

C++ 工程实践经验谈

陈硕 (giantchen@gmail.com)

最后更新 2012-4-20

版权声明

本作品采用 “Creative Commons 署名 -非商业性使用 -禁止演绎 3.0 Unported 许可协议 (cc by-nc-nd)” 进行许可。 <http://creativecommons.org/licenses/by-nc-nd/3.0/>

内容一览

1	慎用匿名 <code>namespace</code>	3
2	不要重载全局 <code>::operator new()</code>	7
3	采用有利于版本管理的代码格式	14
4	二进制兼容性	24
5	避免使用虚函数作为库的接口	29
6	动态库的接口的推荐做法	37
7	以 <code>boost::function</code> 和 <code>boost::bind</code> 取代虚函数	41
8	带符号整数的除法与余数	48
9	用异或来交换变量是错误的	56
10	在单元测试中 <code>mock</code> 系统调用	63
11	<code>iostream</code> 的用途与局限	68
12	值语义与数据抽象	97
13	再探 <code>std::string</code>	115
14	用 STL <code>algorithm</code> 秒杀几道算法面试题	123
15	C++ 编译链接模型精要	132
16	Zero overhead 原则	170

说明

这是我的博客上关于 C++ 的文章的合集。最新版可从陈硕博客的置顶文章中下载，地址见本页右下角。本系列文章适用于 Linux 操作系统，x86/amd64 硬件平台，g++ 4.x 编译器，服务端开发。

<http://blog.csdn.net/Solstice/archive/2011/02/24/6206154.aspx>

1 慎用匿名 namespace

匿名 namespace (anonymous namespace 或称 unnamed namespace) 是 C++ 语言的一项非常有用的功能，其主要目的是让该 namespace 中的成员（变量或函数）具有独一无二的全局名称，避免名字碰撞 (name collisions)。一般在编写 .cpp 文件时，如果需要写一些小的 helper 函数，我们常常会放到匿名 namespace 里。muduo 0.1.7 中的 muduo/base/Date.cc 和 muduo/base/Thread.cc 等处就用到了匿名 namespace。

我最近在工作中遇到并重新思考了这一问题，发现匿名 namespace 并不是多多益善。

1.1 C 语言的 static 关键字的两种用法

C 语言的 static 关键字有两种用途：

1. 用于函数内部修饰变量，即函数内的静态变量。这种变量的生存期长于该函数，使得函数具有一定的“状态”。使用静态变量的函数一般是不可重入的，也不是线程安全的，比如 strtok(3)。
2. 用在文件级别（函数体之外），修饰变量或函数，表示该变量或函数只在本文件可见，其他文件看不到也访问不到该变量或函数。专业的说法叫“具有 internal linkage”（简言之：不暴露给别的 translation unit）。

C 语言的这两种用法很明确，一般也不容易混淆。

1.2 C++ 语言的 static 关键字的四种用法

由于 C++ 引入了 class，在保持与 C 语言兼容的同时，static 关键字又有了两种新用法：

3. 用于修饰 class 的数据成员，即所谓“静态成员”。这种数据成员的生存期大于 class 的对象（实例/instance）。静态数据成员是每个 class 有一份，普通数据成员是每个 instance 有一份，因此也分别叫做 class variable 和 instance variable。

4. 用于修饰 `class` 的成员函数，即所谓“静态成员函数”。这种成员函数只能访问 `class variable` 和其他静态程序函数，不能访问 `instance variable` 或 `instance method`。

当然，这几种用法可以相互组合，比如 C++ 的成员函数（无论 `static` 还是 `instance`）都可以有其局部的静态变量（上面的用法 1）。对于 `class template` 和 `function template`，其中的 `static` 对象的真正个数跟 `template instantiation`（模板具现化）有关，相信学过 C++ 模板的人不会陌生。

可见在 C++ 里 `static` 被 `overload` 了多次。匿名 `namespace` 的引入是为了减轻 `static` 的负担，它替换了 `static` 的第 2 种用途。也就是说，在 C++ 里不必使用文件级的 `static` 关键字，我们可以用匿名 `namespace` 达到相同的效果。（其实严格地说，`linkage` 或许稍有不同，这里不展开讨论了。）

1.3 匿名 namespace 的不利之处

在工程实践中，匿名 `namespace` 有两大不利之处：

1. 匿名 `namespace` 中的函数是“匿名”的，那么在确实需要引用它的时候就比较麻烦。

比如在调试的时候不便给其中的函数设断点，如果你像我一样使用的是 `gdb` 这样的文本模式 `debugger`；又比如 `profiler` 的输出结果也不容易判别到底是哪个文件中的 `calculate()` 函数需要优化。

2. 使用某些版本的 `g++` 时，同一个文件每次编译出来的二进制文件会变化。

比如说拿到一个会发生 `core dump` 的二进制可执行文件，无法确定它是由哪个 `revision` 的代码编译出来的。毕竟编译结果不可复现，具有一定的随机性。（当然，在正式场合，这应该由软件配置管理 (SCM) 流程来解决。）

另外这也可能让某些 `build tool` 失灵，如果该工具用到了编译出来的二进制文件的 MD5 的话。

考虑下面这段简短的代码 (`anon.cc`):

```

namespace
{
    void foo()
    {
    }
}

int main()
{
    foo();
}

```

anon.cc

anon.cc

对于问题 1： gdb 的 <tab> 键自动补全功能能帮我们设定断点，不是什么大问题。前提是你知道那个 “(anonymous namespace)::foo()” 正是你想要的函数。

```

$ gdb ./a.out
GNU gdb (GDB) 7.0.1-debian

(gdb) b '<tab>
(anonymous namespace)      __data_start          _end
(anonymous namespace)::foo() __do_global_ctors_aux    _fini
_DYNAMIC                   __do_global_dtors_aux  _init
_GLOBAL_OFFSET_TABLE_     __dso_handle         _start
_IO_stdin_used            __gxx_personality_v0  anon.cc
__CTOR_END__              __gxx_personality_v0@plt call_gmon_start
__CTOR_LIST__             __init_array_end     completed.6341
__DTOR_END__              __init_array_start   data_start
__DTOR_LIST__             __libc_csu_fini      dtor_idx.6343
__FRAME_END__             __libc_csu_init      foo
__JCR_END__               __libc_start_main    frame_dummy
__JCR_LIST__              __libc_start_main@plt int
__bss_start               _edata               main

(gdb) b '(<tab>
anonymous namespace)      anonymous namespace)::foo()

(gdb) b '(anonymous namespace)::foo()'
Breakpoint 1 at 0x400588: file anon.cc, line 4.

```

麻烦的是，如果两个文件 anon.cc 和 anonlib.cc 都定义了匿名空间中的 foo() 函数（这不会冲突），那么 gdb 无法区分这两个函数，你只能给其中一个设断点。或者你使用 **文件名: 行号** 的方式来分别设断点。（从技术上，匿名 namespace 中的函数是 **weak text**，链接的时候如果发生符号重名，linker 不会报错。）

从根本上解决的办法是使用普通具名 namespace，如果怕重名，可以把源文件名（必要时加上路径）作为 namespace 名字的一部分。

对于问题 2： 把 anon.cc 编译两次，分别生成 a.out 和 b.out：

```
$ g++ -g -o a.out anon.cc

$ g++ -g -o b.out anon.cc

$ md5sum a.out b.out
0f7a9cc15af7ab1e57af17ba16afcd70 a.out
8f22fc2bbfc27beb922aefa97d174e3b b.out

$ g++ --version
g++ (GCC) 4.2.4 (Ubuntu 4.2.4-1ubuntu4)

$ diff -u <(nm a.out) <(nm b.out)
--- /dev/fd/63  2011-02-15 22:27:58.960754999 +0800
+++ /dev/fd/62  2011-02-15 22:27:58.960754999 +0800
@@ -2,7 +2,7 @@
 0000000000600940 d _GLOBAL_OFFSET_TABLE_
 0000000000400634 R _IO_stdin_used
                w _Jv_RegisterClasses
-0000000000400538 t _ZN36_GLOBAL__N_anon.cc_00000000_E2CEE513fooEv
+0000000000400538 t _ZN36_GLOBAL__N_anon.cc_00000000_CB51498D3fooEv
 0000000000600748 d __CTOR_END__
 0000000000600740 d __CTOR_LIST__
 0000000000600758 d __DTOR_END__
```

由上可见，g++ 4.2.4 会随机地给匿名 namespace 生成一个惟一的名称（foo() 函数的 mangled name 中的 E2CEE51 和 CB51498D 是随机的），以保证名称不冲突。也就是说，同样的源文件，两次编译得到的二进制文件内容不相同，这有时候会造成问题或困惑。

这可以用 gcc 的 -frandom-seed 参数解决，具体见文档。

这个现象在 gcc 4.2.4 中存在（之前的版本估计类似），在 gcc 4.4.5 中不存在。

1.4 替代办法

如果前面的“不利之处”给你带来困扰，解决办法也很简单，就是使用普通具名 namespace。当然，要起一个好的名称，比如 boost 里就常常用 boost::detail 来放那些“不应该暴露给客户，但又不得不放到头文件里”的函数或 class。

总而言之，匿名 namespace 没什么大问题，使用它也不是什么过错。万一它碍事了，可以用普通具名 namespace 替代之。

2 不要重载全局 `::operator new()`

本文只考虑 Linux x86 平台，服务端开发（不考虑 Windows 的跨 DLL 内存分配释放问题）。本文假定读者知道 `::operator new()` 和 `::operator delete()` 是干什么的，与通常用的 `new/delete` 表达式有区别和联系，这方面的知识可参考侯捷先生的文章《池内春秋》^[1]，或者这篇文章¹。

C++ 的内存管理是个老生常谈的话题，我在《当析构函数遇到多线程》² 第 7 节“插曲：系统地避免各种指针错误”中简单回顾了一些常见的问题以及在现代 C++ 中的解决办法。基本上，按现代 C++ 的手法（RAII）来管理内存，你很难遇到什么内存方面的错误。“没有错误”是基本要求，不代表“足够好”。我们常常会设法优化性能，如果 `profiling` 表明 `hot spot` 在内存分配和释放上，重载全局的 `::operator new()` 和 `::operator delete()` 似乎是一个一劳永逸好办法（以下简称“重载 `::operator new()`”），本文试图说明这个办法往往行不通。

2.1 内存管理的基本要求

如果只考虑分配和释放，内存管理基本要求是“不重不漏”：既不重复 `delete`，也不漏掉 `delete`。也就说我们常说的 `new/delete` 要配对，“配对”不仅是个数相等，还隐含了 `new` 和 `delete` 的调用本身要匹配，不要“东家借的东西西家还”。例如：

- 用系统默认的 `malloc()` 分配的内存要交给系统默认的 `free()` 去释放；
- 用系统默认的 `new` 表达式创建的对象要交给系统默认的 `delete` 表达式去析构并释放；
- 用系统默认的 `new[]` 表达式创建的对象要交给系统默认的 `delete[]` 表达式去析构并释放；
- 用系统默认的 `::operator new()` 分配的内存要交给系统默认的 `::operator delete()` 去释放；
- 用 `placement new` 创建的对象要用 `placement delete`（为了表述方便，姑且这么说吧）去析构（其实就是直接调用析构函数）；
- 从某个内存池 A 分配的内存要还给这个内存池。

¹<http://www.relisoft.com/book/tech/9new.html>

²<http://blog.csdn.net/Solstice/archive/2010/01/22/5238671.aspx>

- 如果定制 `new/delete`，那么要按规矩来。见 *Effective C++* [2] 第 8 章“定制 `new` 和 `delete`”。

做到以上这些不难，是每个 C++ 开发人员的基本功。不过，如果你想重载全局的 `::operator new()`，事情就麻烦了。

2.2 重载 `::operator new()` 的理由

《Effective C++ 第三版》[2] 第 50 条列举了定制 `new/delete` 的几点理由：

- 检测代码中的内存错误
- 优化性能
- 获得内存使用的统计数据

这些都是正当的需求，文末我们将会看到，不重载 `::operator new()` 也能达到同样的目的。

2.3 `::operator new()` 的两种重载方式

1. 不改变其签名，无缝直接替换系统原有的版本，例如：

```
#include <new>

void* operator new(size_t size);
void operator delete(void* p);
```

用这种方式的重载，使用方不需要包含任何特殊的头文件，也就是说不需要看见这两个函数声明。“性能优化”通常用这种方式。

2. 增加新的参数，调用时也提供这些额外的参数，例如：

```
// 此函数返回的指针必须能被普通的 ::operator delete(void*) 释放
void* operator new(size_t size, const char* file, int line);

// 此函数只在析构函数抛异常的情况下才会被调用
void operator delete(void* p, const char* file, int line);
```

然后用的时候是


```
Foo* p = new (__FILE__, __LINE__) Foo; // 这样能跟踪是哪个文件哪一行代码分配的内存
```

我们也可以用宏替换 `new` 来节省打字。用这第二种方式重载，使用方需要看到这两个函数声明，也就是说要主动包含你提供的头文件。“检测内存错误”和“统计内存使用情况”通常会用这种方式重载。当然，这不是绝对的。

在学习 C++ 的阶段，每个人都可以写个一两百行的程序来验证教科书上的说法，重载 `::operator new()` 在这样的玩具程序里边不会造成什么麻烦。

不过，我认为在现实的产品开发中，重载 `::operator new()` 乃是下策，我们有更简单安全的办法来到达以上目标。

2.4 现实的开发环境

作为 C++ 应用程序的开发人员，在编写稍具规模的程序时，我们通常会用到一些 `library`。我们可以根据 `library` 的提供方把它们大致分为这么几大类：

1. C 语言的标准库，也包括 Linux 编程环境提供的 `glibc` 系列函数。
2. 第三方的 C 语言库，例如 `OpenSSL`。
3. C++ 语言的标准库，主要是 `STL`。（我想没有人在产品中使用 `iostream` 吧？）
4. 第三方的通用 C++ 库，例如 `Boost.Regex`，或者某款 XML 库。
5. 公司其他团队的人开发的内部基础 C++ 库，比如网络通信和日志等基础设施。
6. 本项目组的同事自己开发的针对本应用的基础库，比如某三维模型的仿射变换模块。

在使用这些 `library` 的时候，不可避免地要在各个 `library` 之间交换数据。比方说 `library A` 的输出作为 `library B` 的输入，而 `library A` 的输出本身常常会用到动态分配的内存（比如 `std::vector<double>`）。

如果所有的 C++ `library` 都用同一套内存分配器（就是系统默认的 `new/delete`），那么内存的释放就很方便，直接交给 `delete` 去释放就行。如果不是这样，那就得时时刻刻记住“这一块内存是属于哪个分配器，是系统默认的还是我们定制的，释放的时候不要还错了地方”。

（由于 C 语言不像 C++ 一样提过了那么多的定制性，C `library` 通常都会默认直接用 `malloc/free` 来分配和释放内存，不存在上面提到的“内存还错地方”问题。或

者有的考虑更全面的 C library 会让你注册两个函数，用于它内部分配和释放内存，这就就能完全掌控该 library 的内存使用。这种依赖注入的方式在 C++ 里变得花哨而无用，见陈硕写的《C++ 标准库中的 allocator 是多余的》³。）

但是，如果重载了 `::operator new()`，事情恐怕就没有这么简单了。

2.5 重载 `::operator new()` 的困境

首先，重载 `::operator new()` 不会给 C 语言的库带来任何麻烦。当然，重载它得到的三点好处也无法让 C 语言的库享受到。

以下仅考虑 C++ library 和 C++ 主程序。

规则 1：绝对不能在 library 里重载 `::operator new()`

如果你是某个 library 的作者，你的 library 要提供给别人使用，那么你无权重载全局 `::operator new(size_t)`（注意这是前面提到的第一种重载方式），因为这非常具有侵略性：任何用到你的 library 的程序都被迫使用了你重载的 `::operator new()`，而别人很可能不愿意这么做。另外，如果有两个 library 都试图重载 `::operator new(size_t)`，那么它们会打架，我估计会发生 `duplicated symbol link error`。（这还算是好的，如果某个实现偷偷盖住了另一个实现，会在运行时发生诡异的现象。）干脆，作为 library 的编写者，大家都不要重载 `::operator new(size_t)` 好了。

那么第二种重载方式呢？

首先，`::operator new(size_t size, const char* file, int line)` 这种方式得到的 `void*` 指针必须同时能被 `::operator delete(void*)` 和 `::operator delete(void* p, const char* file, int line)` 这两个函数释放。这时候你需要决定，你的 `::operator new(size_t size, const char* file, int line)` 返回的指针是不是兼容系统默认的 `::operator delete(void*)`。

如果不兼容（也就是说不能用系统默认的 `::operator delete(void*)` 来释放内存），那么你得重载 `::operator delete(void*)`，让它的行为与你的 `::operator new(size_t size, const char* file, int line)` 匹配。一旦你决定重载 `::operator delete(void*)`，那么你必须重载 `::operator new(size_t)`，这就回到了规则 1：你无权重载全局 `::operator new(size_t)`。

³<http://blog.csdn.net/Solstice/archive/2009/08/02/4401382.aspx>

如果选择兼容系统默认的 `::operator delete(void*)`，那么你在 `::operator new(size_t size, const char* file, int line)` 里能做的事情非常有限，比方说你不能额外动态分配内存来做 **house keeping** 或保存统计数据（无论显示还是隐式），因为系统默认的 `::operator delete(void*)` 不会释放你额外分配的内存。（这里隐式分配内存指的是往 `std::map<>` 这样的容器里添加元素。）看到这里估计很多人已经晕了，但这还没完。

其次，在 **library** 里重载 `::operator new(size_t size, const char* file, int line)` 还涉及到你的重载要不要暴露给 **library** 的使用者（其他 **library** 或主程序）。这里“暴露”有两层意思：

1. 包含你的头文件的代码会不会用你重载的 `::operator new()`，
2. 重载之后的 `::operator new()` 分配的内存能不能在你的 **library** 之外被安全地释放。如果不行，那么你是不是要暴露某个接口函数来让使用者安全地释放内存？或者返回 `shared_ptr`，利用其“捕获”**deleter** 的特性？

听上去好像挺复杂？这里就不一一展开讨论了，总之，作为 **library** 的作者，我建议你绝对不要动“重载 `::operator new()`”的念头。

事实 2：在主程序里重载 `::operator new()` 作用不大

这不是一条规则，而是我试图说明这么做没有多大意义。

如果用第一种方式重载全局 `::operator new(size_t)`，会影响本程序用到的所有 C++ **library**，这么做或许不会有什么问题，不过我建议你使用下一节介绍的更简单的“替代办法”。

如果用第二种方式重载 `::operator new(size_t size, const char* file, int line)`，那么你的行为是否惠及本程序用到的其他 C++ **library** 呢？比方说你要不要统计 C++ **library** 中的内存使用情况？如果某个 **library** 会返回它自己用 **new** 分配的内存和对象，让你用完之后自己释放，那么是否打算对错误释放内存做检查？

C++ **library** 从代码组织上有两种形式：

1. 以头文件方式提供（如以 STL 和 Boost 为代表的模板库）；
2. 以头文件 + 二进制库文件方式提供（大多数非模板库以此方式发布）。

对于纯以头文件方式实现的 `library`，那么你可以在你的程序的每个 `.cpp` 文件的第一行包含重载 `::operator new()` 的头文件，这样程序里用到的其他 C++ `library` 也会转而使用你的 `::operator new()` 来分配内存。当然这是一种相当有侵略性的做法，如果运气好，编译和运行都没问题；如果运气差一点，可能会遇到编译错误，这其实还不算坏事；运气更差一点，编译没有错误，运行的时候时不时出现非法访问，导致 `segment fault`；或者在某些情况下你定制的分配策略与 `library` 有冲突，内存数据损坏，出现莫名其妙的行为。

对于以库文件方式实现的 `library`，这么做并不能让其受惠，因为 `library` 的源文件已经编译成了二进制代码，它不会调用你新重载的 `::operator new`（想想看，已经编译的二进制代码怎么可能提供额外的 `new (__FILE__, __LINE__)` 参数呢？）更麻烦的是，如果某些头文件有 `inline function`，还会引起诡异的“串扰”。即 `library` 有的部分用了你的分配器，有的部分用了系统默认的分配器，然后在释放内存的时候没有给对地方，造成分配器的数据结构被破坏。

