额外的开销。

多重继承的继承类中定义的虚成员函数调用

对于继承类中的虚成员函数调用，如果继承关系最顶端的基类没有虚成员函数，那么代码必须要给 this 类实例指针加上一个偏移量来得到继承类的虚函数表，接着会遍历虚函数表来获取函数执行地址。代码还必须向 this 类实例指针加上潜在的不同的偏移量来组成继承类的类实例指针。这种开销在小型嵌入式处理器上非常显著，但是在桌面级处理器上，指令级别的并发掩盖了大部分这种额外的开销。

虚多重继承

为了组成虚多重继承类的实例的指针，代码必须解引类实例中的表，来确定要得到指向虚多重继承类的实例的指针时需要加在类实例指针上的偏移量。如前所述，当被调用的函数是虚函数时，这里也会产生额外的间接开销。

函数指针的开销

- 运行时才会被调用；必须解引用才能获取函数执行地址；编译器一般不会内联这些函数

C++ 提供了函数指针，这样当通过函数指针调用函数时，代码可以在运行时选择要执行的函数体。除了基本的函数调用和返回开销外，这种机制还会产生其他额外的开销。

函数指针（指向非成员函数和静态成员函数的指针）

C++ 允许在程序中定义指向函数的指针。程序员可以通过函数指针显式地选择一个具有特定签名（由参数列表和返回类型组成）的非成员函数。当函数指针被解引后，这个函数将会在运行时会被调用。通过将一个函数赋值给函数指针，程序可以显式地通过函数指针选择要调用的函数。

代码必须解引指针来获取函数的执行地址。编译器也不太可能会内联这些函数。

成员函数指针

成员函数指针声明同时指定了函数签名和解释函数调用的上下文中的类。程序通过将函数赋值给函数指针，显式地选择通过成员函数指针调用哪个函数。

成员函数指针有多种表现形式，一个成员函数只能有一种表现形式。它必须足够通用才

能够在以上列举的各种复杂的场景下调用任意的成员函数。我们有理由认为一个成员函数指针会出现最差情况的性能。

函数调用开销总结

“因此,C 风格的不带参数的 void 函数的调用开销是最小的。如果能够内联它的话,就没有 开销;即使不能内联,开销也仅仅是两次内存读取加上两次程序执行的非局部转移 。 如果基类没有虚函数,而虚函数在多重虚拟继承的继承类中,那么这是最坏的情况。不过 幸运的是,这种情况非常罕见。在这种情况下,代码必须解引类实例中的函数表来确定加 到类实例指针上的偏移量,构成虚拟多重继承函数的实例的指针,接着解引该实例来获取 虚函数表,最后索引虚函数表得到函数执行地址。 此时,读者可能会惊讶函数调用的开销居然如此之大,抑或是惊叹 C++ 居然如此高效地实 现了这么复杂的特性。这两种看法都是合理的。需要理解的是正是有了函数调用开销,才 有优化的机会。坏消息是除非函数会被频繁地调用,否则移除一处非连续内存读取并不足 以改善性能;好消息则是**分析器会直接指出调用最频繁的函数,让开发人员能够快速地集 中精力于最佳优化对象。**”

用模板在编译时选择实现

“C++ 模板特化是另外一种在编译时选择实现的方法。利用模板,开发人员可以创建具有通 用接口的类群,但是它们的行为取决于模板的类型参数。模板参数可以是任意类型——**具 有自己的一组成员函数的类类型或是具有内建运算符的基本类型。**因此,存在两种接口: 模板类的 public 成员,以及由在模板参数上被调用的运算符和函数所定义的接口。抽象基类中定义的接口是非常严格的,继承类必须实现在抽象基类中定义的所有函数。而通过模板定义的接口就没有这么严格了。**只有参数中那些实际会被模板的某种特化所调用的函数才需要被定义。** 模板的特性是一把双刃剑:一方面,即使开发人员在某个模板特化中忘记实现接口了,编 译器也不会立即报出错误消息;但另一方面,开发人员也能够选择不去实现那些在上下文 中没被用到的函数。 从性能优化的角度看,**多态类层次与**