总之，第二种重载方式看似功能更丰富，但其实与程序里使用的其他 C++ `library` 很难无缝配合。

综上，对于现实生活中的 C++ 项目，重载 `::operator new()` 几乎没有用武之地，因为很难处理好与程序所用的 C++ `library` 的关系，毕竟大多数 `library` 在设计的时候没有考虑到你会重载 `::operator new()` 并强塞给它。

如果确实需要定制内存分配，该如何办？

2.6 替代办法：替换 `malloc()`

很简单，替换 `malloc()`。如果需要，直接从 `malloc` 层面入手，通过 `LD_PRELOAD` 来加载一个 `.so`，其中有 `malloc/free` 的替代实现 (`drop-in replacement`)，这样能同时为 C 和 C++ 代码服务，而且避免 C++ 重载 `::operator new()` 的阴暗角落。

对于“检测内存错误”这一用法，我们可以用 `valgrind` 或者 `dmalloc` 或者 `efence` 来达到相同的目的，专业的除错工具比自己山寨一个内存检查器要靠谱。

对于“统计内存使用数据”，替换 `malloc` 同样能得到足够的信息，因为我们可以用 `backtrace()` 函数来获得调用栈，这比 `new (__FILE__, __LINE__)` 的信息更丰富。比方说你通过分析 `(__FILE__, __LINE__)` 发现 `std::string` 大量分配释放内存，有超出预期的开销，但是你却不知道代码里哪一部分在反复创建和销毁 `std::string` 对

象，因为 `(__FILE__, __LINE__)` 只能告诉你最内层的调用函数。用 `backtrace()` 能找到真正的发起调用者。

对于“性能优化”这一用法，我认为这目前的多线程开发中，自己实现一个能打败系统默认的 `malloc` 的内存分配器是不现实的。一个通用的内存分配器本来就有相当的难度，为多线程程序实现一个安全和高效的通用（全局）内存分配器超出了一般开发人员的能力。不如使用现有的针对多核多线程优化的 `malloc`，例如 Google `tcmalloc` 和 Intel TBB 2.2 里的内存分配器⁴。好在这些 `allocator` 都不是侵入式的，也无须重载 `::operator new()`。

2.7 为单独的 `class` 重载 `::operator new()` 有问题吗？

与全局 `::operator new()` 不同，`per-class operator new()` 和 `operator delete ()` 的影响面要小得多，它只影响本 `class` 及其派生类。似乎重载 `member ::operator new()` 是可行的。我对此持反对态度。

如果一个 `class Node` 需要重载 `member ::operator new()`，说明它用到了特殊的内存分配策略，常见的情况是使用了内存池或对象池。我宁愿把这一事实明显地摆出来，而不是改变 `new Node` 语句的默认行为。具体地说，是用 `factory` 来创建对象，比如 `static Node* Node::createNode()` 或者 `static shared_ptr<Node> Node::createNode()`。

这可以归结为最小惊讶原则：如果我在代码里读到 `Node* p = new Node`，我会认为它在 `heap` 上分配了内存，如果 `Node` `class` 重载了 `member ::operator new()`，那么我要事先仔细阅读 `node.h` 才能发现其实这行代码使用了私有的内存池。为什么不写得明确一点呢？写成 `Node* p = NodeFactory::createNode()`，那么我能猜到 `NodeFactory::createNode()` 肯定做了什么与 `new Node` 不一样的事情，免得将来大吃一惊。

The Zen of Python⁵ 说 `explicit is better than implicit`，我深信不疑。

2.8 有必要自行定制内存分配器吗？

如果写一个简单的只能分配固定大小的 `allocator`，确实很容易做到比系统的 `malloc` 更快，因为每次分配操作就是移动一下指针。但是我认为普通程序员很难写

⁴http://locklessinc.com/benchmarks_allocator.shtml

⁵<http://www.python.org/dev/peps/pep-0020/>

出可以与 `libc` 的 `malloc` 媲美的通用内存分配器，在多核多线程时代更是如此。应为 `libc` 有专人维护，会不断把适合新硬件的分配算法与策略整合进去。在打算写自己的内存池之前，建议先看一看 Andrei Alexandrescu 在 ACCU 2008 会议的演讲 *Memory Allocation: Either Love it or Hate It (Or Think It's Just OK)*.⁶ 和这篇论文 *Reconsidering Custom Memory Allocation*.^{7 8}

总结：重载 `::operator new()` 或许在某些临时的场合能应个急，但是不应该作为一种策略来使用。如果需要，我们可以从 `malloc` 层面入手，彻底而全面地替换内存分配器。

3 采用有利于版本管理的代码格式

版本管理 (version controlling) 是每个程序员的基本技能，C++ 程序员也不例外。版本管理的基本功能之一是追踪代码变化，让你能清楚地知道代码是如何一步步变成现在的这个样子，以及每次 `check-in` 都具体改动了哪些内部。无论是传统的集中式版本管理工具，如 `Subversion`，还是新型的分布式管理工具，如 `Git/Hg`，比较两个版本 (revision) 的差异都是其基本功能，即俗称“做一下 `diff`”。

`diff` 的输出是个窥孔 (peephole)，它的上下文有限 (`diff -u` 默认显示前后 3 行)。在做 `code review` 的时候，如果能凭这“一孔之见”就能发现代码改动有问题，那就再好不过了。

C 和 C++ 都是自由格式的语言，代码中的换行符被当做 `white space` 来对待。(当然，我们说的是预处理 (preprocess) 之后的情况)。对编译器来说一模一样的代码可以有多种写法，比如

```
foo(1, 2, 3, 4);
```

和

```
foo(1,
    2,
    3,
    4);
```

⁶<http://accu.org/content/conf2008/Alexandrescu-memory-allocation.screen.pdf>

⁷<http://www.cs.umass.edu/emery/pubs/berger-oopsla2002.pdf>

⁸<http://www.cs.umass.edu/emery/talks/OOPSLA-2002.ppt>

词法分析的结果是一样的，语意也完全一样。

对人来说，这两种写法读起来不一样，对与版本管理工具来说，同样功能的修改造成的差异 (diff) 也往往不一样。所谓“有利于版本管理”，就是指在代码中合理使用换行符，对 diff 工具友好，让 diff 的结果清晰明了地表达代码的改动。(diff 一般以行为单位，也可以以单词为单位，本文只考虑最常见的 diff by lines。)

这里举一些例子。

3.1 对 diff 友好的代码格式

3.1.1 多行注释也用 //，不用 /* */

Scott Meyers 写的《Effective C++》第二版第 4 条建议使用 C++ 风格，我这里为他补充一条理由：对 diff 友好。比如，我要注释一大段代码（其实这不是个好的做法，但是在实践中有时会遇到），如果用 /* */，那么得到的 diff 是：

```
--- a/examples/asio/tutorial/timer5/timer.cc
+++ b/examples/asio/tutorial/timer5/timer.cc
@@ -18,6 +18,7 @@ class Printer : boost::noncopyable
     loop2_>runAfter(1, boost::bind(&Printer::print2, this));
 }
+ /*
+ ~Printer()
+ {
+     std::cout << "Final count is " << count_ << "\n";
@@ -38,6 +39,7 @@ class Printer : boost::noncopyable
     loop1_>quit();
 }
+ */

void print2()
{
```

从这样的 diff output 能看出注释了哪些代码吗？

如果用 //，结果会清晰很多：

```
--- a/examples/asio/tutorial/timer5/timer.cc
+++ b/examples/asio/tutorial/timer5/timer.cc
@@ -18,26 +18,26 @@ class Printer : boost::noncopyable
     loop2_>runAfter(1, boost::bind(&Printer::print2, this));
 }
```

```

- ~Printer()
- {
-     std::cout << "Final count is " << count_ << "\n";
- }
+ // ~Printer()
+ // {
+ //     std::cout << "Final count is " << count_ << "\n";
+ // }

- void print1()
- {
-     muduo::MutexLockGuard lock(mutex_);
-     if (count_ < 10)
-     {
-         std::cout << "Timer 1: " << count_ << "\n";
-         ++count_;
-
-         loop1_->runAfter(1, boost::bind(&Printer::print1, this));
-     }
-     else
-     {
-         loop1_->quit();
-     }
- }
+ // void print1()
+ // {
+ //     muduo::MutexLockGuard lock(mutex_);
+ //     if (count_ < 10)
+ //     {
+ //         std::cout << "Timer 1: " << count_ << "\n";
+ //         ++count_;
+ //
+ //         loop1_->runAfter(1, boost::bind(&Printer::print1, this));
+ //     }
+ //     else
+ //     {
+ //         loop1_->quit();
+ //     }
+ // }

void print2()
{

```

同样的道理，取消注释的时候 `//` 也比 `/* */` 更清晰。

另外，如果用 `/* */` 来做多行注释，从 `diff` 不一定能看出来你是在修改代码还是修改注释。比如以下 `diff` 似乎修改了 `muduo::EventLoop::runAfter()` 的调用参数：

```

--- a/examples/asio/tutorial/timer5/timer.cc
+++ b/examples/asio/tutorial/timer5/timer.cc
@@ -32,7 +32,7 @@ class Printer : boost::noncopyable

```



```

        std::cout << "Timer 1: " << count_ << std::endl;
        ++count_;
-    loop1_>runAfter(1, boost::bind(&Printer::print1, this));
+    loop1_>runAfter(2, boost::bind(&Printer::print1, this));
    }
    else
    {

```

其实这个修改发生在注释里边（要增加上下文才能看到，diff -U 20，多一道手续，降低了工作效率），对代码行为没有影响：

```

--- a/examples/asio/tutorial/timer5/timer.cc
+++ b/examples/asio/tutorial/timer5/timer.cc
@@ -20,31 +20,31 @@ class Printer : boost::noncopyable

/*
~Printer()
{
    std::cout << "Final count is " << count_ << std::endl;
}

void print1()
{
    muduo::MutexLockGuard lock(mutex_);
    if (count_ < 10)
    {
        std::cout << "Timer 1: " << count_ << std::endl;
        ++count_;
-    loop1_>runAfter(1, boost::bind(&Printer::print1, this));
+    loop1_>runAfter(2, boost::bind(&Printer::print1, this));
    }
    else
    {
        loop1_>quit();
    }
}
*/

void print2()
{
    muduo::MutexLockGuard lock(mutex_);
    if (count_ < 10)
    {
        std::cout << "Timer 2: " << count_ << std::endl;
        ++count_;

```

总之，不要用 `/* */` 来注释多行代码。

或许是时过境迁，大家都在用 `//` 注释了，《Effective C++》第三版去掉了这一条建议。

3.1.2 局部变量与成员变量的定义

基本原则是，一行代码只定义一个变量，比如

```
double x;  
double y;
```

将来代码增加一个 `double z` 的时候，`diff` 输出一眼就能看出改了什么：

```
@@ -63,6 +63,7 @@ private:  
    int count_;  
    double x;  
    double y;  
+   double z;  
};  
  
int main()
```

如果把 `x` 和 `y` 写在一行，`diff` 的输出就得多看几眼才知道。

```
@@ -61,7 +61,7 @@ private:  
    muduo::net::EventLoop* loop1_;  
    muduo::net::EventLoop* loop2_;  
    int count_;  
-   double x, y;  
+   double x, y, z;  
};  
  
int main()
```

所以，一行只定义一个变量更利于版本管理。同样的道理适用于 `enum` 成员的定义，数组的初始化列表等等。

3.1.3 函数声明中的参数

如果函数的参数大于 3 个，那么在逗号后面换行，这样每个参数占一行，便于 `diff`。以 `muduo::net::TcpClient` 为例：

```
----- muduo/net/TcpClient.h  
class TcpClient : boost::noncopyable  
{  
public:  
    TcpClient(EventLoop* loop,  
              const InetAddress& serverAddr,  
              const string& name);  
----- muduo/net/TcpClient.h
```

如果将来 `TcpClient` 的构造函数增加或修改一个参数，那么很容易从 `diff` 看出来。这恐怕比在一行长代码里数逗号要高效一些。

3.1.4 函数调用时的参数

在函数调用的时候，如果参数大于 3 个，那么把实参分行写。

以 `muduo::net::EPollPoller` 为例：

```
Timestamp EPollPoller::poll(int timeoutMs, ChannelList* activeChannels)
{
    int numEvents = ::epoll_wait(epollfd_,
                                &*events_.begin(),
                                static_cast<int>(events_.size()),
                                timeoutMs);
    Timestamp now(Timestamp::now());
}
```

muduo/net/poller/EPollPoller.cc

这样一来，如果将来重构引入了一个新参数（好吧，`epoll_wait` 不会有这个问题），那么函数定义和函数调用的地方的 `diff` 具有相同的形式（比方说都是在倒数第二行加了一行内容），很容易肉眼验证有没有错位。如果参数写在一行里边，就得睁大眼睛数逗号了。

3.1.5 class 初始化列表的写法

同样的道理，`class` 初始化列表 (initializer list) 也遵循一行一个的原则，这样将来如果加入新的成员变量，那么两处（`class` 定义和 `ctor` 定义）的 `diff` 具有相同的形式，让错误无所遁形。以 `muduo::net::Buffer` 为例：

```
class Buffer : public muduo::copyable
{
public:
    static const size_t kCheapPrepend = 8;
    static const size_t kInitialSize = 1024;

    Buffer()
        : buffer_(kCheapPrepend + kInitialSize),
          readerIndex_(kCheapPrepend),
          writerIndex_(kCheapPrepend)
    {
    }
    // 省略
private:
    std::vector<char> buffer_;
    size_t readerIndex_;
    size_t writerIndex_;
};
```

muduo/net/Buffer.h

注意，初始化列表的顺序必须和数据成员声明的顺序相同。

3.1.6 与 namespace 有关的缩进

Google 的 C++ 编程规范明确指出，namespace 不增加缩进⁹。这么做非常有道理，方便 diff -p 把函数名显示在每个 diff chunk 的头上。

如果对函数实现做 diff，chunk name 是函数名，让人一眼就能看出改的是哪个函数。如下图，阴影部分。

```
diff --git a/muduo/net/SocketsOps.cc b/muduo/net/SocketsOps.cc
--- a/muduo/net/SocketsOps.cc
+++ b/muduo/net/SocketsOps.cc
@@ -125,7 +125,7 @@ int sockets::accept(int sockfd, struct sockaddr_in* addr)
     case ENOTSOCK:
     case EOPNOTSUPP:
         // unexpected errors
-        LOG_FATAL << "unexpected error of ::accept";
+        LOG_FATAL << "unexpected error of ::accept " << savedErrno;
         break;
     default:
         LOG_FATAL << "unknown error of ::accept " << savedErrno;
```

如果对 class 做 diff，那么 chunk name 就是 class name。

```
diff --git a/muduo/net/Buffer.h b/muduo/net/Buffer.h
--- a/muduo/net/Buffer.h
+++ b/muduo/net/Buffer.h
@@ -60,13 +60,13 @@ class Buffer : public muduo::copyable
     std::swap(writerIndex_, rhs.writerIndex_);
 }

- size_t readableBytes();
+ size_t readableBytes() const;

- size_t writableBytes();
+ size_t writableBytes() const;

- size_t prependableBytes();
+ size_t prependableBytes() const;

    const char* peek() const;
```

diff 原本是为 C 语言设计的，C 语言没有 namespace 缩进一说，所以它默认会找到“顶格写”的函数作为一个 diff chunk 的名字，如果函数名前面有空格，它就不认得了。muduo 的代码都遵循这一规则，例如：

⁹http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml#Namespace_Formatting

```

namespace muduo                                     muduo/base/Timestamp.h
{

// class 从第一列开始写，不缩进
class Timestamp : public muduo::copyable
{
    // ...
};

```

```

// 函数的实现也从第一列开始写，不缩进。
Timestamp Timestamp::now()
{
    struct timeval tv;
    gettimeofday(&tv, NULL);
    int64_t seconds = tv.tv_sec;
    return Timestamp(seconds * kMicroSecondsPerSecond + tv.tv_usec);
}

```

```


```

相反，boost 中的某些库的代码是按 **namespace** 来缩进的，这样的话看 diff 往往不知道改动的是哪个 **class** 的哪个成员函数。

这个或许可以通过设置 diff 取函数名的正则表达式来解决，但是如果我们写代码的时候就注意把函数“顶格写”，那么就不用去动 diff 的默认设置了。另外，正则表达式不能完全匹配函数名，因为函数名是上下文无关语法 (context-free syntax)，你没办法写一个正则语法去匹配上下文无关语法。我总能写出某种函数声明，让你的正则表达式失效（想想函数的返回类型，它可能是一个非常复杂的东西，更别说参数了）。更何况 C++ 的语法是上下文相关的，比如你猜 `Foo<Bar> qux;` 是个表达式还是变量定义？

3.1.7 public 与 private

我认为这是 C++ 语法的一个缺陷，如果我把一个成员函数从 **public** 区移到 **private** 区，那么从 diff 上看不出来我干了什么，例如：

```

diff --git a/muduo/net/TcpClient.h b/muduo/net/TcpClient.h
--- a/muduo/net/TcpClient.h
+++ b/muduo/net/TcpClient.h
@@ -37,7 +37,6 @@ class TcpClient : boost::noncopyable
     void connect();
     void disconnect();

```

```

- bool retry() const;
  void enableRetry() { retry_ = true; }

  /// Set connection callback.
@@ -60,6 +59,7 @@ class TcpClient : boost::noncopyable
  void newConnection(int sockfd);
  /// Not thread safe, but in loop
  void removeConnection(const TcpConnectionPtr& conn);
+ bool retry() const;

  EventLoop* loop_;
  boost::scoped_ptr<Connector> connector_; // avoid revealing Connector

```

从上面的 diff 能看出我把 `retry()` 变成 `private` 了吗？对此我也没有好的解决办法，总不能每个函数前面都写上 `public:` 或 `private:` 吧？

对此 Java 和 C# 都做得比较好，它们把 `public/private` 等修饰符放到每个成员函数的定义中。这么做增加了信息的冗余度，让 diff 的结果更直观。

3.1.8 避免使用版本控制软件的 keyword substitution 功能

这么做是为了避免 diff 噪音。

比方说，如果我想比较 0.1.1 和 0.1.2 两个代码分支有哪些改动，我通常会在 `branches` 目录执行 `diff 0.1.1 0.1.2 -ru`。两个 `branch` 中的 `muduo/net/EventLoop.h` 其实是一样的（先后从同一个 `revision` 分支出来）。但是如果这个文件使用了 SVN 的 `keyword substitution` 功能（比如 `Id`），diff 会报告这两个 `branches` 中的文件不一样，如下。

```

diff -rup 0.1.1/muduo/net/EventLoop.h 0.1.2/muduo/net/EventLoop.h
--- 0.1.1/muduo/net/EventLoop.h 2011-05-02 23:11:02.000000000 +0800
+++ 0.1.2/muduo/net/EventLoop.h 2011-05-02 23:12:22.000000000 +0800
@@ -8,7 +8,7 @@
  //
  // This is a public header file, it must only include public header files.

-// $Id: EventLoop.h 4 2011-05-01 10:11:02Z schen $
+// $Id: EventLoop.h 5 2011-05-02 15:12:22Z schen $

  #ifndef MUDUO_NET_EVENTLOOP_H
  #define MUDUO_NET_EVENTLOOP_H

```

这样纯粹增加了噪音，这是 RCS/CVS 时代过时的做法。文件的 `Id` 不应该在文件内容中出现，这些 `metadata` 跟源文件的内容无关，应该由版本管理软件额外提供。

3.2 对 grep 友好的代码风格

3.2.1 操作符重载

C++ 工具匮乏，在一个项目里，要找到一个函数的定义或许不算太难（最多就是分析一下重载和模板特化），但是要找到一个函数的使用就难多了。不比 Java，在 Eclipse 里 Ctrl+Shift+G 就能找到所有的引用点。

假如我要做一个重构，想先找到代码里所有用到 `muduo::timeDifference()` 的地方，判断一下工作是否可行，基本上惟一的办法是 `grep`。用 `grep` 还不能排除同名的函数和注释里的内容。这也说明为什么要用 `//` 来引导注释，因为在 `grep` 的时候，一眼就能看出这行代码是在注释里的。

在我看来，`operator overloading` 应仅限于和 STL `algorithm/container` 配合时使用，比如 `std::transform()` 和 `map<Key, Value>`，其他情况都用具名函数为宜。原因之一是，我根本用 `grep` 找不到在哪儿用到了减号 `operator-()`。这也是 `muduo::Timestamp` class 只提供 `operator<()` 而不提供 `operator+()` `operator-()` 的原因，我提供了两个函数 `timeDifference()` 和 `addTime()` 来实现所需的功能。

又比如，Google Protocol Buffers 的回调是 `class Closure`，它的接口用的是 `virtual function Run()` 而不是 `virtual operator()()`。

3.2.2 static_cast 与 C-style cast

为什么 C++ 要引入 `static_cast` 之类的转型操作符，原因之一就是像 `(int*) pBuffer` 这样的表达式基本上没办法用 `grep` 判断出它是个强制类型转换，写不出一个刚好只匹配类型转换的正则表达式。（again，语法是上下文无关的，无法用正则搞定。）

如果类型转换都用 `*_cast`，那只要 `grep` 一下我就能知道代码里哪儿用了 `reinterpret_cast` 转换，便于迅速地检查有没有用错。为了强调这一点，`muduo` 开启了编译选项 `-Wold-style-cast` 来帮助查找 C-style casting，这样在编译时就能帮我们找到问题。

3.3 一切为了效率

如果用图形化的文件比较工具，似乎能避免上面列举的问题。但无论是 `web` 还是客户端，无论是 `diff by words` 还是 `diff by lines` 都不能解决全部问题，效率也不一定更高。

对于 (3.1.2), 如果想知道是谁在什么时候增加的 `double z`, 在分行写的情况下, 用 `git blame` 或 `svn blame` 立刻就能找到始作俑者。如果写成一行, 那就得把文件的 `revisions` 拿来一个个人工比较, 因为这一行 `double x = 0.0, y = 1.0, z = -1.0`; 可能修改过多次, 你得一个个看才知道什么时候加入了变量 `z`。这个 `blame` 的 `case` 也适用于 3、4、5。

比如 (3.1.6) 改动了一行代码, 你还是要 `scroll up` 去找改的是哪个 `function`, 人眼看的话还有“看走眼”的可能, 又得再定睛观瞧。这一切都是浪费人的时间, 使用更好的图形化工具并不能减少浪费, 相反, 我认为增加了浪费。

另外一个常见的工作场景, 早上来到办公室, `update` 一下代码, 然后扫一眼 `diff output` 看看别人昨天动了哪些文件, 改了哪些代码。这就是一两条命令的事, 几秒钟就能解决战斗。如果用图形化的工具, 得一个个点开文件 `diff` 的链接或点开新 `tab` 来看文件的 `side-by-side` 比较 (不这么做的话看不到足够多的上下文, 跟看 `diff output` 无异), 然后点击鼠标滚动页面去看别人到底改了什么。说实话我觉得这么做效率不比 `diff` 高。

4 二进制兼容性

本文主要讨论 Linux x86/x86-64 平台, 偶尔会举 Windows 作为反面教材。

C++ 程序员有不同的角色, 比如有主要编写应用程序的 (`application`), 也有主要编写程序库的 (`library`), 有的程序员或许还身兼多职。如果公司的规模比较大, 会出现更细致和明确的分工。比如有的团队专门负责一两个公用的 `library`, 有的团队负责某个 `application`, 并使用了前一个团队的 `library`。

举一个具体的例子。假设你负责一个图形库, 这个图形库功能强大, 且经过了充分测试, 于是在公司内慢慢推广开来。目前已经有二三十个内部项目用到了你的图形库, 大家日子过得挺好。前几天, 公司新买了一批大屏幕显示器 (2560×1600 分辨率), 不巧你的图形库不能支持这么高的分辨率。(这其实不怪你, 因为在你编写这个库的时候, 市面上显示器的最高分辨率是 1920×1200 。)

结果用到了你的图形库的应用程序在 2560×1600 分辨率下不能正常工作, 你该怎么办? 你可以发布一个新版的图形库, 并要求那二三十个项目组用你的新库重新编译他们的程序, 然后让他们重新发布应用程序。或者, 你提供一个新的库文件, 直接替换现有的库文件, 应用程序可执行文件保持不变。

这两种做法各有优劣。第一种声势浩大，凡是用到你的库的团队都要经历一个 **release cycle**。后一种办法似乎节省人力，但是有风险：如果新的库文件和原有的应用程序可执行文件不兼容怎么办？

所以，作为 C++ 程序员，只要工作涉及二进制的程序库（特别是动态库），都需要了解二进制兼容性方面的知识。

C/C++ 的二进制兼容性 (binary compatibility) 有多重含义，本文主要在“库文件单独升级，现有可执行文件是否受影响”这个意义下讨论，我称之为 **library**（主要是 **shared library**，即动态链接库）的 **ABI** (application binary interface)。至于编译器与操作系统的 **ABI** 留给下一篇谈 C++ 标准与实践的文章。

4.1 什么是二进制兼容性

在解释这个定义之前，先看看 Unix/C 语言的一个历史问题：`open()` 的 `flags` 参数的取值。`open(2)` 函数的原型是

```
int open(const char *pathname, int flags);
```

其中 `flags` 的取值有三个：`O_RDONLY`, `O_WRONLY`, `O_RDWR`。

与一般人的直觉相反，这几个值不是**按位或** (bitwise-OR) 的关系，即 `O_RDONLY | O_WRONLY != O_RDWR`。如果你想以读写方式打开文件，必须用 `O_RDWR`，而不能用 `(O_RDONLY | O_WRONLY)`。为什么？因为 `O_RDONLY`, `O_WRONLY`, `O_RDWR` 的值分别是 0, 1, 2。它们不满足按位或。

那么为什么 C 语言从诞生到现在一直没有纠正这个小小的缺陷？比方说把 `O_RDONLY`, `O_WRONLY`, `O_RDWR` 分别定义为 1, 2, 3，这样 `O_RDONLY | O_WRONLY == O_RDWR`，符合直觉。而且这三个值都是宏定义，也不需要修改现有的源代码，只需要改改系统的头文件就行了。

因为这么做会破坏二进制兼容性。对于已经编译好的可执行文件，它调用 `open(2)` 的参数是写死的，更改头文件并不能影响已经编译好的可执行文件。比方说这个可执行文件会调用 `open(path, 1)` 来写文件，而在新规定中，这表示读文件，程序就错乱了。

以上这个例子说明，如果以 **shared library** 方式提供函数库，那么头文件和库文件不能轻易修改，否则容易破坏已有的二进制可执行文件，或者其他用到这个 **shared library** 的 **library**。

操作系统的 `system call` 可以看成 Kernel 与 User space 的 interface, kernel 在这个意义下也可以当成 `shared library`, 你可以把内核从 2.6.30 升级到 2.6.35, 而不需要重新编译所有用户态的程序。

所谓“二进制兼容性”指的就是在升级（也可能是 `bug fix`）库文件的时候，不必重新编译使用了这个库的可执行文件或其他库文件，程序的功能不被破坏。

见 QT FAQ 的有关条款：http://developer.qt.nokia.com/faq/answer/you_frequently_say_that_you_cannot_add_this_or_that_feature_because_it_woul

在 Windows 下有恶名叫 DLL Hell, 比如 MFC 有一堆 DLL, `mfc40.dll`, `mfc42.dll`, `mfc71.dll`, `mfc80.dll`, `mfc90.dll`, 这是动态链接库的本质问题, 怪不到 MFC 头上。

4.2 有哪些情况会破坏库的 ABI

到底如何判断一个改动是不是二进制兼容呢？这跟 C++ 的实现方式直接相关, 虽然 C++ 标准没有规定 C++ 的 ABI, 但是几乎所有主流平台都有明文或事实上的 ABI 标准。比方说 ARM 有 EABI, Intel Itanium 有 Itanium ABI¹⁰, x86-64 有仿 Itanium 的 ABI, SPARC 和 MIPS 也都有明文规定的 ABI, 等等。x86 是个例外, 它只有事实上的 ABI, 比如 Windows 就是 Visual C++, Linux 是 G++ (G++ 的 ABI 还有多个版本, 目前最新的是 G++ 3.4 的版本), Intel 的 C++ 编译器也得按照 Visual C++ 或 G++ 的 ABI 来生成代码, 否则就不能与系统其它部件兼容。

C++ ABI 的主要内容:

- 函数参数传递的方式, 比如 x86-64 用寄存器来传函数的前 4 个整数参数
- 虚函数的调用方式, 通常是 `vptr/vtbl` 然后用 `vtbl[offset]` 来调用
- `struct` 和 `class` 的内存布局, 通过偏移量来访问数据成员
- `name mangling`
- RTTI 和异常处理的实现 (以下本文不考虑异常处理)

C/C++ 通过头文件暴露出动态库的使用方法, 这个“使用方法”主要是给编译器看的, 编译器会据此生成二进制代码, 然后在运行的时候通过装载器 (`loader`) 把可执行文件和动态库绑到一起。如何判断一个改动是不是二进制兼容, 主要就是看头文

¹⁰<http://www.codesourcery.com/public/cxx-abi/abi.html>

件暴露的这份“使用说明”能否与新版本的动态库的实际使用方法兼容。因为新的库必然有新的头文件，但是现有的二进制可执行文件还是按旧的头文件来调用动态库。

这里举一些源代码兼容但是二进制代码不兼容例子

- 给函数增加默认参数，现有的可执行文件无法传这个额外的参数。
- 增加虚函数，会造成 `vtbl` 里的排列变化。(不要考虑“只在末尾增加”这种取巧行为，因为你的 `class` 可能已被继承。)
- 增加默认模板类型参数，比方说 `Foo<T>` 改为 `Foo<T, Alloc=alloc<T> >`，这会改变 `name mangling`
- 改变 `enum` 的值，把 `enum Color { Red = 3 };` 改为 `Red = 4`。这会造成错位。当然，由于 `enum` 自动排列取值，添加 `enum` 项也是不安全的，在末尾添加除外。

给 `class Bar` 增加数据成员，造成 `sizeof(Bar)` 变大，以及内部数据成员的 `offset` 变化，这是不是安全的？通常不是安全的，但也有例外。

- 如果客户代码里有 `new Bar`，那么肯定不安全，因为 `new` 的字节数不够装下新 `Bar` 对象。相反，如果 `library` 通过 `factory` 返回 `Bar*`（并通过 `factory` 来销毁对象）或者直接返回 `shared_ptr<Bar>`，客户端不需要用到 `sizeof(Bar)`，那么可能是安全的。
- 如果客户代码里有 `Bar* pBar; pBar->memberA = xx;`，那么肯定不安全，因为 `memberA` 的新 `Bar` 的偏移可能会变。相反，如果只通过成员函数来访问对象的数据成员，客户端不需要用到 `data member` 的 `offsets`，那么可能是安全的。
- 如果客户调用 `pBar->setMemberA(xx)`；而 `Bar::setMemberA()` 是个 `inline function`，那么肯定不安全，因为偏移量已经被 `inline` 到客户的二进制代码里了。如果 `setMemberA()` 是 `outline function`，其实现位于 `shared library` 中，会随着 `Bar` 的更新而更新，那么可能是安全的。

那么只使用 `header-only` 的库文件是不是安全呢？不一定。如果你的程序用了 `boost 1.36.0`，而你依赖的某个 `library` 在编译的时候用的是 `1.33.1`，那么你的程序和这个 `library` 就不能正常工作。因为 `1.36.0` 和 `1.33.1` 的 `boost::function` 的模板参数类型的个数不一样，后者一个多了 `allocator`。

这里有一份黑名单，列在这里的肯定是二级制不兼容，没有列出的也可能二进制不兼容，见 KDE 的文档：http://techbase.kde.org/Policies/Binary_Compatibility_Issues_With_C%2B%2B

4.3 哪些做法多半是安全的

前面我说“不能轻易修改”，暗示有些改动多半是安全的，这里有一份白名单，欢迎添加更多内容。

只要库改动不影响现有的可执行文件的二进制代码的正确性，那么就是安全的，我们可以先部署新的库，让现有的二进制程序受益。

- 增加新的 `class`
- 增加 `non-virtual` 成员函数或 `static` 成员函数
- 修改数据成员的名称，因为生产的二进制代码是按偏移量来访问的，当然，这会造成源码级的不兼容。
- 还有很多，不一一列举了。

欢迎补充

4.4 反面教材：COM

在 C++ 中以虚函数作为接口基本上就跟二进制兼容性说拜拜了。具体地说，以只包含虚函数的 `class`（称为 `interface class`）作为程序库的接口，这样的接口是僵硬的，一旦发布，无法修改。

此处的内容挪到“避免使用虚函数作为库的接口”一节中详细论述。

另外，Windows 下，Visual C++ 编译的时候要选择 Release 或 Debug 模式，而且 Debug 模式编译出来的 `library` 通常不能在 Release binary 中使用（反之亦然），这也是因为两种模式下的 CRT 二进制不兼容（主要是内存分配方面，Debug 有自己的簿记）。Linux 就没有这个麻烦，可以混用。

4.5 解决办法

采用静态链接

这个是王道。在分布式系统这，采用静态链接也带来部署上的好处，只要把可执行文件放到机器上就行运行，不用考虑它依赖的 `libraries`。目前 `muduo` 就是采用静态链接。

通过动态库的版本管理来控制兼容性

这需要非常小心检查每次改动的二进制兼容性并做好发布计划，比如 1.0.x 系列做到二进制兼容，1.1.x 系列做到二进制兼容，而 1.0.x 和 1.1.x 二进制不兼容。《程序员自我修养》[6] 里边讲过 .so 文件的命名与二进制兼容性相关的话题，值得一读。

用 pimpl 技法，编译器防火墙

在头文件中只暴露 non-virtual 接口，并且 class 的大小固定为 sizeof(Impl*)，这样可以随意更新库文件而不影响可执行文件。具体做法见第 6 节。当然，这么做有多了一道间接性，可能有一定的性能损失。另见 *Exceptional C++* 有关条款和 C++ *Coding Standards 101* [4]。

5 避免使用虚函数作为库的接口

摘要 作为 C++ 动态库的作者，应当避免使用虚函数作为库的接口。这么做会给保持二进制兼容性带来很大麻烦，不得不增加很多不必要的 interfaces，最终重蹈 COM 的覆辙。

本文主要讨论 Linux x86 平台，会继续举 Windows/COM 作为反面教材。

本文是上一篇《二进制兼容性》的延续，在写这篇文章的时候，我原本以外大家都对“以 C++ 虚函数作为接口”的害处达成共识，我就写得比较简略，看来情况不是这样，我还得展开谈一谈。

“接口”有广义和狭义之分，本文用中文“接口”表示广义的接口，即一个库的代码界面；用英文 interface 表示狭义的接口，即只包含 virtual function 的 class，这种 class 通常没有 data member，在 Java 里有一个专门的关键字 interface 来表示它。

5.1 C++ 程序库的作者的生存环境

假设你是一个 shared library 的维护者，你的 library 被公司另外两三个团队使用了。你发现了一个安全漏洞，或者某个会导致 crash 的 bug 需要紧急修复，那么你修复之后，能不能直接部署 library 的二进制文件？有没有破坏二进制兼容性？会不会破坏别人团队已经编译好的投入生成环境的可执行文件？是不是要强迫别的团队重

新编译链接，把可执行文件也发布新版本？会不会打乱别人的 **release cycle**？这些都是工程开发中经常要遇到的问题。

如果你打算新写一个 **C++ library**，那么通常要做以下几个决策：

- 以什么方式发布？动态库还是静态库？（本文不考虑源代码发布这种情况，这其实和静态库类似。）
- 以什么方式暴露库的接口？可选的做法有：以全局（含 **namespace** 级别）函数为接口、以 **class** 的 **non-virtual** 成员函数为接口、以 **virtual** 函数为接口（**interface**）。

（Java 程序员没有这么多需要考虑的，直接写 **class** 成员函数就行，最多考虑一下要不要给 **method** 或 **class** 标上 **final**。也不必考虑动态库静态库，都是 **.jar** 文件。）

在作出上面两个决策之前，我们考虑两个基本假设：

- 代码会有 **bug**，库也不例外。将来可能会发布 **bug fixes**。
- 会有新的功能需求。写代码不是一锤子买卖，总是会有新的需求冒出来，需要程序员往库里增加东西。这是好事情，让程序员不丢饭碗。

（如果你的代码第一次发布的时候就已经做到完美，将来不需要任何修改，那么怎么做都行，也就不必继续阅读本文。）

也就是说，在设计库的时候必须要考虑将来如何升级。

基于以上两个基本假设来做决定。第一个决定很好做，如果需要 **hot fix**，那么只能用动态库；否则，在分布式系统中使用静态库更容易部署，这在前文中已经谈过。（“动态库比静态库节约内存”这种优势在今天看来已不太重要。）

以下本文假定你或者你的老板选择以动态库方式发布，即发布 **.so** 或 **.dll** 文件，来看看第二个决定怎么做。（再说一句，如果你能够以静态库方式发布，后面的麻烦都不会遇到。）

第二个决定不是那么容易做，关键问题是，要选择一种可扩展的 (**extensible**) 接口风格，让库的升级变得更轻松。“升级”有两层意思：

- 对于 **bug fix only** 的升级，二进制库文件的替换应该兼容现有的二进制可执行文件，二进制兼容性方面的问题已经在前文谈过，这里从略。

- 对于新增功能的升级，应该对客户代码的友好。升级库之后，客户端使用新功能的代价应该比较小。只需要包含新的头文件（这一步都可以省略，如果新功能已经加入原有的头文件中），然后编写新代码即可。而且，不要在客户代码中留下垃圾，后文我们会谈到什么是垃圾。

在讨论虚函数接口的弊端之前，我们先看看虚函数做接口的常见用法。

5.2 虚函数作为库的接口的两大用途

虚函数为接口大致有这么两种用法：

- **调用**，也就是库提供一个什么功能（比如绘图 `Graphics`），以虚函数为接口方式暴露给客户端代码。客户端代码一般不需要继承这个 `interface`，而是直接调用其 `member function`。这么做据说是有利于接口和实现分离，我认为纯属脱了裤子放屁。
- **回调**，也就是事件通知，比如网络库的“连接建立”、“数据到达”、“连接断开”等等。客户端代码一般会继承这个 `interface`，然后把对象实例注册到库里边，等库来回调自己。一般来说客户端不会自己去调用这些 `member function`，除非是为了写单元测试模拟库的行为。
- **混合**，一个 `class` 既可以被客户端代码继承用作回调，又可以被客户端直接调用。说实话我没看出这么做的好处，但实际中某些面向对象的 C++ 库就是这么设计的。

对于“回调”方式，现代 C++ 有更好的做法，即 `boost::function + boost::bind`，见第 7 节，`muduo` 的回调全部采用这种新方法，见《`Muduo` 网络编程示例之零：前言》¹¹。本文以下不考虑以虚函数为回调的过时做法。

对于“调用”方式，这里举一个虚构的图形库，这个库的功能是画线、画矩形、画圆弧：

```
struct Point
{
    int x;
    int y;
};
```

¹¹<http://blog.csdn.net/Solstice/archive/2011/02/02/6171831.aspx>

```
class Graphics
{
    virtual void drawLine(int x0, int y0, int x1, int y1);
    virtual void drawLine(Point p0, Point p1);

    virtual void drawRectangle(int x0, int y0, int x1, int y1);
    virtual void drawRectangle(Point p0, Point p1);

    virtual void drawArc(int x, int y, int r);
    virtual void drawArc(Point p, int r);
};
```

这里略去了很多与本文主题无关细节，比如 `Graphics` 的构造与析构、`draw*()` 函数应该是 `public`、`Graphics` 应该不允许复制，还比如 `Graphics` 可能会用 `pure virtual functions` 等等，这些都不影响本文的讨论。

这个 `Graphics` 库的使用很简单，客户端看起来是这个样子。

```
Graphics* g = getGraphics();
g->drawLine(0, 0, 100, 200);
releaseGraphics(g);
g = NULL;
```