模板实例之间的最重要的区别是,通常在编译时整个 模板都是可用的。在大多数用例下,C++ 都会内联函数调用,用多种方法改善程序性能 模板编程提供了一种强力的优化手段。对于那些不熟悉模板的开发人员来说,需要学习如 何高效地使用 C++ 的这个特性。”

避免使用PIMPL惯用法

> PIMPL 是“Pointer to IMPLementation”的缩写，它是一种用作**编译防火墙**——一种防止修改一个头文件会触发许多源文件被重编译的机制——的编程惯用法。20 世纪 90 年代是 C++ 的快速成长期，在那时使用 PIMPL 是合理的，因为在那个年代，大型程序的编译时间是以小时为单位计算的。下面是 PIMPL 的工作原理。

在运行时情况就不同了。PIMPL 给程序带来了延迟。之前 `BigClass` 中的成员函数可能会被内联，而现在则会发生一次成员函数调用。而且，现在每次成员函数调用都会调用 `Impl` 的成员函数。使用了 PIMPL 的工程往往会在很多地方使用它，导致形成了多层嵌套函数调用。更甚者，这些额外的函数调用层次使得调试变得更加困难。

2016 年，PIMPL 已经不是必需的了，因为编译时间可能已经减少至了 20 世纪 90 年代的 1%。而且，即使是在 20 世纪 90 年代，也只有当 `BigClass` 是一个非常大的类，依赖于许多头文件时，才需要使用 PIMPL。这样的类违背了许多面向对象编程原则。采用将 `BigClass` 分解，使接口功能更加集中的方法，可能与 PIMPL 同样有效。

使用静态成员函数取代成员函数

每次对成员函数的调用都有一个额外的隐式参数：指向成员函数被调用的类实例的 `this` 指针。通过对 `this` 指针加上偏移量可以获取类成员数据。虚成员函数必须解引 `this` 指针来获得虚函数表指针。

有时，一个成员函数中的处理仅仅使用了它的参数，而不用访问成员数据，也不用调用其他的虚成员函数。在这种情况下，`this` 指针没有任何作用。

我们应当将这样的成员函数声明为静态函数。静态成员函数不会计算隐式 `this` 指针，可以通过普通函数指针，而不是开销更加昂贵的成员函数指针找到它们（请参见 7.2.1 节中的“函数指针的开销”）。

优化表达式

1. 简化表达式

C++ 会严格地以运算符的优先级和可结合性的顺序来计算表达式。只有像 `((a*b)+(a*c))` 这样书写表达式时才会进行 `a*b+a*c` 的计算，因为 C++ 的优先级规则规定乘法的优先级高于加法。C++ 编译器绝对不会使用分配律将表达式重新编码为像 `a*(b+c)` 这样的更高效的形式。只有像 `((a+b)+c)` 这样书写表达式才会进行 `a+b+c` 的计算，因为 + 运算符具有左结合性。编译器绝对不会重写表达式为 `(a+(b+c))`，尽管在进行整数和实数数学计算时其结果并不会发生改变。

C++ 之所以让程序员手动优化表达式，是因为 C++ 的 `int` 类型的模运算并非是整数的数学运算，C++ 的 `float` 类型的近似计算也并非真正的数学运算。C++ 必须给予程序员足够的权力来清晰地表达他的意图，否则编译器会对表达式进行重排序，从而导致控制流程发生各种变化。这意味着开发人员必须尽可能使用最少的运算符来书写表达式。

用于计算多项式的霍纳法则（Horner Rule）证明了以一种更高效的形式重写表达式有多么厉害。尽管大多数 C++ 开发人员并不会每天都进行多项式计算，但是我们都很熟悉它。

多项式 $y = ax^3 + bx^2 + cx + d$ 在 C++ 中可以写为：

```
y = a*x*x*x + b*x*x + c*x + d;
```

这条语句将会执行 6 次乘法运算和 3 次加法运算。我们可以根据霍纳法则重复地使用分配律来重写这条语句：

```
y = (((a*x + b)*x) + c)*x + d;
```

2. 将常量组合在一起

因此，我们应当总是用括号将常量表达式组合在一起，或是将它们放在表达式的左端，或者更好的一种的做法是，将它们独立出来初始化给一个常量，或者将它们放在一个常量表达式（constexpr）函数中（如果你的编译器支持 C++11 的这一特性）。这样编译器能够在编译时高效地计算常量表达式。