似乎一切都很好，阳光明媚，符合“面向对象的原则”，但是一旦考虑升级，前景立刻变得昏暗。

5.3 虚函数作为接口的弊端

以虚函数作为接口在二进制兼容性方面有本质困难：“一旦发布，不能修改”。

假如我需要给 `Graphics` 增加几个绘图函数，同时保持二进制兼容性。这几个新函数的坐标以浮点数表示，我理想中的新接口是：

```
--- old/graphics.h 2011-03-12 13:12:44.000000000 +0800
+++ new/graphics.h 2011-03-12 13:13:30.000000000 +0800
@@ -7,11 +7,14 @@
class Graphics
{
    virtual void drawLine(int x0, int y0, int x1, int y1);
+   virtual void drawLine(double x0, double y0, double x1, double y1);
    virtual void drawLine(Point p0, Point p1);

    virtual void drawRectangle(int x0, int y0, int x1, int y1);
+   virtual void drawRectangle(double x0, double y0, double x1, double y1);
    virtual void drawRectangle(Point p0, Point p1);
```



```

    virtual void drawArc(int x, int y, int r);
+   virtual void drawArc(double x, double y, double r);
    virtual void drawArc(Point p, int r);
};

```

受 C++ 二进制兼容性方面的限制，我们不能这么做。其本质问题在于 C++ 以 `vtable[offset]` 方式实现虚函数调用，而 `offset` 又是根据虚函数声明的位置隐式确定的，这造成了脆弱性。我增加了 `drawLine(double x0, double y0, double x1, double y1)`，造成 `vtable` 的排列发生了变化，现有的二进制可执行文件无法再用旧的 `offset` 调用到正确的函数。

怎么办呢？有一种危险且丑陋的做法：把新的虚函数放到 `interface` 的末尾，例如：

```

--- old/graphics.h 2011-03-12 13:12:44.000000000 +0800
+++ new/graphics.h 2011-03-12 13:58:22.000000000 +0800
@@ -7,11 +7,15 @@
class Graphics
{
    virtual void drawLine(int x0, int y0, int x1, int y1);
    virtual void drawLine(Point p0, Point p1);

    virtual void drawRectangle(int x0, int y0, int x1, int y1);
    virtual void drawRectangle(Point p0, Point p1);

    virtual void drawArc(int x, int y, int r);
    virtual void drawArc(Point p, int r);
+
+   virtual void drawLine(double x0, double y0, double x1, double y1);
+   virtual void drawRectangle(double x0, double y0, double x1, double y1);
+   virtual void drawArc(double x, double y, double r);
};

```

这么做很丑陋，因为新的 `drawLine(double x0, double y0, double x1, double y1)` 函数没有和原来的 `drawLine()` 函数呆在一起，造成阅读上的不便。这么做同时很危险，因为 `Graphics` 如果被继承，那么新增虚函数会改变派生类中的 `vtable offset` 变化，同样不是二进制兼容的。

另外有两种似乎安全的做法，这也是 COM 采用的办法：

1. 通过链式继承来扩展现有 `interface`，例如从 `Graphics` 派生出 `Graphics2`。

```

--- graphics.h 2011-03-12 13:12:44.000000000 +0800
+++ graphics2.h 2011-03-12 13:58:35.000000000 +0800
@@ -7,11 +7,19 @@
class Graphics

```

```

{
    virtual void drawLine(int x0, int y0, int x1, int y1);
    virtual void drawLine(Point p0, Point p1);

    virtual void drawRectangle(int x0, int y0, int x1, int y1);
    virtual void drawRectangle(Point p0, Point p1);

    virtual void drawArc(int x, int y, int r);
    virtual void drawArc(Point p, int r);
};
+
+class Graphics2 : public Graphics
+{
+    using Graphics::drawLine;
+    using Graphics::drawRectangle;
+    using Graphics::drawArc;
+
+    // added in version 2
+    virtual void drawLine(double x0, double y0, double x1, double y1);
+    virtual void drawRectangle(double x0, double y0, double x1, double y1);
+    virtual void drawArc(double x, double y, double r);
+};

```

将来如果继续增加功能，那么还会有 `class Graphics3 : public Graphics2;` 以及 `class Graphics4 : public Graphics3;` 等等。这么做和前面的做法一样丑陋，因为新的 `drawLine(double x0, double y0, double x1, double y1)` 函数位于派生 `Graphics2` `interface` 中，没有和原来的 `drawLine()` 函数呆在一起，造成割裂。

2. 通过多重继承来扩展现有 `interface`，例如定义一个与 `Graphics` `class` 有同样成员的 `Graphics2`，再让实现同时继承这两个 `interfaces`。

```

--- graphics.h 2011-03-12 13:12:44.000000000 +0800
+++ graphics2.h 2011-03-12 13:16:45.000000000 +0800
@@ -7,11 +7,32 @@
class Graphics
{
    virtual void drawLine(int x0, int y0, int x1, int y1);
    virtual void drawLine(Point p0, Point p1);

    virtual void drawRectangle(int x0, int y0, int x1, int y1);
    virtual void drawRectangle(Point p0, Point p1);

    virtual void drawArc(int x, int y, int r);
    virtual void drawArc(Point p, int r);
};
+
+class Graphics2
+{
+    virtual void drawLine(int x0, int y0, int x1, int y1);
+    virtual void drawLine(double x0, double y0, double x1, double y1);

```

```
+ virtual void drawLine(Point p0, Point p1);
+
+ virtual void drawRectangle(int x0, int y0, int x1, int y1);
+ virtual void drawRectangle(double x0, double y0, double x1, double y1);
+ virtual void drawRectangle(Point p0, Point p1);
+
+ virtual void drawArc(int x, int y, int r);
+ virtual void drawArc(double x, double y, double r);
+ virtual void drawArc(Point p, int r);
+};
+
+// 在实现中采用多重接口继承
+class GraphicsImpl : public Graphics, // version 1
+                    public Graphics2, // version 2
+{
+    // ...
+};
```

这种带版本的 `interface` 的做法在 COM 使用者的眼中看起来是很正常的（比如 `IXMLDOMDocument`、`IXMLDOMDocument2`、`IXMLDOMDocument3`，又比如 `ITaskbarList`、`ITaskbarList2`、`ITaskbarList3`、`ITaskbarList4` 等等），解决了二进制兼容性的问题，客户端源代码也不受影响。

在我看来带版本的 `interface` 实在是很丑陋，因为每次改动都引入了新的 `interface class`，会造成日后客户端代码难以管理。比如，如果新版应用程序的代码使用了 `Graphics3` 的功能，要不要把现有代码中出现的 `Graphics2` 都替换掉？

- 如果不替换，一个程序同时依赖多个版本的 `Graphics`，一直背着历史包袱。依赖的 `Graphics` 版本越积越多，将来如何管理得过来？
- 如果要替换，为什么不相干的代码（现有的运行得好好的使用 `Graphics2` 的代码）也会因为别处用到了 `Graphics3` 而被修改？

这种二难境地纯粹是“以虚函数为库的接口”造成的。如果我们能直接原地扩充 `class Graphics`，就不会有这些屁事，见下一节“动态库接口的推荐做法”。

5.4 假如 Linux 系统调用以 COM 接口方式实现

或许上面这个 `Graphics` 的例子太简单，没有让“以虚函数为接口”的缺点充分暴露出来，让我们看一个真实的案例：Linux Kernel。

Linux kernel 从 0.10 的 67 个¹² 系统调用发展到 2.6.37 的 340 个¹³, kernel interface 一直在扩充, 而且保持良好的兼容性, 它保持兼容性的办法很土, 就是给每个 system call 赋予一个终身不变的数字代号, 等于把虚函数表的排列固定下来。点开本段开头的两个链接, 你就能看到 fork() 在 Linux 0.10 和 Linux 2.6.37 里的代号都是 2。(系统调用的编号跟硬件平台有关, 这里我们看的是 x86 32-bit 平台。)

试想假如 Linus 当初选择用 COM 接口的链式继承风格来描述, 将会是怎样一种壮观的景象? 为了避免扰乱视线, 请移步观看近百层继承的代码¹⁴。(先后关系与版本号不一定 100% 准确, 我是用 git blame 去查的, 现在列出的代码只从 0.01 到 2.5.31, 相信已经足以展现 COM 接口方式的弊端。)

不要误认为“接口一旦发布就不能更改”是天经地义的, 那不过是“以 C++ 虚函数为接口”的固有弊端, 如果跳出这个框框去思考, 其实 C++ 库的接口很容易做得更好。

为什么不能改? 还不是因为用了 C++ 虚函数作为接口。Java 的 interface 可以添加新函数, C 语言的库也可以添加新的全局函数, C++ class 也可以添加新 non-virtual 成员函数和 namespace 级别的 non-member 函数, 这些都不需要继承出新 interface 就能扩充原有接口。偏偏 COM 的 interface 不能原地扩充, 只能通过继承来 workaround, 产生一堆带版本的 interfaces。有人说 COM 是二进制兼容性的正面例子, 某深不以为然。COM 确实以一种最丑陋的方式做到了“二进制兼容”。脆弱与僵硬就是以 C++ 虚函数为接口的宿命。

相反, Linux 系统调用以编译期常数方式固定下来, 万年不变, 轻而易举地解决了这个问题。在其他面向对象语言 (Java/C#) 中, 我也没有见过每改动一次就给 interface 递增版本号的诡异做法。

还是应了《The Zen of Python》中的那句话, Explicit is better than implicit, Flat is better than nested.

5.5 Java 是如何应对的

Java 实际上把 C/C++ 的 linking 这一步骤推迟到 class loading 的时候来做。就不存在“不能增加虚函数”, “不能修改 data member”等问题。在 Java 里边用面向 interface 编程远比 C++ 更通用和自然, 也没有上面提到的“僵硬的接口”问题。

¹²<http://lxr.linux.no/linux-old+v0.01/include/unistd.h#L60>

¹³http://lxr.linux.no/linux+v2.6.37.3/arch/x86/include/asm/unistd_32.h

¹⁴<https://gist.github.com/867174>

6 动态库的接口的推荐做法

取决于动态库的使用范围，有两类做法。

其一，如果动态库的使用范围比较窄，比如本团队内部的两三个程序在用，用户都是受控的，要发布新版本也比较容易协调，那么不用太费事，只要做好发布的版本管理就行了。再在可执行文件中使用 `rpath` 把库的完整路径确定下来。

比如现在 `Graphics` 库发布了 1.1.0 和 1.2.0 两个版本，这两个版本可以不必是二进制兼容。用户的代码从 1.1.0 升级到 1.2.0 的时候要重新编译一下，反正他们要用新功能都是要重新编译代码的。如果要原地打补丁，那么 1.1.1 应该和 1.1.0 二进制兼容，而 1.2.1 应该和 1.2.0 兼容。如果要加入新的功能，而新的功能与 1.2.0 不兼容，那么应该发布到 1.3.0 版本。

为了便于检查二进制兼容性，可考虑把库的代码的暴露情况分辨清楚。`muduo` 的头文件和 `class` 就有意识地分为用户可见和用户不可见两部分，见 <http://blog.csdn.net/Solstice/archive/2010/08/29/5848547.aspx>。对于用户可见的部分，升级时要注意二进制兼容性，选用合理的版本号；对于用户不可见的部分，在升级库的时候就不必在意。另外 `muduo` 本身设计来是以原文件方式发布，在二进制兼容性方面没有做太多的考虑。

其二，如果库的使用范围很广，用户很多，各家的 `release cycle` 不尽相同，那么推荐 `pimpl`¹⁵ 技法 [4, item 43]，并考虑多采用 `non-member non-friend function in namespace` [2, item 23] [4, item 44 and 57] 作为接口。这里以前面的 `Graphics` 为例，说明 `pimpl` 的基本手法。

1. 暴露的接口里边不要有虚函数，而且
`sizeof(Graphics) == sizeof(Graphics::Impl*)`。

```
class Graphics                                     graphics.h
{
public:
    Graphics(); // outline ctor
    ~Graphics(); // outline dtor

    void drawLine(int x0, int y0, int x1, int y1);
    void drawLine(Point p0, Point p1);

    void drawRectangle(int x0, int y0, int x1, int y1);
    void drawRectangle(Point p0, Point p1);
```

¹⁵`pimpl` 是 `pointer to implementation` 的缩写。

```
void drawArc(int x, int y, int r);
void drawArc(Point p, int r);

private:
    class Impl;
    boost::scoped_ptr<Impl> impl;
};
```

graphics.h

2. 在库的实现中把调用转发 (forward) 给实现 `Graphics::Impl`，这部分代码位于 `.so/.dll` 中，随库的升级一起变化。

```
#include <graphics.h>

class Graphics::Impl
{
public:
    void drawLine(int x0, int y0, int x1, int y1);
    void drawLine(Point p0, Point p1);

    void drawRectangle(int x0, int y0, int x1, int y1);
    void drawRectangle(Point p0, Point p1);

    void drawArc(int x, int y, int r);
    void drawArc(Point p, int r);
};

Graphics::Graphics()
    : impl(new Impl)
{
}

Graphics::~Graphics()
{
}

void Graphics::drawLine(int x0, int y0, int x1, int y1)
{
    impl->drawLine(x0, y0, x1, y1);
}

void Graphics::drawLine(Point p0, Point p1)
{
    impl->drawLine(p0, p1);
}

// ...
```

graphics.cc

3. 如果要加入新的功能，不必通过继承来扩展，可以原地修改，且很容易保持二进制兼容性。先动头文件：

```
--- old/graphics.h      2011-03-12 15:34:06.000000000 +0800
+++ new/graphics.h      2011-03-12 15:14:12.000000000 +0800
@@ -7,19 +7,22 @@
class Graphics
{
public:
    Graphics(); // outline ctor
    ~Graphics(); // outline dtor

    void drawLine(int x0, int y0, int x1, int y1);
+   void drawLine(double x0, double y0, double x1, double y1);
    void drawLine(Point p0, Point p1);

    void drawRectangle(int x0, int y0, int x1, int y1);
+   void drawRectangle(double x0, double y0, double x1, double y1);
    void drawRectangle(Point p0, Point p1);

    void drawArc(int x, int y, int r);
+   void drawArc(double x, double y, double r);
    void drawArc(Point p, int r);

private:
    class Impl;
    boost::scoped_ptr<Impl> impl;
};
```

然后在实现文件里增加 **forward**，这么做不会破坏二进制兼容性，因为增加 **non-virtual** 函数不影响现有的可执行文件。

```
--- old/graphics.cc      2011-03-12 15:15:20.000000000 +0800
+++ new/graphics.cc      2011-03-12 15:15:26.000000000 +0800
@@ -1,35 +1,43 @@
#include <graphics.h>

class Graphics::Impl
{
public:
    void drawLine(int x0, int y0, int x1, int y1);
+   void drawLine(double x0, double y0, double x1, double y1);
    void drawLine(Point p0, Point p1);

    void drawRectangle(int x0, int y0, int x1, int y1);
+   void drawRectangle(double x0, double y0, double x1, double y1);
    void drawRectangle(Point p0, Point p1);

    void drawArc(int x, int y, int r);
+   void drawArc(double x, double y, double r);
    void drawArc(Point p, int r);
```

```
};

Graphics::Graphics()
: impl(new Impl)
{
}

Graphics::~Graphics()
{
}

void Graphics::drawLine(int x0, int y0, int x1, int y1)
{
    impl->drawLine(x0, y0, x1, y1);
}

+void Graphics::drawLine(double x0, double y0, double x1, double y1)
+{
+    impl->drawLine(x0, y0, x1, y1);
+}
+
+void Graphics::drawLine(Point p0, Point p1)
+{
+    impl->drawLine(p0, p1);
+}
```

采用 `pimpl` 多了一道 `explicit forward` 的手续，带来的好处是可扩展性与二进制兼容性，通常是划算的。`pimpl` 扮演了编译器防火墙的作用。

`pimpl` 不仅 C++ 语言可以用，C 语言的库同样可以用，一样带来二进制兼容性的好处，比如 `libevent2` 里边的 `struct event_base` 是个 `opaque pointer`，客户端看不到其成员，都是通过 `libevent` 的函数和它打交道，这样库的版本升级比较容易做到二进制兼容。

为什么 `non-virtual` 函数比 `virtual` 函数更健壮？因为 `virtual function` 是 `bind-by-vtable-offset`，而 `non-virtual function` 是 `bind-by-name`。加载器 (loader) 会在程序启动时做决议 (resolution)，通过 `mangled name` 把可执行文件和动态库链接到一起。就像使用 `Internet` 域名比使用 `IP` 地址更能适应变化一样。

万一要跨语言怎么办？很简单，暴露 C 语言的接口。Java 有 `JNI` 可以调用 C 语言的代码；Python/Perl/Ruby 等等的解释器都是 C 语言编写的，使用 C 函数也不在话下。C 函数是 Linux 下的万能接口。

本文只谈了使用 `class` 为接口，其实用 `free function` 有时候更好（比如 `muduo/base/Timestamp.h` 除了定义 `class Timestamp`，还定义了 `muduo::timeDifference()` 等 `free function`），这也是 C++ 比 Java 等纯面向对象语言优越的地方。留给

将来再细谈吧。

7 以 boost::function 和 boost::bind 取代虚函数

这篇文章的中心思想是“面向对象的继承就像一条贼船，上去就下不来了”，而借助 boost::function 和 boost::bind，大多数情况下，你都不用上贼船。

boost::function 和 boost::bind 已经纳入了 std::tr1，这或许是 C++0x 最值得期待的功能，它将彻底改变 C++ 库的设计方式，以及应用程序的编写方式。

Scott Meyers 的 Effective C++ 3rd ed [2] 第 35 条款提到了以 boost::function 和 boost::bind 取代虚函数的做法，这里谈谈我自己使用的感受。

（这篇文章写得比较早，那会儿我还没有开始写 muduo，所以文章的例子与现在的代码有些脱节。另见孟岩《function/bind 的救赎（上）》¹⁶，《回复几个问题》¹⁷中的“四个半抽象”）

7.1 基本用途

boost::function 就像 C# 里的 delegate，可以指向任何函数，包括成员函数。当用 bind 把某个成员函数绑到某个对象上时，我们得到了一个 closure（闭包）。例如：

```
class Foo
{
public:
    void methodA();
    void methodInt(int a);
};

class Bar
{
public:
    void methodB();
};

boost::function<void()> f1; // 无参数，无返回值

Foo foo;
```

¹⁶<http://blog.csdn.net/myan/archive/2010/10/09/5928531.aspx>

¹⁷<http://blog.csdn.net/myan/archive/2010/09/14/5884695.aspx>

```
f1 = boost::bind(&Foo::methodA, &foo);
f1(); // 调用 foo.methodA();
Bar bar;
f1 = boost::bind(&Bar::methodB, &bar);
f1(); // 调用 bar.methodB();

f1 = boost::bind(&Foo::methodInt, &foo, 42);
f1(); // 调用 foo.methodInt(42);

boost::function<void(int)> f2; // int 参数, 无返回值
f2 = boost::bind(&Foo::methodInt, &foo, _1);
f2(53); // 调用 foo.methodInt(53);
```

如果没有 `boost::bind`, 那么 `boost::function` 就什么都不是, 而有了 `bind`, “同一个类的不同对象可以 `delegate` 给不同的实现, 从而实现不同的行为” (myan 语), 简直就无敌了。

7.2 对程序库的影响

程序库的设计不应该给使用者带来不必要的限制 (耦合), 而继承是第二强的一种耦合 (最强耦合的是友元)。如果一个程序库限制其使用者必须从某个 `class` 派生, 那么我觉得这是一个糟糕的设计。不巧的是, 目前不少 C++ 程序库就是这么做的。

例 1: 线程库

常规 OO 设计 :

写一个 `Thread` base class, 含有 (纯) 虚函数 `Thread::run()`, 然后应用程序派生一个 `derived class`, 覆写 `run()`。程序里的每一种线程对应一个 `Thread` 的派生类。例如 Java 的 `Thread` class 可以这么用。

缺点: 如果一个 `class` 的三个 `method` 需要在三个不同的线程中执行, 就得写 `helper class(es)` 并玩一些 OO 把戏。

基于 `boost::function` 的设计 :

令 `Thread` 是一个具体类, 其构造函数接受 `Callable` 对象。应用程序只需提供一个 `Callable` 对象, 创建一份 `Thread` 实体, 调用 `Thread::start()` 即可。Java 的 `Thread` 也可以这么用, 传入一个 `Runnable` 对象。C# 的 `Thread` 只支持这种用法, 构造函数的参数是 `delegate ThreadStart`。 `boost::thread` 也只支持这种用法。

```
// 一个基于 boost::function 的 Thread class 基本结构
class Thread
{
public:
    typedef boost::function<void()> ThreadCallback;

    Thread(ThreadCallback cb) : cb_(cb)
    { }

    void start()
    {
        /* some magic to call run() in new created thread */
    }

private:
    void run()
    {
        cb_();
    }

    ThreadCallback cb_;
    // ...
};
```

使用:

```
class Foo // 不需要继承
{
public:
    void runInThread();
    void runInAnotherThread(int)
};

Foo foo;
Thread thread1(boost::bind(&Foo::runInThread, &foo));
Thread thread2(boost::bind(&Foo::runInAnotherThread, &foo, 43));
thread1.start();
thread2.start();
```

例 2: 网络库

以 boost::function 作为桥梁, NetServer class 对其使用者没有任何类型上的限制, 只对成员函数的参数和返回类型有限制。使用者 EchoService 也完全不知道 NetServer 的存在, 只要在 main() 里把两者装配到一起, 程序就跑起来了。

```
class Connection;
class NetServer : boost::noncopyable
```

network library

```

{
public:
    typedef boost::function<void (Connection*)> ConnectionCallback;
    typedef boost::function<void (Connection*,
                                const void*,
                                int len)> MessageCallback;

    NetServer(uint16_t port);
    ~NetServer();
    void registerConnectionCallback(const ConnectionCallback&);
    void registerMessageCallback(const MessageCallback&);
    void sendMessage(Connection*, const void* buf, int len);

private:
    // ...
};

```

network library

user code

```

class EchoService
{
public:
    // 符合 NetServer::sendMessage 的原型
    typedef boost::function<void(Connection*,
                                const void*,
                                int)> SendMessageCallback;

    EchoService(const SendMessageCallback& sendMsgCb)
        : sendMessageCb_(sendMsgCb) // 保存 boost::function
    { }

    // 符合 NetServer::MessageCallback 的原型
    void onMessage(Connection* conn, const void* buf, int size)
    {
        printf("Received Msg from Connection %d: %.s\n",
              conn->id(), size, (const char*)buf);
        sendMessageCb_(conn, buf, size); // echo back
    }

    // 符合 NetServer::ConnectionCallback 的原型
    void onConnection(Connection* conn)
    {
        printf("Connection from %s:%d is %s\n",
              conn->ipAddr(),
              conn->port(),
              conn->connected() ? "UP" : "DOWN");
    }

private:
    SendMessageCallback sendMessageCb_;
};

```

// 扮演上帝的角色，把各部件拼起来

```
int main()
{
    NetServer server(7);
    EchoService echo(bind(&NetServer::sendMessage, &server, _1, _2, _3));
    server.registerMessageCallback(
        bind(&EchoService::onMessage, &echo, _1, _2, _3));
    server.registerConnectionCallback(
        bind(&EchoService::onConnection, &echo, _1));
    server.run();
}
```

user code

7.3 对面向对象程序设计的影响

一直以来，我对面向对象有一种厌恶感，叠床架屋，绕来绕去的，一拳拳打在棉花上，不解决实际问题。面向对象三要素是封装、继承和多态。我认为封装是根本的，继承和多态则是可有可无。用 `class` 来表示 `concept`，这是根本的；至于继承和多态，其耦合性太强，往往不划算。

继承和多态不仅规定了函数的名称、参数、返回类型，还规定了类的继承关系。在现代的 OO 编程语言里，借助反射和 `attribute/annotation`，已经大大放宽了限制。举例来说，JUnit 3.x 是用反射，找出派生类里的名字符合 `void test*()` 的函数来执行，这里就没继承什么事，只是对函数的名称有部分限制（继承是全面限制，一字不差）。至于 JUnit 4.x 和 NUnit 2.x 则更进一步，以 `annoatation/attribute` 来标明 `test case`，更没继承什么事了。

我的猜测是，当初提出面向对象的时候，`closure` 还没有一个通用的实现，所以它没能算作基本的抽象工具之一。现在既然 `closure` 已经这么方便了，或许我们应该重新审视面向对象设计，至少不要那么滥用继承。

自从找到了 `boost::function+boost::bind` 这对神兵利器，不用再考虑类直接的继承关系，只需要基于对象的设计 (`object-based`)，拳拳到肉，程序写起来顿时顺手了很多。

7.4 对面向对象设计模式的影响

既然虚函数能用 `closure` 代替，那么很多 OO 设计模式，尤其是行为模式，失去了存在的必要。另外，既然没有继承体系，那么创建型模式似乎也没啥用了。

最明显的是 `Strategy`，不用累赘的 `Strategy` 基类和 `ConcreteStrategyA`、`ConcreteStrategyB` 等派生类，一个 `boost::function<>` 成员就解决问题。另外一个例

子是 Command 模式，有了 boost::function，函数调用可以直接变成对象，似乎就没 Command 什么事了。同样的道理，Template Method 可以不必使用基类与继承，只要传入几个 boost::function 对象，在原来调用虚函数的地方换成调用 boost::function 对象就能解决问题。

在《设计模式》这本书提到了 23 个模式，我认为 iterator 有用（或许再加个 State），其他都在摆谱，拉虚架子，没啥用。或许它们解决了面向对象中的常见问题，不过要是我的程序里连面向对象（指继承和多态）都不用，那似乎也不用叨扰面向对象设计模式了。

或许 closure-based programming 将作为一种新的 programming paradigm 而流行起来。

7.5 依赖注入与单元测试

前面的 EchoService 可算是依赖注入的例子，EchoService 需要一个什么东西来发送消息，它对这个“东西”的要求只是函数原型满足 SendMessageCallback，而并不关系数据到底发到网络上还是发到控制台。在正常使用的时候，数据应该发给网络，而在做单元测试的时候，数据应该发给某个 DataSink。

按照面向对象的思路，先写一个 AbstractDataSink interface，包含 sendMessage() 这个虚函数，然后派生出两个 classes: NetDataSink 和 MockDataSink，前面那个干活用，后面那个单元测试用。EchoService 的构造函数应该以 AbstractDataSink* 为参数，这样就实现了所谓的接口与实现分离。

我认为这么做纯粹是脱了裤子放屁，直接传入一个 SendMessageCallback 对象就能解决问题。在单元测试的时候，可以 boost::bind() 到 MockServer 上，或某个全局函数上，完全不用继承和虚函数，也不会影响现有的设计。

7.6 什么时候使用继承？

如果是指 OO 中的 public 继承，即为了接口与实现分离，那么我只会在派生类的数目和功能完全确定的情况下使用。换句话说，不为将来的扩展考虑，这时候面向对象或许是一种不错的描述方法。一旦要考虑扩展，什么办法都没用，还不如把程序写简单点，将来好大改或重写。

如果是功能继承，那么我会考虑继承 `boost::noncopyable` 或 `boost::enable_shared_from_this`，下一篇 blog 会讲到 `enable_shared_from_this` 在实现多线程安全的对象回调时的妙用。

例如，IO-Multiplex 在不同的操作系统下有不同的推荐实现，最通用的 `select()`，POSIX 的 `poll()`，Linux 的 `epoll()`，FreeBSD 的 `kqueue` 等等，数目固定，功能也完全确定，不用考虑扩展。那么设计一个 `NetLoop base class` 加若干具体 `classes` 就是不错的解决办法。换句话说，用多态来代替 `switch-case` 以达到简化代码的目的。

7.7 基于接口的设计

这个问题来自那个经典的讨论：不会飞的企鹅（Penguin）究竟应不应该继承自鸟（Bird），如果 Bird 定义了 `virtual function fly()` 的话。讨论的结果是，把具体的行为提出来，作为 `interface`，比如 `Flyable`（能飞的），`Runnable`（能跑的），然后让企鹅实现 `Runnable`，麻雀实现 `Flyable` 和 `Runnable`。（其实麻雀只能双脚跳，不能跑，这里不作深究。）

进一步的讨论表明，`interface` 的粒度应足够小，或许包含一个 `method` 就够了，那么 `interface` 实际上退化成了给类型打的标签（tag）。在这种情况下，完全可以使用 `boost::function` 来代替，比如：

```
// 企鹅能游泳，也能跑
class Penguin
{
public:
    void run();
    void swim();
};

// 麻雀能飞，也能跑
class Sparrow
{
public:
    void fly();
    void run();
};

// 以 boost::function 作为接口
typedef boost::function<void()> FlyCallback;
typedef boost::function<void()> RunCallback;
typedef boost::function<void()> SwimCallback;
```

```
// 一个既用到 run, 也用到 fly 的客户 class
class Foo
{
public:
    Foo(FlyCallback flyCb, RunCallback runCb)
        : flyCb_(flyCb), runCb_(runCb)
    { }

private:
    FlyCallback flyCb_;
    RunCallback runCb_;
};

// 一个既用到 run, 也用到 swim 的客户 class
class Bar
{
public:
    Bar(SwimCallback swimCb, RunCallback runCb)
        : swimCb_(swimCb), runCb_(runCb)
    { }

private:
    SwimCallback swimCb_;
    RunCallback runCb_;
};

int main()
{
    Sparrow s;
    Penguin p;
    // 装配起来, Foo 要麻雀, Bar 要企鹅。
    Foo foo(bind(&Sparrow::fly, &s), bind(&Sparrow::run, &s));
    Bar bar(bind(&Penguin::swim, &p), bind(&Penguin::run, &p));
}
```

最后, 向伟大的 C 语言致敬!

8 带符号整数的除法与余数

最近研究整数到字符串的转换, 读到了 Matthew Wilson 的《Efficient Integer to String Conversions》系列文章。¹⁸ 他的巧妙之处在于, 用一个对称的 `digits` 数组搞定了负数转换的边界条件 (二进制补码的正负整数表示范围不对称)。代码大致如下, 经过改写:

¹⁸<http://synesis.com.au/publications.html> 搜 conversions

```

const char* convert(char buf[], int value)
{
    static char digits[19] =
        { '9', '8', '7', '6', '5', '4', '3', '2', '1',
          '0', '1', '2', '3', '4', '5', '6', '7', '8', '9' };
    static const char* zero = digits + 9; // zero 指向 '0'
    // works for -2147483648 .. 2147483647
    int i = value;
    char* p = buf;
    do {
        // lsd - least significant digit
        int lsd = i % 10; // lsd 可能小于 0
        i /= 10;          // 是向下取整还是向零取整？
        *p++ = zero[lsd]; // 下标可能为负
    } while (i != 0);
    if (value < 0) {
        *p++ = '-';
    }
    *p = '\0';
    std::reverse(buf, p);
    return p; // p - buf 即为整数长度
}

```

这段简短的代码对 32-bit int 的全部取值都是正确的（从 -2147483648 到 2147483647）。可以视为 itoa() 的参考实现，面试的标准答案。

读到这份代码，我心中顿时升起一个疑虑：《C Traps and Pitfalls》第 7.7 节¹⁹

讲到，C 语言中的整数除法 (/) 和取模 (%) 运算在操作数为负的时候，结果是 implementation-defined。

也就是说，如果 m、d 都是整数，

```

int q = m / d;
int r = m % d;

```

那么 C 语言只保证 $m = q \times d + r$ 。如果 m、d 当中有负数，那么 q 和 r 的正负号是由实现决定的。比如 $(-13)/4 = (-3)$ 或 $(-13)/4 = (-4)$ 都是合法的。如果采用后一种实现，那么这段转换代码就错了（因为将有 $(-1)\%10 = 9$ ）。只有商向 0 取整，代码才能正常工作。

为了弄清这个问题，我研究了一番。

¹⁹网上能下载到的一份简略版也有相同的内容，<http://www.literateprogramming.com/ctraps.pdf> 第 7.5 节。

8.1 语言标准怎么说

C89 我手头没有 ANSI C89 的文稿，只好求助于 K&R88，此书第 41 页第 2.5 节讲到 *The direction of truncation for / and the sign of the result for % are machine-dependent for negative operands, ...*。确实是实现相关的。为此，C89 专门提供了 `div()` 函数，这个函数算出的商是向 0 取整的，便于编写可移植的程序。我得再去查 C++ 标准。

C++98 第 5.6.4 节写到 *If the second operand of / or % is zero the behavior is undefined; otherwise (a/b)*b + a%b is equal to a. If both operands are nonnegative then the remainder is nonnegative; if not, the sign of the remainder is implementation-defined.* C++ 也没有规定余数的正负号（C++03 的叙述一模一样）。

不过这里有一个注脚，提到 *According to work underway toward the revision of ISO C, the preferred algorithm for integer division follows the rules defined in the ISO Fortran standard, ISO/IEC 1539:1991, in which the quotient is always rounded toward zero.* 即 C 语言的修订标准会采用和 Fortran 一样的取整算法。我又去查了 C99。

C99 第 6.5.5.6 节说 *When integers are divided, the result of the / operator is the algebraic quotient with any fractional part discarded.*（脚注：*This is often called "truncation toward zero".*）

C99 明确规定了商是向 0 取整，也就意味着余数的符号与被除数相同，前面的转换算法能正常工作。C99 Rationale²⁰ 提到了这个规定的原因，*In Fortran, however, the result will always truncate toward zero, and the overhead seems to be acceptable to the numeric programming community. Therefore, C99 now requires similar behavior, which should facilitate porting of code from Fortran to C.* 既然 Fortran 在数值计算领域都做了如此规定，说明开销（如果有的话）是可以接受的。

C++0x（x 已经确定无疑是个十六进制数了）

最近的 n2800 草案第 5.6.4 节采用了与 C99 类似的表述：*For integral operands the / operator yields the algebraic quotient with any fractional part discarded; (This is often called truncation towards zero.)* 可见 C++ 还是尽力保持与 C 的兼容性。

小结：C89 和 C++98 都留给实现去决定，而 C99 和 C++0x 都规定商向 0 取整，这算是语言的进步吧。

²⁰<http://www.open-std.org/jtc1/sc22/wg14/www/C99RationaleV5.10.pdf>

8.2 C/C++ 编译器的表现

我主要关心 G++ 和 VC++ 这两个编译器。需要说明的是，用代码案例来探查编译器的行为是靠不住的，尽管前面的代码在两个编译器下都能正常工作。除非在文档里有明确表述，否则编译器可能会随时更改实现——毕竟我们关心的就是 *implementation-defined* 行为。

G++ 4.4 ²¹ GCC always follows the C99 requirement that the result of division is truncated towards zero. G++ 一直遵循 C99 规范，商向 0 取整，算法能正常工作。

Visual C++ 2008 ²² The sign of the remainder is the same as the sign of the dividend. 这个说法与商向 0 取整是等价的，算法也能正常工作。

8.3 其他语言的规定

既然 C89/C++98/C99/C++0x 已经很有多样性了，索性弄清楚其他语言是怎么定义整数除法的。这里只列出我（陈硕）接触过的几种常用语言。

Java Java 语言规范²³ 明确说 Integer division rounds toward 0. 另外对于 int 整数除法溢出，特别规定不抛异常，且 $-2147483648 / -1 = -2147483648$ （以及相应的 long 版本）。

C# ²⁴ C# 3.0 语言规定 The division rounds the result towards zero. 对于溢出的情况，规定在 checked 上下文中抛 ArithmeticException 异常；在 unchecked 上下文里没有明确规定，可抛可不抛。（据了解，C# 1.0/2.0 可能有所不同。）

Python Python 在语言参考手册²⁵ 的显著位置标明，商是向负无穷取整。Plain or long integer division yields an integer of the same type; the result is that of mathematical division with the 'floor' function applied to the result.

²¹<http://gcc.gnu.org/onlinedocs/gcc/Integers-implementation.html>

²²<http://msdn.microsoft.com/en-us/library/eayc4fzk.aspx>

²³http://java.sun.com/docs/books/jls/third_edition/html/expressions.html#15.17.2

²⁴<http://msdn.microsoft.com/en-us/vcsharp/aa336809.aspx>

²⁵<http://docs.python.org/reference/expressions.html#binary-arithmetic-operations>

Ruby Ruby 的语言手册没有明说, 不过库的手册说到也是向负无穷取整。The quotient is rounded toward -infinity.²⁶

Perl²⁷ Perl 语言默认按浮点数来计算除法, 所以没有这个问题。Perl 的整数取模运算规则与 Python/Ruby 一致。

不过要注意, `use integer;` 有可能会改变运算结果, 例如。

```
print -10 % 3; // => 2

use integers;
print -10 % 3; // => -1
```

Lua Lua 缺省没有整数类型, 除法一律按浮点数来算, 因此不涉及商的取整问题。

可以看出, 在整数除法的取整问题上, 语言分为两个阵营, 脚本语言彼此是相似的, C99/C++0x/Java/C# 则属于另一个阵营。既然 Python 和 Ruby 都是用 C 实现的, 但是运算规则又自成一体, 那么必定能从代码中找到证据。

8.4 脚本语言解释器代码

Python 的代码很好读, 我很快就找到了 2.6.6 版实现整数除法和取模运算的函数 `i_divmod()`²⁸

python/tags/r266/Objects/intobject.c

```
565 /* Return type of i_divmod */
566 enum divmod_result {
567     DIVMOD_OK,                /* Correct result */
568     DIVMOD_OVERFLOW,          /* Overflow, try again using longs */
569     DIVMOD_ERROR              /* Exception raised */
570 };
571
572 static enum divmod_result
573 i_divmod(register long x, register long y,
574          long *p_xdivy, long *p_xmody)
575 {
576     long xdivy, xmody;
577
578     if (y == 0) {
```

²⁶http://www.ruby-doc.org/docs/ProgrammingRuby/html/ref_c_numeric.html#Numeric.divmod

²⁷<http://perldoc.perl.org/perlop.html#Multiplicative-Operators>

²⁸<http://svn.python.org/view/python/tags/r266/Objects/intobject.c?view=markup>

```

579         PyErr_SetString(PyExc_ZeroDivisionError,
580                         "integer division or modulo by zero");
581         return DIVMOD_ERROR;
582     }
583     /* (-sys.maxint-1)/-1 is the only overflow case. */
584     if (y == -1 && UNARY_NEG_WOULD_OVERFLOW(x))
585         return DIVMOD_OVERFLOW;
586     xdivy = x / y;
587     /* xdivy*y can overflow on platforms where x/y gives floor(x/y)
588      * for x and y with differing signs. (This is unusual
589      * behaviour, and C99 prohibits it, but it's allowed by C89;
590      * for an example of overflow, take x = LONG_MIN, y = 5 or x =
591      * LONG_MAX, y = -5.) However, x - xdivy*y is always
592      * representable as a long, since it lies strictly between
593      * -abs(y) and abs(y). We add casts to avoid intermediate
594      * overflow.
595      */
596     xmody = (long)(x - (unsigned long)xdivy * y);
597     /* If the signs of x and y differ, and the remainder is non-0,
598      * C89 doesn't define whether xdivy is now the floor or the
599      * ceiling of the infinitely precise quotient. We want the floor,
600      * and we have it iff the remainder's sign matches y's.
601      */
602     if (xmody && ((y ^ xmody) < 0) /* i.e. and signs differ */) {
603         xmody += y;
604         --xdivy;
605         assert(xmody && ((y ^ xmody) >= 0));
606     }
607     *p_xdivy = xdivy;
608     *p_xmody = xmody;
609     return DIVMOD_OK;
610 }

```

python/tags/r266/Objects/intobject.c

注意到这段代码甚至考虑了 $-2147483648 / -1$ 在 32-bit 下会溢出这个特殊情况，让我大吃一惊。宏定义 `UNARY_NEG_WOULD_OVERFLOW` 和函数 `int_mul()` 前面的注释也值得一读。

```

554 /* Integer overflow checking for unary negation: on a 2's-complement
555  * box, -x overflows iff x is the most negative long. In this case we
556  * get -x == x. However, -x is undefined (by C) if x /is/ the most
557  * negative long (it's a signed overflow case), and some compilers care.
558  * So we cast x to unsigned long first. However, then other compilers
559  * warn about applying unary minus to an unsigned operand. Hence the
560  * weird "0-".
561  */
562 #define UNARY_NEG_WOULD_OVERFLOW(x) \
563     ((x) < 0 && (unsigned long)(x) == 0-(unsigned long)(x))