3. 使用更高效的运算符

有些数学运算符在计算时比其他运算符更低效。例如，如今，所有处理器（除了最小型的处理器）都可以在一个内部时钟周期中执行一次位移或是加法操作。某些专业的数字信号处理器芯片有单周期乘法器，但是对于 PC，乘法是一种类似于我们在小学学到的十进制乘法的迭代计算。除法是一种更复杂的迭代处理。这种开销结构为性能优化提供了机会。

例如，整数表达式 `x*4` 可以被重编码为更高效的 `x<<2`。任何差不多的编译器都可以优化这个表达式。但是如果表达式是 `x*y` 或 `x*func()` 会怎样呢？许多情况下，编译器都无法确定 `y` 或 `func()` 的返回值一定是 2 的幂。这时就需要依靠程序员了。如果其中一个参数可以用指数替换掉 2 的幂，那么开发人员就可以重写表达式，用位移运算替代乘法运算。

另一种优化是用位移运算和加法运算替代乘法。例如，整数表达式 `x*9` 可以被重写为 `x*8+x*1`，进而可以重写为 `(x<<3)+x`。当常量运算子中没有许多置为 1 的位时，这种优化最有效，因为每个置为 1 的位都会扩展为一个位移和加法表达式。在拥有指令缓存和流水线执行单元的桌面级或是手持级处理器上，以及在长乘法被实现为子例程调用的小型处理器上，这种优化同样有效。与所有性能优化方法一样，我们必须测试性能结果来确保在某种处理器上它确实提高了性能，但通常情况下确实都是这样的。

4. 使用整数计算替代浮点型计算

5. 双精度类型可能会比浮点型更快

为什么会出现这种现象呢？ Visual C++ 生成的浮点型指令会引用老式的“x87 FPU coprocessor”寄存器栈。在这种情况下，所有的浮点计算都会以 80 位格式进行。当单精度 `float` 和双精度 `double` 值被移动到 FPU 寄存器中时，它们都会被加长。对 `float` 进行转换的时间可能比对 `double` 进行转换的时间更长。

有多种编译浮点型计算的方式。在 x86 平台上，使用 SSE 寄存器允许直接以四种不同大小完成计算。使用了 SSE 指令的编译器的行为可能会与为非 x86 处理器进行编译的编译器不同。

6. 用闭形式替代迭代计算

代码清单 7-21 展示了一个判断 x 是否是 2 的幂的闭形式的函数。

代码清单 7-21　判断一个整数是否是 2 的幂的闭形式

```
inline bool is_power_2_closed(unsigned n) {
    return ((n != 0) && !(n & (n - 1)));
}
```

使用这个修改后的函数进行测试的结果是耗时 238 毫秒，比之前的版本快了 2.3 倍。其实还有更快的方法。Rick Regan 在他的网页（http://www.exploringbinary.com/ten-ways-to-check-if-an-integer-is-a-power-of-two-in-c/）上记录了 10 种方法，而且都附有时间测量结果。

7. 优化控制流程惯用法

1. 用switch替代if-else
2. 用虚函数替代switch或if

在 C++ 出现之前，如果开发人员想要在程序中引入多态行为，那么他们必须编写一个带有标识变量的结构体或是联合体，然后通过这个标识变量来辨别出当前使用的是哪个结构体或是联合体。程序中应该会有很多类似下面的代码：

```
if (p->animalType == TIGER) {
    tiger_pounce(p->tiger);
}
else if (p->animalType == RABBIT) {
    rabit_hop(p->rabbit);
}
else if (...)
```

经验丰富的开发人员都知道这个反模式是面向对象编程的典型代表。但是新手开发人员要想熟练掌握面向对象思想是需要时间的。我在很多软件产品中看到过下面这样不纯粹的面向对象的 C++ 代码：

```
Animal::move() {
    if (this->animalType == TIGER) {
        pounce();
    }
    else if (this->animalType == RABBIT) {
        hop();
    }
    else if (...)
...
}
```