```

python/tags/r266/Objects/intobject.c

python/tags/r266/Objects/intobject.c

```

489  /*
490  Integer overflow checking for * is painful: Python tried a couple ways, but
491  they didn't work on all platforms, or failed in endcases (a product of
492  -sys.maxint-1 has been a particular pain).
493
494  Here's another way:
495
496  The native long product x*y is either exactly right or *way* off, being
497  just the last n bits of the true product, where n is the number of bits
498  in a long (the delivered product is the true product plus i*2**n for
499  some integer i).
500
501  The native double product (double)x * (double)y is subject to three
502  rounding errors: on a sizeof(long)==8 box, each cast to double can lose
503  info, and even on a sizeof(long)==4 box, the multiplication can lose info.
504  But, unlike the native long product, it's not in *range* trouble: even
505  if sizeof(long)==32 (256-bit longs), the product easily fits in the
506  dynamic range of a double. So the leading 50 (or so) bits of the double
507  product are correct.
508
509  We check these two ways against each other, and declare victory if they're
510  approximately the same. Else, because the native long product is the only
511  one that can lose catastrophic amounts of information, it's the native long
512  product that must have overflowed.
513  */
514
515  static PyObject *
516  int_mul(PyObject *v, PyObject *w)
517  {
518      long a, b;
519      long longprod;           /* a*b in native long arithmetic */
520      double doubled_longprod; /* (double)longprod */
521      double doubleprod;       /* (double)a * (double)b */
522
523      CONVERT_TO_LONG(v, a);
524      CONVERT_TO_LONG(w, b);
525      /* casts in the next line avoid undefined behaviour on overflow */
526      longprod = (long)((unsigned long)a * b);
527      doubleprod = (double)a * (double)b;
528      doubled_longprod = (double)longprod;
529
530      /* Fast path for normal case: small multiplicands, and no info
531       is lost in either method. */
532      if (doubled_longprod == doubleprod)
533          return PyInt_FromLong(longprod);
534
535      /* Somebody somewhere lost info. Close enough, or way off? Note
536       that a != 0 and b != 0 (else doubled_longprod == doubleprod == 0).
537       The difference either is or isn't significant compared to the
538       true value (of which doubleprod is a good approximation).
539      */
540      {

```

```

541     const double diff = doubled_longprod - doubleprod;
542     const double absdiff = diff >= 0.0 ? diff : -diff;
543     const double absprod = doubleprod >= 0.0 ? doubleprod :
544                           -doubleprod;
545     /* absdiff/absprod <= 1/32 iff
546        32 * absdiff <= absprod -- 5 good bits is "close enough" */
547     if (32.0 * absdiff <= absprod)
548         return PyInt_FromLong(longprod);
549     else
550         return PyLong_Type.tp_as_number->nb_multiply(v, w);
551 }
552 }

```

python/tags/r266/Objects/intobject.c

Ruby 的代码要混乱一些，花点时间还是能找到，这是 1.8.7-p334 的实现，在 `fixdivmod()` 函数。²⁹

```

2185 static void
2186 fixdivmod(x, y, divp, modp)
2187     long x, y;
2188     long *divp, *modp;
2189 {
2190     long div, mod;
2191
2192     if (y == 0) rb_num_zerodiv();
2193     if (y < 0) {
2194         if (x < 0)
2195             div = -x / -y;
2196         else
2197             div = - (x / -y);
2198     }
2199     else {
2200         if (x < 0)
2201             div = - (-x / y);
2202         else
2203             div = x / y;
2204     }
2205     mod = x - div*y;
2206     if ((mod < 0 && y > 0) || (mod > 0 && y < 0)) {
2207         mod += y;
2208         div -= 1;
2209     }
2210     if (divp) *divp = div;
2211     if (modp) *modp = mod;
2212 }

```

ruby/tags/v1_8_7_334/numeric.c

注意到 Ruby 的 `Fixnum` 整数的表示范围比机器字长小 1bit，直接避免了溢出的可能。

²⁹http://svn.ruby-lang.org/cgi-bin/viewvc.cgi/tags/v1_8_7_334/numeric.c?view=markup

8.5 硬件实现

既然 C/C++ 以效率著称，那么应该是贴近硬件实现的。我考察了几种熟悉的硬件平台，它们基本都支持 C99/C++0x 的语意，也就是说新规定没有额外开销。列举如下。（其实我们只关系带符号除法，不过为了完整性，这里一并列出 unsigned/signed 整数除法指令。）

Intel x86/x64 Intel x86 系列的 DIV/IDIV 指令明确提到是向 0 取整，与 C99、C++0x、Java、C# 一致。

MIPS 很奇怪，我在 MIPS 的参考手册里没有查到 DIV/DIVU 指令的取整方向，不过根据 Patterson&Hennessy 的讲解，似乎向 0 取整硬件上实现起来比较容易。或许我没找对地方？

ARM/Cortex-M3 ARM 没有硬件除法指令，所以不存在这个问题。Cortex-M3 有硬件除法，SDIV/UDIV 指令都是向 0 取整。Cortex-M3 的除法指令不能同时算出余数，这很特殊。

MMIX MMIX 是 Knuth 设计的 64-bit CPU，替换原来的 MIX 机器。DIV 和 DIVU 都是向负无穷取整（依据 TAOCP 第 1.2.4 节的定义，在第一卷 40 页头几行），这是我知道的惟一支持 Python/Ruby 语义的“硬件”平台。

总结：想不到小小的整数除法都有这么大名堂。一段只涉及整数运算的代码，即便能在各种语法相似的语言里运行，结果也可能完全不同。把 C 语言里运行得好好的整数运算代码原样复制到 Python 里，也可能因为负数除法而出错。

9 用异或来交换变量是错误的

翻转一个字符串，例如把 “12345” 变成 “54321”，这是一个最简单的不过的编码任务，即便是 C 语言初学者的也能毫不费力地写出类似如下的代码：

```
// 版本一，用中间变量交换两个数，好代码
void reverse_by_swap(char* str, int n)
```



```
{
    char* begin = str;
    char* end = str + n - 1;
    while (begin < end) {
        char tmp = *begin;
        *begin = *end;
        *end = tmp;
        ++begin;
        --end;
    }
}
```

Version 1

这个代码清晰，直白，没有任何高深的技巧。

不知从什么时候开始，有人发明了不使用临时变量交换两个数的办法，用“不用临时变量交换两个数”在 [google](#) 上能搜到很多文章。下面是一个典型的实现：

```
// 版本二，用异或运算交换两个数，烂代码
void reverse_by_xor(char* str, int n)
{
    // WARNING: BAD code
    char* begin = str;
    char* end = str + n - 1;
    while (begin < end) {
        *begin ^= *end;
        *end ^= *begin;
        *begin ^= *end;
        ++begin;
        --end;
    }
}
```

Version 2

Version 2

受一些过时的教科书的误导，有人认为程序里少用一个变量，节省一个字节的空间，会让程序运行更快。这是不对的，至少在这里不成立：

1. 这个所谓的“技巧”在现代的机器上只会更慢（我甚至怀疑它从来就不可能比原始办法快）。原始办法是两次内存读和写，这个“技巧”是六读三写加三次异或（或许编译器可以优化成两读三写加三次异或）。
2. 同样也不能节省内存，因为中间变量 **tmp** 通常会是在寄存器（稍后有汇编代码供分析）。就算它在函数的局部堆栈 (**stack**) 上，反正栈已经开在那儿了，也没有进一步的函数调用，根本节约不了一丁点内存。

3. 相反，由于计算步骤较多，会使用更多的指令，编译后的机器码长度会增加。（这不是什么大问题，短的代码不一定快，后面有另外一个例子。）

这个技巧的意义完全在于应付变态的面试，所以知道就行，但绝对不能放在产品代码中。我也想不出问这样的面试题意义何在。

更有甚者，把其中三句：

```
*begin ^= *end;  
*end ^= *begin;  
*begin ^= *end;
```

写成一句：

```
*begin ^= *end ^= *begin ^= *end; // WRONG
```

这更是大有问题，会导致未定义的行为 (undefined behavior)。C 语言的一条语句中，一个变量的值只允许改变一次，像 `x = x++` 这种代码都是未定义行为。在 C 语言里没有哪条规则保证这两种写法是等价的。（致语言律师：我知道，黑话叫序列点³⁰，一个语句可能不止一个序列点，请允许我在这里使用不精确的表述。）

这不是一个值得炫耀的技巧，只会丑化劣化代码。

翻转字符串这个问题在 C++ 有更简单的解法——调用 STL 里的 `std::reverse()` 函数。有人担心调用函数会有开销，这种担心是多余的，现在的编译器会把 `std::reverse()` 这种简单函数自动内联展开，生成出来的优化汇编代码和“版本一”一样快。

```
// 版本三，用 std::reverse 颠倒一个区间，优质代码  
void reverse_by_std(char* str, int n)  
{  
    std::reverse(str, str + n);  
}
```

Version 3

Version 3

9.1 编译器会分别生成什么代码

注意：查看编译器生成的汇编代码固然是了解程序行为的一个重要手段，但是千万不要认为看到的東西是永恒真理，它只是一时一地的真相。将来换了硬件平台或编

³⁰GCC 4.x 有一个编译警告选项 `-Wsequence-point` 可以报告这种错误。
见 http://gcc.gnu.org/bugzilla/show_bug.cgi?id=39121

译器，情况可能会变化。重要的不是为什么版本一比版本二快，而是如何发现这个事实。不要“猜 guess”，要“测 benchmark”。

g++ 版本 4.4.1，编译参数 -O2 -march=core2，x86 Linux 系统。

版本一 版本一编译的汇编代码是：

```
.L3:
    movzbl    (%edx), %ecx
    movzbl    (%eax), %ebx
    movb      %bl, (%edx)
    movb      %cl, (%eax)
    incl      %edx
    decl      %eax
    cmpl      %eax, %edx
    jb        .L3
```

我用 C 语言翻译一下：

```
register char bl, cl;
register char* eax;
register char* edx;

L3:
cl = *edx; // 读
bl = *eax; // 读
*edx = bl; // 写
*eax = cl; // 写
++edx;
--eax;
if (edx < eax) goto L3;
```

一共两读两写，临时变量没有使用内存，都在寄存器里完成。考虑指令级并行和 cache 的话，中间六条语句估计能在 3、4 个周期执行完。

版本二

```
.L9:
    movzbl    (%edx), %ecx
    xorb      (%eax), %cl
    movb      %cl, (%eax)
    xorb      (%edx), %cl
    movb      %cl, (%edx)
    decl      %edx
    xorb      %cl, (%eax)
    incl      %eax
    cmpl      %edx, %eax
    jb        .L9
```

C 语言翻译:

```
// 声明与前面一样
cl = *edx;    // 读
cl ^= *eax;   // 读, 异或
*eax = cl;    // 写
cl ^= *edx;   // 读, 异或
*edx = cl;    // 写
--edx;
*eax ^= cl;   // 读、写, 异或
++eax;
if (eax < edx) goto L9;
```

一共六读三写三次异或, 多了两条指令。指令多不一定就慢, 但是这里异或版实测比临时变量版要慢许多, 因为它每条指令都用到了前面一条指令的计算结果, 没法并行执行。

版本三 , 生成的代码与版本一一样快。

```
.L21:
    movzbl    (%eax), %ecx
    movzbl    (%edx), %ebx
    movb      %bl, (%eax)
    movb      %cl, (%edx)
    incl      %eax

.L23:
    decl      %edx
    cmpl      %edx, %eax
    jb        .L21
```

这告诉我们, 不要想当然地优化, 也不要低估编译器的能力。关于现在的编译器有多聪明, 这里有一个不错的介绍³¹

Bjarne Stroustrup 说过, *I like my code to be elegant and efficient. The logic should be straightforward to make it hard for bugs to hide, the dependencies minimal to ease maintenance, error handling complete according to an articulated strategy, and performance close to optimal so as not to tempt people to make the code messy with unprincipled optimizations. Clean code does one thing well.* 中文据韩磊的翻译《代码整洁之道》³² (陈硕对文字有修改, 出错责任在我): “我喜欢优雅和高效的代码。代码逻辑应当直截了当, 叫缺陷难以隐藏; 尽量减少依赖关系, 使之便于维护; 以某种全局策略一以贯

³¹http://www.linux-kongress.org/2009/slides/compiler_survey_felix_von_leitner.pdf

³²<http://www.china-pub.com/196266>

之地处理全部出错情况；性能调校至接近最优，省得引诱别人实施无原则的优化 (unprincipled optimizations)，搞出一团乱麻。整洁的代码只做好一件事。”

这恐怕就是 Bjarne 提及的没有原则的优化，甚至根本连优化都不是。代码的清晰性是首要的。

9.2 为什么短的代码不一定快

我前一篇短文谈到负整数的除法运算（第 8 节），其中引用了一段把整数转为字符串的代码。函数反复计算一个整数除以 10 的商和余数。我原以为编译器会用一条 DIV 除法指令来算，实际生成的代码让我大吃一惊：

```
.L2:
    movl    $1717986919, %eax
    imull    %ebx
    movl    %ebx, %eax
    sarl     $31, %eax
    sarl     $2, %edx
    subl     %eax, %edx
    movl     %edx, %eax
    leal     (%edx,%edx,4), %edx
    addl     %edx, %edx
    subl     %edx, %ebx
    movl     %ebx, %edx
    movl     %eax, %ebx
    movzbl   (%edi,%edx), %eax
    movb     %al, (%esi)
    addl     $1, %esi
    testl    %ebx, %ebx
    jne      .L2
```

一条 DIV 指令被替换成了十来条指令，编译器不是傻子，必然有原因。这里我不详细解释到底是怎么算的，基本思路是把除法转换为乘法，用倒数来算。其中出现了一个魔数 1717986919，转换成 16 进制是 0x66666667，等于 $(2^{33} + 3)/5$ 。

现代处理器上乘法运算和加减法一样快，比除法快一个数量级左右，编译器生成这样的代码是有理由的。10 多年前出版的神作《程序设计实践》上介绍过如何做 micro benchmarking，方法和结果都值得一读，当然里边的数据恐怕有点过时了。

有本奇书《Hacker's Delight》，国内译作《高效程序的奥秘》³³，展示了大量这种速算技巧，第 10 章专门讲整数常量的除法。我不会把书中如天书般的技巧应用到产品代码中，但是我相信现代编译器的作者是知道这些技巧的，他们会合理地使用这

³³<http://www.china-pub.com/18801>

些技巧来提高生成代码的质量。现在已经不是那个懂点汇编就能打败编译器的时代了。

Mark C. Chu-Carroll 有一篇博客文章《The “C is Efficient” Language Fallacy》³⁴ 的观点我非常赞同：

Making real applications run really fast is something that's done with the help of a compiler. Modern architectures have reached the point where people can't code effectively in assembler anymore - switching the order of two independent instructions can have a dramatic impact on performance in a modern machine, and the constraints that you need to optimize for are just more complicated than people can generally deal with.

So for modern systems, *writing an efficient program is sort of a partnership*. The human needs to carefully choose algorithms - the machine can't possibly do that. And the machine needs to carefully compute instruction ordering, pipeline constraints, memory fetch delays, etc. The two together can build really fast systems. But the two parts aren't independent: *the human needs to express the algorithm in a way that allows the compiler to understand it well enough to be able to really optimize it*.

最后，说几句 C++ 模板。假如要编写一个任意进制的转换程序。C 语言的函数声明是：

```
bool convert(char* buf, size_t bufsize, int value, int radix);
```

既然进制是编译期常量，C++ 可以用带非类型模板参数的函数模板来实现，函数里边的代码与 C 相同。

```
template<int radix>
bool convert(char* buf, size_t bufsize, int value);
```

模板确实会使代码膨胀，但是这样的膨胀有时候是好事情，编译器能针对不同的常数生成快速算法。滥用 C++ 模板当然是错的，适当使用不会有问题。

³⁴http://scienceblogs.com/goodmath/2006/11/the_c_is_efficient_language_fa.php

10 在单元测试中 mock 系统调用

陈硕在《分布式程序的自动化回归测试》³⁵一文中曾经谈到单元测试在分布式程序开发中的优缺点（好吧，主要是缺点）。但是，在某些情况下，单元测试是很有必要的，在测试 **failure** 场景的时候尤显重要，比如：

- 在开发存储系统时，模拟 `read(2)/write(2)` 返回 `EIO` 错误（有可能是磁盘写满了，有可能是磁盘出坏道读不出数据）。
- 在开发网络库的时候，模拟 `write(2)` 返回 `EPIPE` 错误（对方意外断开连接）。
- 在开发网络库的时候，模拟自连接 (`self-connection`)，网络库应该用 `getsockname(2)` 和 `getpeername(2)` 判断是否是自连接，然后断开之。
- 在开发网络库的时候，模拟本地 `ephemeral port` 耗尽，`connect(2)` 返回 `EAGAIN` 临时错误。
- 让 `gethostbyname(2)` 返回我们预设的值，防止单元测试给公司的 `DNS server` 带来太大压力。

这些 **test case** 恐怕很难用前文提到的 **test harness** 来测试，该单元测试上场了。现在的问题是，如何 **mock** 这些系统函数？或者换句话说，如何把对系统函数的依赖注入到被测程序中？

10.1 系统函数的依赖注入

在《修改代码的艺术》^[7]一书第 4.3.2 节中，作者介绍了链接期接缝 (`link seam`)，正好可以解决我们的问题。另外，在 `Stack Overflow` 的一个帖子³⁶里也总结了几种做法。

如果程序（库）在编写的时候就考虑了可测试性，那么用不到上面的 **hack** 手段，我们可以从设计上解决依赖注入的问题。这里提供两个思路。

³⁵<http://blog.csdn.net/Solstice/archive/2011/04/25/6359748.aspx>

³⁶<http://stackoverflow.com/questions/2924440/advice-on-mocking-system-calls>

其一 采用传统的面向对象的手法，借助运行期的迟绑定实现注入与替换。自己写一个 `System interface`，把程序里用到的 `open`、`close`、`read`、`write`、`connect`、`bind`、`listen`、`accept`、`gethostname`、`getpeername`、`getsockname` 等等函数统用虚函数封装一层。然后在代码里不要直接调用 `open()`，而是调用 `System::instance().open()`。这样代码主动把控制权交给了 `System interface`，我们可以在这里动动手脚。在写单元测试的时候，把这个 `singleton instance` 替换为我们的 `mock object`，这样就能模拟各种 `error code`。

其二 采用编译期或链接期的迟绑定。注意到在第一种做法中，运行期多态是不必要的，因为程序从生到死只会用到一个 `implementation object`。为此付出虚函数调用的代价似乎有些不值。（其实，跟系统调用比起来，虚函数这点开销可忽略不计。）

我们可以写一个 `system namespace` 头文件，在其中声明 `read()` 和 `write()` 等普通函数，然后在 `.cc` 文件里转发给对应系统的系统函数 `::read()` 和 `::write()` 等。

```

----- muduo/net/SocketsOps.h
namespace sockets
{
    int connect(int sockfd, const struct sockaddr_in& addr);
}
----- muduo/net/SocketsOps.h

----- muduo/net/SocketsOps.cc
int sockets::connect(int sockfd, const struct sockaddr_in& addr)
{
    return ::connect(sockfd, sockaddr_cast(&addr), sizeof addr);
}
----- muduo/net/SocketsOps.cc
```

有了这么一层间接性，就可以在编写单元测试的时候动动手脚，链接我们的 `stub` 实现，以达到替换实现的目的：

```

----- MockSocketsOps.cc
int sockets::connect(int sockfd, const struct sockaddr_in& addr)
{
    errno = EAGAIN;
    return -1;
}
----- MockSocketsOps.cc
```

C++ 一个程序只能有一个 `main()` 入口，所以要先把程序做成 `library`，再用单元测试代码链接这个 `library`。假设有一个 `mynetcat` 程序，为了编写 C++ 单元测试，我们把它拆成两部分，`library` 和 `main()`，源文件分别是 `mynetcat.cc` 和 `main.cc`。

在编译普通程序的时候：

```
g++ main.cc mynetcat.cc SocketsOps.cc -o mynetcat
```

在编译单元测试时这么写：

```
g++ test.cc mynetcat.cc MockSocketsOps.cc -o test
```

以上是最简单的例子，在实际开发中可以让 **stub** 功能更强大一些，比如根据不同的 **test case** 返回不同的错误。

第二种做法无需用到虚函数，代码写起来也比较简洁，只用前缀 **sockets::** 即可。例如应用程序的代码里写 **sockets::connect(fd, addr)**。

muduo 目前还没有单元测试，只是预留了这些 **stubs**。

namespace 的好处在于它不是封闭的，我们可以随时打开往里添加新的函数，而不用改动原来的头文件。这也是以 **non-member non-friend** 函数为接口的优点。

以上两种做法还有一个好处，即只 **mock** 我们关心的部分代码。如果程序用到了 **SQLite** 或 **Berkeley DB** 这些会访问本地文件系统的第三方库，那么我们的 **System interface** 或 **system namespace** 不会拦截这些第三方库的 **open(2)**、**close(2)**、**read(2)**、**write(2)** 等系统调用。

10.2 链接期垫片 (link seam)

如果程序在一开始编码的时候没有考虑单元测试，那么又该如何注入 **mock** 系统调用呢？

上面第二种做法已经给出了答案，那就是使用 **link seam**（链接期垫片）。

比方说要仿冒 **connect(2)** 函数，那么我们在单元测试程序里实现一个自己的 **connect()** 函数，它遮盖了同名的系统函数。在链接的时候，**linker** 会优先采用我们自己定义的函数。（这对动态链接是成立的，如果是静态链接，会报 **multiple definition** 错误。好在绝大多数情况下 **libc** 是动态链接的。）

```
typedef int (*connect_func_t)(int sockfd,
                             const struct sockaddr *addr,
                             socklen_t addrlen);

connect_func_t connect_func = dlsym(RTDL_NEXT, "connect");

bool mock_connect;
```

```

int mock_connect_errno;

// mock connect
extern "C" int connect(int sockfd,
                      const struct sockaddr *addr,
                      socklen_t addrlen)
{
    if (mock_connect) {
        errno = mock_connect_errno;
        return errno == 0 ? 0 : -1;
    } else {
        return connect_func(sockfd, addr, addrlen);
    }
}

```

mock connect(2)

如果程序真的要调用 `connect(2)` 怎么办？在我们自己的 `mock connect(2)` 里不能再调用 `connect()` 了，否则会出现无限递归。为了防止这种情况，我们用 `dlsym(RTDL_NEXT, "connect")` 获得 `connect(2)` 系统函数的真实地址，然后通过函数指针 `connect_func` 来调用它。

10.3 例子：ZooKeeper 的 C client library

ZooKeeper 的 C client library 正是采用了 link seams 来编写单元测试，代码见：

<http://svn.apache.org/repos/asf/zookeeper/tags/release-3.3.3/src/c/tests/LibCMocks.h>

<http://svn.apache.org/repos/asf/zookeeper/tags/release-3.3.3/src/c/tests/LibCMocks.cc>

10.4 其他做法

Stack Overflow 的帖子里还提到一个做法，可以方便地替换动态库里的函数，即使用 `ld` 的 `-wrap` 参数，文档里说得很清楚，这里不再赘述。

`--wrap=symbol` *man ld(1)*

Use a wrapper function for symbol. Any undefined reference to symbol will be resolved to `__wrap_symbol`. Any undefined reference to `__real_symbol` will be resolved to symbol.

This can be used to provide a wrapper for a system function. The wrapper function should be called `__wrap_symbol`. If it wishes to call the system function, it should call `__real_symbol`.

Here is a trivial example:

```
void *
__wrap_malloc (size_t c)
{
    printf ("malloc called with %zu\n", c);
    return __real_malloc (c);
}
```

If you link other code with this file using `--wrap malloc`, then all calls to "malloc" will call the function "`__wrap_malloc`" instead. The call to "`__real_malloc`" in "`__wrap_malloc`" will call the real "malloc" function.

You may wish to provide a "`__real_malloc`" function as well, so that links without the `--wrap` option will succeed. If you do this, you should not put the definition of "`__real_malloc`" in the same file as "`__wrap_malloc`"; if you do, the assembler may resolve the call before the linker has a chance to wrap it to "malloc".

man ld(1)

10.5 第三方 C++ 库

Link seam 同样适用于第三方 C++ 库

比方说公司某个基础库团队提供了 `File` class，但是这个 class 没有使用虚函数，我们无法通过 `sub-classing` 的办法来实现 mock object。

```
class File : boost::noncopyable
{
public:
    File(const char* filename);
    ~File();

    int readn(void* data, int len);
    int writen(const void* data, int len);
    size_t getSize() const;
private:
};
```

File.h

File.h

如果需要为用到 `File` class 的程序编写单元测试，那么我们可以自己定义其成员函数的实现，这样可以注入任何我们想要的结果。

```
int File::readn(void* data, int len)
{
```

MockFile.cc

```
    return -1;  
}
```

MockFile.cc

(这个做法对动态库是可行的，静态库会报错。我们要么让对方提供专供单元测试的动态库，要么拿过源码来自己编译一个。)

Java 也有类似的做法，在 `class path` 里替换我们自己的 `stub jar` 文件，以实现 `link seam`。不过 Java 有动态代理，很少用得着 `link seam` 来实现依赖注入。

11 iostream 的用途与局限

本文主要考虑 x86 Linux 平台，不考虑跨平台的可移植性，也不考虑国际化 (i18n)，但是要考虑 32-bit 和 64-bit 的兼容性。本文以 `stdio` 指代 C 语言的 `scanf/printf` 系列格式化输入输出函数。本文提及“C 语言”（包括库函数和线程安全性），指的是 Linux 下 `gcc + glibc` 这一套编译器和库的具体实现，也可以认为是符合 POSIX.1-2001 的实现。本文注意区分“编程初学者”和“C++ 初学者”，二者含义不同。

C++ `iostream` 的主要作用是让初学者有一个方便的命令行输入输出试验环境，在真实的项目中很少用到 `iostream`，因此不必把精力花在深究 `iostream` 的格式化与 `manipulator`（格式操控符）。`iostream` 的设计初衷是提供一个可扩展的类型安全的 IO 机制，但是后来莫名其妙地加入了 `locale` 和 `facet` 等累赘。其整个设计复杂不堪，多重 + 虚拟继承的结构也很巴洛克，性能方面几无亮点。`iostream` 在实际项目中的用处非常有限，为此投入过多学习精力实在不值。

11.1 stdio 格式化输入输出的缺点

11.1.1 对编程初学者不友好

看看下面这段简单的输入输出代码，这是 C 语言教学的基本示例。

```
#include <stdio.h>  
  
int main()  
{  
    int i;  
    short s;
```

```
float f;  
double d;  
char name[80];  
  
scanf("%d %hd %f %lf %s", &i, &s, &f, &d, name);  
printf("%d %d %f %f %s\n", i, s, f, d, name);  
}
```

注意到其中

- 输入和输出用的格式字符串不一样。输入 **short** 要用 `%hd`，输出用 `%d`；输入 **double** 要用 `%lf`，输出用 `%f`。
- 输入的参数不统一。对于 `i`、`s`、`f`、`d` 等变量，在传入 `scanf()` 的时候要取地址 (`&`)，而对于字符数组 `name`，则不用取地址。

读者可以试一试如何用几句话向刚开始学编程的初学者解释上面两条背后原因 (涉及到传递函数不定参数时的类型转换，函数调用栈的内存布局，指针的意义，字符数组退化为字符指针等等)，如果一开始解释不清，只好告诉学生“这是规定”。

- 缓冲区溢出的危险。上面的例子在读入 `name` 的时候没有指定大小，这是用 C 语言编程的安全漏洞的主要来源。应该在一开始就强调正确的做法，避免养成错误的习惯。

正确而安全的做法如 Bjarne Stroustrup 在《Learning Standard C++ as a New Language》所示：

```
int main()  
{  
    const int max_name = 80;  
    char name[max_name];  
  
    char fmt[10];  
    sprintf(fmt, "%%%ds", max_name - 1);  
    scanf(fmt, name);  
    printf("%s\n", name);  
}
```

这个动态构造格式化字符串的做法恐怕更难向初学者解释。

11.1.2 安全性 (security)

C 语言的安全性问题近十几年来引起了广泛的注意，C99 增加了 `snprintf()` 等能够指定输出缓冲区大小的函数，输出方面的安全性问题已经得到解决；输入方面似乎没有太大进展，还要靠程序员自己动手。

考虑一个简单的编程任务：从文件或标准输入读入一行字符串，行的长度不确定。我发现竟然没有哪个 C 语言标准库函数能完成这个任务，除非自己动手 (`roll your own`)。

首先，`gets()` 是错误的，因为不能指定缓冲区的长度。

其次，`fgets()` 也有问题。它能指定缓冲区的长度，所以是安全的。但是程序必须预设一个长度的最大值，这不满足题目要求“行的长度不确定”。另外，程序无法判断 `fgets()` 到底读了多少个字节。为什么？考虑一个文件的内容是 9 个字节的字符串 `"Chen\000Shuo"`，注意中间出现了 `'\0'` 字符，如果用 `fgets()` 来读取，客户端如何知道 `"\000Shuo"` 也是输入的一部分？毕竟 `strlen()` 只返回 4，而且整个字符串里没有 `'\n'` 字符。

最后，可以用 `glibc` 定义的 `getline(3)` 函数来读取不定长的“行”。这个函数能正确处理各种情况，不过它返回的是 `malloc()` 分配的内存，要求调用端自己 `free()`。

11.1.3 类型安全 (type-safety)

如果 `printf()` 的整数参数类型是 `int`、`long` 等内置类型，那么 `printf()` 的格式化字符串很容易写。但是如果参数类型是系统头文件里 `typedef` 的类型呢？

如果你想在程序中用 `printf()` 来打印日志，你能一眼看出下面这些类型该用 `"%d"` `"%ld"` `"%lld"` 中的哪一个来输出？你的选择是否同时兼容 32-bit 和 64-bit 平台？

- `clock_t`。这是 `clock(3)` 的返回类型
- `dev_t`。这是 `mknod(3)` 的参数类型
- `in_addr_t`、`in_port_t`。这是 `struct sockaddr_in` 的成员类型
- `nfds_t`。这是 `poll(2)` 的参数类型
- `off_t`。这是 `lseek(2)` 的参数类型，麻烦的是，这个类型与宏定义 `_FILE_OFFSET_BITS` 有关。
- `pid_t`、`uid_t`、`gid_t`。这是 `getpid(2)` `getuid(2)` `getgid(2)` 的返回类型

- `ptrdiff_t`。 `printf()` 专门定义了 `"t"` 前缀来支持这一类型（即使用 `"%td"` 来打印）。
- `size_t`、`ssize_t`。这两个类型到处都在用。 `printf()` 为此专门定义了 `"z"` 前缀来支持这两个类型（即使用 `"%zu"` 或 `"%zd"` 来打印）。
- `socklen_t`。这是 `bind(2)` 和 `connect(2)` 的参数类型
- `time_t`。这是 `time(2)` 的返回类型，也是 `gettimeofday(2)` 和 `clock_gettime(2)` 的结构体参数的成员类型

如果在 C 程序里要正确打印以上类型的整数，恐怕要费一番脑筋。《The Linux Programming Interface》的作者建议（3.6.2 节）先统一转换为 `long` 类型再用 `"%ld"` 来打印；对于某些类型仍然需要特殊处理，比如 `off_t` 的类型可能是 `long long`。

还有，`int64_t` 在 32-bit 和 64-bit 平台上是不同的类型，为此，如果程序要打印 `int64_t` 变量，需要包含 `<inttypes.h>` 头文件，并且使用 `PRId64` 宏：

```
#include <stdio.h>
#define __STDC_FORMAT_MACROS
#include <inttypes.h>

int main()
{
    int64_t x = 100;
    printf("%" PRId64 "\n", x);
    printf("%06" PRId64 "\n", x);
}
```

`muduo` 的 `Timestamp` 使用了 `PRId64`。³⁷ Google C++ 编码规范也提到了 64-bit 兼容性。³⁸

这些问题在 C++ 里都不存在，在这方面 `iostream` 是个进步。

C `stdio` 在类型安全方面原本还有一个缺点，即格式化字符串与参数类型不匹配会造成难以发现的 `bug`，不过现在的编译器已经能够检测很多这种错误（使用 `-Wall` 编译选项）：

```
int main()
{
    double d = 100.0;
    // warning: format '%d' expects type 'int', but argument 2 has type 'double'
```

³⁷<http://code.google.com/p/muduo/source/browse/trunk/muduo/base/Timestamp.cc#25>

³⁸http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml#64-bit_Portability

```
printf("%d\n", d);

short s;
// warning: format '%d' expects type 'int*', but argument 2 has type 'short int*'
scanf("%d", &s);

size_t sz = 1;
// no warning
printf("%zd\n", sz);
}
```

11.1.4 不可扩展?

C stdio 的另外一个缺点是无法支持自定义的类型, 比如我写了一个 `Date class`, 我无法像打印 `int` 那样用 `printf()` 来直接打印 `Date` 对象。

```
struct Date
{
    int year, month, day;
};

Date date;
printf("%D", &date); // WRONG
```

Glibc 放宽了这个限制, 允许用户调用 `register_printf_function(3)` 注册自己的类型。当然, 前提是与现有的格式字符不冲突 (这其实大大限制了这个功能的用处, 现实中也几乎没有人真的去用它)。^{39 40}

11.1.5 性能

C stdio 的性能方面有两个弱点。

1. 使用一种 **little language** (现在流行叫 DSL) 来配置格式。固然有利于紧凑性和灵活性, 但损失了一点点效率。每次打印一个整数都要先解析 `"%d"` 字符串, 大多数情况下不是问题, 某些场合需要自己写整数到字符串的转换。
2. C locale 的负担。locale 指的是不同语种对“什么是空白”、“什么是字母”, “什么是小数点”有不同的定义 (德语里边小数点是逗号, 不是句点)。C 语言的 `printf()`、`scanf()`、`isspace()`、`isalpha()`、`ispunct()`、`strtod()` 等等函数都和 locale 有关, 而且可以在运行时动态更改 locale。就算是程序只使用默认的“C” locale, 仍然要为此付出代价。

³⁹<http://www.gnu.org/s/hello/manual/libc/Printf-Extension-Example.html>

⁴⁰http://en.wikipedia.org/wiki/Printf#Custom_format_placeholders

11.2 iostream 的设计初衷

iostream 的设计初衷包括克服 C stdio 的缺点，提供一个高效的可扩展的类型安全的 IO 机制。“可扩展”有两层意思，一是可以扩展到用户自定义类型，二是通过继承 iostream 来定义自己的 stream，本文把前一种称为“类型可扩展”后一种称为“功能可扩展”。

类型可扩展和类型安全

“类型可扩展”和“类型安全”都是通过函数重载来实现的。

iostream 对初学者很友好，用 iostream 重写与前面同样功能的代码：

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
    int i;
    short s;
    float f;
    double d;
    string name;

    cin >> i >> s >> f >> d >> name;
    cout << i << " " << s << " " << f << " " << d << " " << name << endl;
}
```

这段代码恐怕比 scanf/printf 版本容易解释得多，而且没有安全性 (security) 方面的问题。

我们自己的类型也可以融入 iostream，使用起来与 built-in 类型没有区别。这主要得力于 C++ 可以定义 non-member functions/operators。

```
#include <ostream> // 是不是太重量级了？

class Date
{
public:
    Date(int year, int month, int day)
        : year_(year), month_(month), day_(day)
    {
    }
}
```

```
void writeTo(std::ostream& os) const
{
    os << year_ << '-' << month_ << '-' << day_;
}

private:
    int year_, month_, day_;
};

std::ostream& operator<<(std::ostream& os, const Date& date)
{
    date.writeTo(os);
    return os;
}

int main()
{
    Date date(2011, 4, 3);
    std::cout << date << std::endl;
}
```

iostream 凭借这两点（类型安全和类型可扩展），基本克服了 stdio 在使用上的不便与不安全。如果 iostream 止步于此，那它将是一个非常便利的库，可惜它前进了另外一步。

iostream 的演变大致可分为三个阶段，第一阶段是 Bjarne Stroustrup 在 CFront 1.0 里实现的 streams 库。⁴¹ 这个库符合前述“类型安全、可扩展、高效”等特征，Bjarne 发明了用移位操作符（<< 和 >>）做 I/O 的办法，istream 和 ostream 都是具体类，也没有 manipulator。第二阶段，Jerry Schwarz 设计了“经典”iostream，在 CFront 2.0 中他的设计大部分得以体现。他发明了 manipulator，实现手法是以函数指针参数来重载输入输出操作符；他还采用多重继承和虚拟继承手法，设计了现在我们看到的 ios 菱形继承体系；此外，istream 有了基类 ios，也有了派生类 ifstream 和 istrstream，ostream 也是如此。第三阶段，在 C++ 标准化的过程中，iostream 有大幅更新，Nathan Myers 设计了 Locale/Facet 体系，iostream 被模板化以适应宽窄两种字符，以及以 stringstreams 替换 strstreams 等等。

⁴¹http://www.softwarepreservation.org/projects/c_plus_plus/cfront/release-1.0/src/cfront/incl/stream.h/view

11.3 iostream 与标准库其他组件的交互

11.3.1 “值语义”与“对象语义”

不同于标准库其他 class 的“值语义/value semantics”，`iostream` 是“对象语义/object semantics”⁴²，即 `iostream` 是 non-copyable。这是正确的，因为如果 `fstream` 代表一个打开的文件的话，拷贝一个 `fstream` 对象意味着什么呢？表示打开了两个文件吗？如果销毁一个 `fstream` 对象，它会关闭文件句柄，那么另一个 `fstream` 对象副本会因此受影响吗？

`iostream` 禁止拷贝，利用对象的生命期来明确管理资源（如文件），很自然地就避免了这些问题。这就是 RAII，一种重要且独特的 C++ 编程手法。

C++ 同时支持“数据抽象/data abstraction”和“面向对象编程/object-oriented”，其实主要就是“值语义”与“对象语义”的区别，这是一个比较大的话题，见12节。

11.3.2 std::string

`iostream` 可以与 `std::string` 配合得很好。但是有一个问题：谁依赖谁？

`std::string` 的 `operator<<` 和 `operator>>` 是如何声明的？注意 `operator<<` 是个二元操作符，它的参数是 `std::ostream` 和 `std::string`。`<string>` 头文件在声明这两个 operators 的时候要不要 `#include <iostream>`？

`iostream` 和 `std::string` 都可以单独 include 来使用，显然 `iostream` 头文件里不会定义 `std::string` 的 `<<` 和 `>>` 操作。但是，如果 `<string>` 要 `#include <iostream>`，岂不是让 `string` 的用户被迫也用了 `iostream`？编译 `iostream` 头文件可是相当的慢啊（因为 `iostream` 是 template，其实现代码都放到了头文件中）。

标准库的解决办法是定义 `<iosfwd>` 头文件，其中包含 `istream` 和 `ostream` 等的前向声明 (forward declarations)，这样 `<string>` 头文件在定义输入输出操作符时就可以不必包含 `<iostream>`，只需要包含简短得多的 `<iosfwd>`，避免引入不必要的依赖。我们自己写程序也可借此学习如何支持可选的功能。

值得注意的是，`istream::getline()` 成员函数的参数类型是 `char*`，因为 `<istream>` 没有包含 `<string>`，而我们常用的 `std::getline()` 函数是个 non-member function，定义在 `<string>` 里边。

⁴²对象语义在其他面向对象的语言里通常叫做“引用语义/reference semantics”，为了避免与 C++ 的“引用”类型冲突，我这里用“对象语义”这个术语。

11.3.3 std::complex

标准库的复数类 `std::complex` 的情况比较复杂。`<complex>` 头文件会自动包含 `<sstream>`，后者会包含 `<istream>` 和 `<ostream>`，这是个不小的负担。问题是，为什么？

它的 `operator>>` 操作比 `string` 复杂得多，如何应对格式不正确的情况？输入字符串不会遇到格式不正确，但是输入一个复数可能遇到各种问题，比如数字的格式不对等。我怀疑有谁会真的在产品项目里用 `operator>>` 来读入字符方式表示的复数，这样的代码的健壮性如何保证？基于同样的理由，我认为产品代码中应该避免用 `istream` 来读取带格式的内容，后面也不再谈 `istream` 格式化输入的缺点，它已经被秒杀。

它的 `operator<<` 也很奇怪，它不是直接使用参数 `ostream& os` 对象来输出，而是先构造 `ostringstream`，输出到该 `string stream`，再把结果字符串输出到 `ostream`。简化后的代码如下：