3. 使用无开销的异常处理

C++11 弃用了传统的异常规范。

在 C++11 中引入了一种新的异常规范，称为 noexcept。声明一个函数为 noexcept 会告诉编译器这个函数不可能抛出任何异常。如果这个函数抛出了异常，那么如同在 throw() 规范中一样，terminate() 将会被调用。不过不同的是，编译器要求将移动构造函数和移动赋值语句声明为 noexcept 来实现移动语义（请参见 6.6 节中有关移动语义的讨论）。在这些函数上的 noexcept 规范的作用就像是发表了一份声明，表明对于某些对象而言，移动语义比强异常安全保证更重要。我知道这非常晦涩。

优化战争故事

下面是一句我称为“printf() 不是你的朋友”的格言。

“Hello, World”可能是最简单的 C++（或 C）程序了：

```
# include <stdio.h>
int main(int, char**) {
    printf("Hello, World!\n");
    return 0;
}
```

这段程序包含了多少个可执行字节呢？如果你猜“大约 50 或 100 字节”，那么你弄错了两个数量级。在我编写的一个嵌入式控制器中，这段程序占用了 8KB。而且这仅仅是代码的大小，不包含符号表、加载器信息和其他任何东西。

下面这段代码完成的工作与之前的代码相同：

```
# include <stdio.h>
int main(int, char**) {
    puts("Hello, World!");
    return 0;
}
```

这段程序实际上与之前的程序是一样的，只是使用了 puts() 来输出字符串，而没有用 printf()。但是第二个程序只占用了大约 100 字节。导致程序大小区别这么大的原因是什么呢？

printf() 正是罪魁祸首。printf() 能够以三四种格式打印各种类型的数据。它能够将某种格式的字符串解释为读取可变数量的参数。printf() 自身就是一个大函数，但是真正让它变大的原因是，它引入了格式化各种基本类型的标准库函数。在我的嵌入式控制器上，情况更加糟糕，由于处理器没有实现硬件浮点类型计算，因此我使用了一个函数扩展库。事实上，printf() 是上帝函数的典型代表——一个吸收了 C 运行时库，可以做许多事情的函数。

另一方面，puts() 只是将字符串放到标准输出中而已。它的内部非常简单，而且它不会链接标准库中的许多函数。

优化查找和排序

优化std::map的查找

1. 以固定长度的字符数组作为std::map的键
2. 以C风格的字符串组作为键使用std::map
3. 当键就是值的时候,使用map的表亲std::set
4. 使用<algorithm>头文件优化算法

5. 以序列容器作为被查找的键值对表

相比于 std::map 或它的表亲 std::set，有几个理由使得选择序列容器实现键值对表更好：序列容器消耗的内存比 map 少，它们的启动开销也更小。标准库算法的一个非常有用的特性是它们能够遍历任意类型的普通数组，因此，它们能够高效地查找静态初始化的结构体的数组。这样可以移除所有启动表的开销和销毁表的开销。而且，诸如 MISRA C++（http://www.misra-cpp.com）等编码标准都禁止或是限制了动态分配内存的数据结构的使用。因此，使用序列容器是一种能够高效地在这些环境中进行查找的解决方案。

本节中的示例代码使用了如下定义的结构体：

```
struct kv { // (键,值)对
    char const* key;
    unsigned    value; // 可以是任何类型
};
```

由这些键值对构成的静态数组的定义如下：

```
kv names[] = {// 以字母顺序排序
    { "alpha",   1 },   { "bravo",    2 },
    { "charlie", 3 },   { "delta",    4 },
    { "echo",    5 },   { "foxtrot",  6 },
    { "golf",    7 },   { "hotel",    8 },
    { "india",   9 },   { "juliet",  10 },
    { "kilo",   11 },   { "lima",    12 },
    { "mike",   13 },   { "november",14 },
    { "oscar",  15 },   { "papa",    16 },
    { "quebec", 17 },   { "romeo",   18 },
    { "sierra", 19 },   { "tango",   20 },
    { "uniform",21 },   { "victor",  22 },
    { "whiskey",23 },   { "x-ray",   24 },
    { "yankee", 25 },   { "zulu",    26 }
};
```

names 数组的初始化是静态集合初始化。C++ 编译器会在编译时为 C 风格的结构体创建初始化数据。创建这样的数组不会有任何运行时开销。

我们通过在这张小型表中查找 26 个键和 27 个不存在于表中的字符串来测量这些算法。为了得到可测量的时间，我们会重复 100 万次这 53 次查找。这个测试与上一节中对 std::map 进行的测试是相同的。

标准库容器类提供了 begin() 和 end() 成员函数，这样程序就能够得到一个指向待查找范围的迭代器。C 风格的数组更加简单，通常没有提供这些函数。不过，我们可以通过用一点模板“魔法”提供类型安全的模板函数来实现这个需求。由于它们接收一个数组类型作为参数，数组并不会像通常那样退化为一个指针：

“斯特潘诺夫3的抽象惩罚”

但是使用标准库的这种极其强大和通用的机制是有开销的。即使标准库算法具有优秀的性 能,它也往往无法与最佳手工编码的算法匹敌。这可能是因为模板代码中的缺点或是编译 器设计中的缺点,抑或是因为标准库代码需要能够工作于通用情况下(如只使用 < 函数运 算符,且不使用 strcmp())。这种开销可能会导致开发人员不得不自己去编写那些确实非 常重要的查找算法。 这个存在于标准算法和手工编写的优秀算法之间的鸿沟被称为“斯特潘诺夫的抽象惩罚”, 它是以亚历山大 • 斯特潘诺夫的名字命名的。在亚历山大 • 斯特潘诺夫设计出了初始版本 的标准库算法和容器类后,一度没有编译器能够编译它们。相对于手动编码的解决方案, 斯特潘诺夫的抽象惩罚是通用解决方案无法避免的开销,它也是使用 C++ 标准库算法这样 的能够提高生产力的工具的代价。这并非一件坏事,但却是当开发人员需要提高程序性能 时必须注意的事情。”

优化数据结构

- 斯特潘诺夫的标准模板库是第一个可复用的高效容器和算法库。
- 各容器类的大 *O* 标记性能并不能反映真实情况。有些容器比其他容器快许多倍。
- 在进行插入、删除、遍历和排序操作时 `std::vector` 都是最快的容器。
- 使用 `std::lower_bound` 查找有序 `std::vector` 的速度可以与查找 `std::map` 的速度相匹敌。
- `std::deque` 只比 `std::list` 稍快一点。
- `std::forward_list` 并不比 `std::list` 更快。
- 散列表 `std::unordered_map` 比 `std::map` 更快，但是相比所受到的开发人员的器重程度，它并没有比 `std::map` 快上一个数量级。
- 互联网上有丰富的类似标注库容器的容器资源。

优化I/O

- 不论你是在哪个网站上看到的，互联网上的“快速”文件 I/O 代码不一定快。
- 增大 `rdbuf` 的大小可以让读取文件的性能提高几个百分点。
- 我测试到的最快的读取文件的方法是预先为字符串分配与文件大小相同的缓冲区，然后调用 `std::streambuf::sgetn()` 函数填充字符串缓冲区。
- `std::endl` 会刷新输出。如果你并不打算在控制台上输出，那么它的开销是昂贵的。
- `std::cout` 是与 `std::cin` 和 `stdout` 捆绑在一起的。打破这种连接能够改善性能。

优化并发

优化多线程C++程序

1. “实现任务队列和线程池”
2. “用std::async替代std::thread”
3. “创建与核心数量一样多的可执行线程”
4. “单独的线程中执行I/O”
5. “移除启动和停止代码”

优化内存管理

- 相比于内存管理器，在其他地方看看有没有可能会带来更好性能改善效果的优化机会。
- 对几个大型开源程序的研究表明，替换默认内存管理器对程序整体运行速度的性能提升最多只有 30%。
- 为申请相同大小内存块的请求分配内存的内存管理器是很容易编写的，它的运行效率也很高。
- 同一个类的实例的分配内存的请求所申请的内存的大小是一样的。
- 可以在类级别重写 `new()` 运算符。
- 标准库容器类 `std::list`、`std::map`、`std::multimap`、`std::set` 和 `std::multiset` 都从许多同等的节点中创建数据结构。
- 标准库容器接收一个 `Allocator` 作为参数，与类专用 `new()` 运算符一样，它也允许自定义内存管理。
- 编写一个自定义的内存管理器或分配器可以提高程序性能，但相比于移除对内存管理器的调用等其他优化方法，它的效果没有那么明显。