```
template<typename T>
std::ostream& operator<<(std::ostream& os, const std::complex<T>& x)
{
    std::ostringstream s;
    s << '(' << x.real() << ',' << x.imag() << ')';
    return os << s.str();
}
```

注意到 `ostringstream` 会用到动态分配内存，也就是说，每输出一个 `complex` 对象就会分配释放一次内存，效率堪忧。

根据以上分析，我认为 `iostream` 和 `complex` 配合得不好，但是它们耦合得更紧密（与 `string/iostream` 相比），这可能是个不得已的技术限制吧（`complex` 是 `class template`，其 `operator<<` 必须在头文件中定义，而这个定义又用到了 `ostringstream`，不得已包含了 `sstream` 的实现）。

如果程序要对 `complex` 做 IO，从效率和健壮性方面考虑，建议不要使用 `iostream`。

11.4 iostream 在使用方面的缺点

在简单使用 `iostream` 的时候，它确实比 `stdio` 方便，但是深入一点就会发现，二者可说各擅胜场。下面谈一谈 `iostream` 在使用方面的缺点。

11.4.1 格式化输出很繁琐

iostream 采用 manipulator 来格式化，如果我想按照 2010-04-03 的格式输出前面定义的 Date class，那么代码要改成：

```
class Date
{
    // ...

    void writeTo(std::ostream& os) const
    {
-   os << year_ << '-' << month_ << '-' << day_;
+   os << year_ << '-'
+   << std::setw(2) << std::setfill('0') << month_ << '-'
+   << std::setw(2) << std::setfill('0') << day_;
    }

    // ...
}
```

假如用 stdio，会简短得多，因为 printf 采用了一种表达能力较强的小语言来描述输出格式。

```
class Date
{
    // ...

    void writeTo(std::ostream& os) const
    {
-   os << year_ << '-' << month_ << '-' << day_;
+   char buf[32];
+   snprintf(buf, sizeof buf, "%d-%02d-%02d", year_, month_, day_);
+   os << buf;
    }

    // ...
}
```

使用小语言来描述格式还带来另外一个好处：外部可配置。

11.4.2 外部可配置性

能不能用外部的配置文件来定义程序中日期的格式？C stdio 很好办，把格式字符串 "%d-%02d-%02d" 保存到配置里就行。但是 iostream 呢？它的格式是写在代码里的，灵活性大打折扣。

再举一个例子，程序的 message 的多语言化。

```
const char* name = "Shuo Chen";
int age = 29;
printf("My name is %1$s, I am %2$d years old.\n", name, age);
cout << "My name is " << name << ", I am " << age << " years old." << endl;
```

对于 `stdio`，要让这段程序支持中文的话，把代码中的“`My name is ...`”，替换为“`我叫%1$s，今年%2$d 岁。`”即可。也可以把这段提示语做成资源文件，在运行时读入。而对于 `iostream`，恐怕没有这么方便，因为代码是支离破碎的。

C `stdio` 的格式化字符串体现了重要的“数据就是代码”的思想，这种“数据”与“代码”之间的相互转换是程序灵活性的根源，远比 OO 更为灵活。

11.4.3 stream 的状态

如果我想用 16 进制方式输出一个整数 `x`，那么可以用 `hex` 操控符，但是这会改变 `ostream` 的状态。比如说

```
int x = 8888;
cout << hex << showbase << x << endl; // print 0x22b8
cout << 123 << endl; // print 0x7b
```

这段代码会把 123 也按照 16 进制方式输出，这恐怕不是我们想要的。

再举一个例子，`setprecision()` 也会造成持续影响：

```
double d = 123.45;
printf("%.3f\n", d);
cout << d << endl;
cout << setw(8) << fixed << setprecision(3) << d << endl;
cout << d << endl;
```

输出是：

```
$ ./a.out
123.450    %8.3f 的输出
123.45     默认 cout 格式
123.450    我们设置的精度
123.450    精度持续影响后续输出
```

可见代码中的 `setprecision()` 影响了后续输出的精度。注意 `setw()` 不会造成影响，它只对下一个输出有效。

这说明，如果使用 `manipulator` 来控制格式，需要时刻小心防止影响了后续代码。而使用 C `stdio` 就没有这个问题，它是“上下文无关的”。

11.4.4 知识的通用性

在 C 语言之外，有其他很多语言也支持 `printf()` 风格的格式化，例如 Java、Perl、Ruby 等等⁴³。学会 `printf()` 的格式化方法，这个知识还可以用到其他语言中。但是 C++ `iostream` 只此一家别无分店，反正都是格式化输出，学习 `stdio` 的投资回报率更高。

基于这点考虑，我认为不必深究 `iostream` 的格式化方法，只需要用好它最基本的类型安全输出即可。在真的需要格式化的场合，可以考虑 `snprintf()` 打印到栈上缓冲，再用 `ostream` 输出。

11.4.5 线程安全与原子性

`iostream` 的另外一个问题是线程安全性。POSIX.1-2001 明确要求 `stdio` 函数是线程安全的，⁴⁴ 而且还提供了 `flockfile(3)/funlockfile(3)` 之类的函数来明确控制 `FILE*` 的加锁与解锁。

`iostream` 在线程安全方面没有保证，就算单个 `operator<<` 是线程安全的，也不能保证原子性。因为 `cout << a << b;` 是两次函数调用，相当于 `cout.operator<<(a).operator<<(b)`。两次调用中间可能会被打断进行上下文切换，造成输出内容不连续，插入了其他线程打印的字符。

而 `fprintf(stdout, "%s %d", a, b);` 是一次函数调用，而且是线程安全的，打印的内容不会受其他线程影响。

因此，`iostream` 并不适合在多线程程序中做 logging。

11.4.6 iostream 的局限

根据以上分析，我们可以归纳 `iostream` 的局限：

- 输入方面，`istream` 不适合输入带格式的数据，因为“纠错”能力不强，进一步的分析请见孟岩写的《契约思想的一个反面案例》，孟岩说“复杂的设计必然带来复杂的使用规则，而面对复杂的使用规则，用户是可以投票的，那就是：你做你的，我不用！”可谓鞭辟入里。如果要用 `istream`，我推荐的做法是用 `std::getline()` 读入一行数据到 `std::string`，然后用正则表达式来判断内

⁴³http://en.wikipedia.org/wiki/Printf#Programming_languages_with_printf

⁴⁴<http://www.kernel.org/doc/man-pages/online/pages/man7/pthreads.7.html>

容正误，并做分组，最后用 `strtod()/strtol()` 之类的函数做类型转换。这样似乎更容易写出健壮的程序。

- 输出方面，`ostream` 的格式化输出非常繁琐，而且写死在代码里，不如 `stdio` 的小语言那么灵活通用。建议只用作简单的无格式输出。
- `log` 方面，由于 `ostream` 没有办法在多线程程序中保证一行输出的完整性，建议不要直接用它来写 `log`。如果是简单的单线程程序，输出数据量较少的情况下可以酌情使用。当然，产品代码应该用成熟的 `logging` 库，而不要用其它东西来凑合。
- `in-memory` 格式化方面，由于 `ostringstream` 会动态分配内存，它不适合性能要求较高的场合。
- 文件 IO 方面，如果用作文本文件的输入或输出，`fstreams` 有上述的缺点；如果用作二进制数据输入输出，那么自己简单封装一个 `File class` 似乎更好用，也不必为用不到的功能付出代价（后文还有具体例子）。`ifstream` 的一个用处是在程序启动时读入简单的文本配置文件。如果配置文件是其他文本格式（XML 或 JSON），那么用相应的库来读，也用不到 `ifstream`。
- 性能方面，`iostream` 没有兑现“高效性”诺言。`iostream` 在某些场合比 `stdio` 快，在某些场合比 `stdio` 慢，对于性能要求较高的场合，我们应该自己实现字符串转换（见后文的代码与测试）。`iostream` 性能方面的一个注脚：在线 ACM/ICPC 判题网站上，如果一个简单的题目发生超时错误，那么把其中 `iostream` 的输入输出换成 `stdio`，有时就能过关⁴⁵。

既然有这么多局限，`iostream` 在实际项目中的应用就大为受限了，在这上面投入太多的精力实在不值得。说实话，我没有见过哪个 C++ 产品代码使用 `iostream` 来作为输入输出设施。Google 的 C++ 编程规范也对 `stream` 的使用做了明确的限制。⁴⁶

11.5 `iostream` 在设计方面的缺点

`iostream` 的设计有相当多的 WTFs，`stackoverflow` 有人吐槽说 “If you had to judge by today’s software engineering standards, would C++’s `IOStreams` still be considered well-designed?”⁴⁷。

⁴⁵另外可以先试试调用 `cin.sync_with_stdio(false);`，见 <http://stackoverflow.com/questions/9371238>

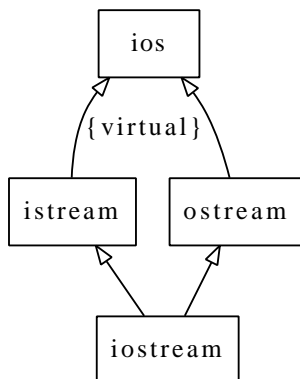
⁴⁶<http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml#Streams>

⁴⁷<http://stackoverflow.com/questions/2753060/who-architected-designed-cs-iostreams-and-would-it-still-be-considered-well>

11.5.1 面向对象的设计

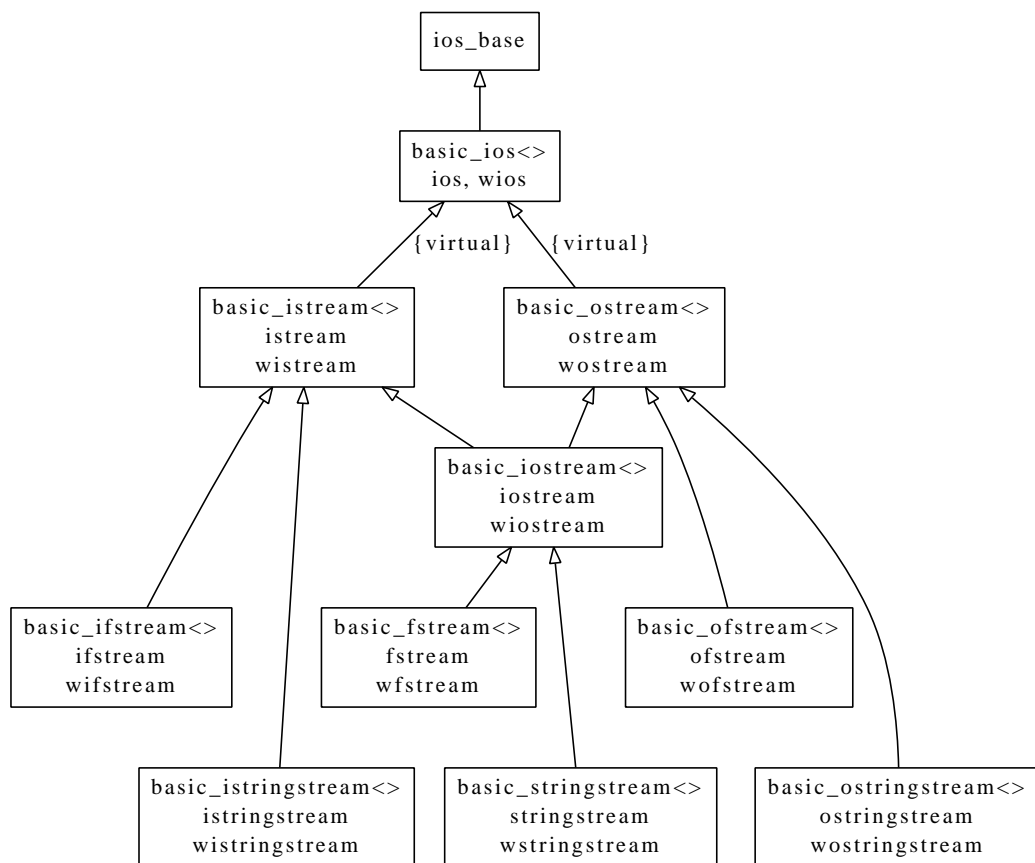
`iostream` 是个面向对象的 IO 类库，本节简单介绍它的继承体系。

对 `iostream` 略有了解的人会知道它用了多重继承和虚拟继承，简单地画个类图如下，是典型的菱形继承：

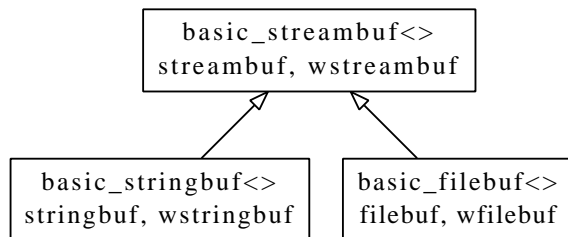


如果加深一点了解，会发现 `iostream` 现在是模板化的，同时支持窄字符和宽字符。下图是现在的继承体系，同时画出了 `fstreams` 和 `stringstreams`。图中方框的第二三行是模板的具现化类型，也就是我们代码里常用的具体类型（通过 `typedef` 定义）。

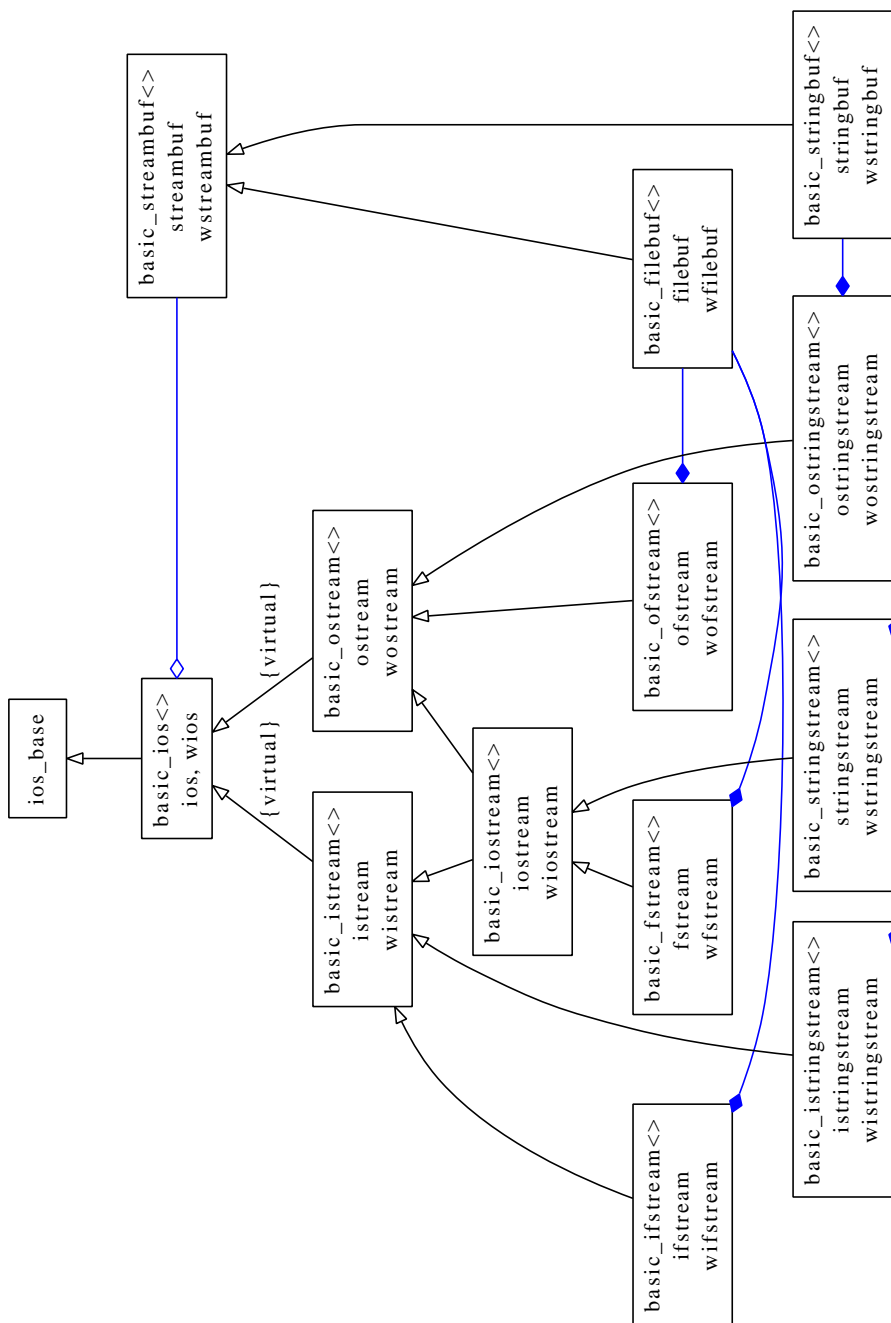
这个继承体系糅合了面向对象与泛型编程，但可惜它两方面都不讨好。



再进一步加深了解，发现还有一个平行的 `streambuf` 继承体系，`fstream` 和 `stringstream` 的不同之处主要就在于它们使用了不同的 `streambuf` 派生类型。



再把这两个继承体系画到一幅图里：



注意到 `basic_ios` 持有了 `streambuf` 的指针；而 `fstreams` 和 `stringstreams` 则

分别包含 `filebuf` 和 `stringbuf` 的对象。看上去有点像 Bridge 模式。

看了这样巴洛克的设计，有没有人还打算在自己的项目中想通过继承 `iostream` 来实现自己的 `stream`，以实现功能扩展么？

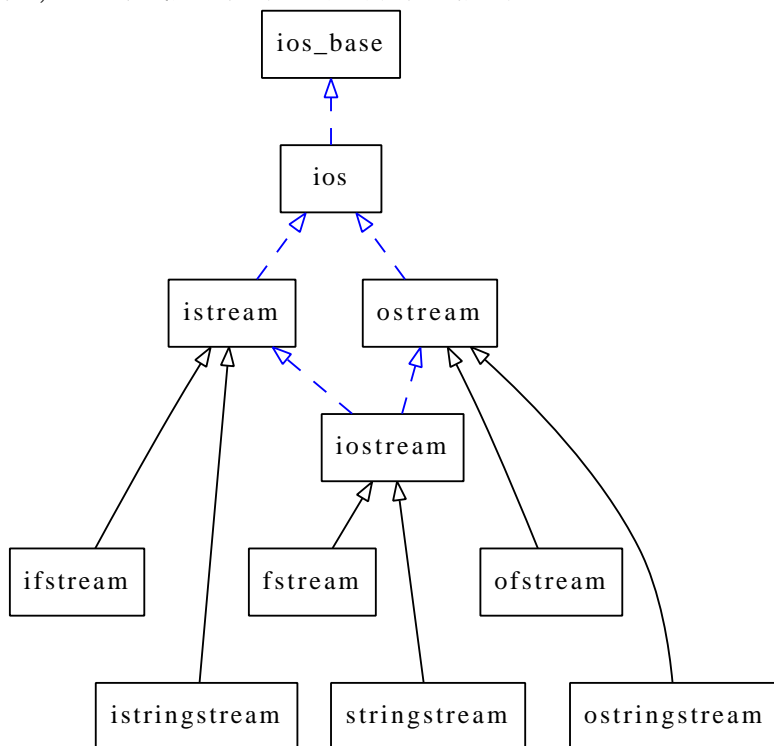
11.5.2 面向对象方面的设计缺陷

本节我们分析一下 `iostream` 的设计违反了哪些 OO 准则。

我们知道，面向对象中的 `public` 继承需要满足 Liskov 替换原则。（见《Effective C++ 第3版》[2, item 32]：确保你的 `public` 继承模塑出 is-a 关系。《C++ 编程规范》[4, item 37]：`public` 继承意味可替换性。继承非为复用，乃为被复用。）

在程序里需要用到 `ostream` 的地方（例如 `operator<<`），我传入 `ofstream` 或 `ostringstream` 都应该能按预期工作，这就是 OO 继承强调的“可替换性”，派生类的对象可以替换基类对象，从而被客户端代码 `operator<<` 复用。

`iostream` 的继承体系多次违反了 Liskov 原则，这些地方继承的目的是为了复用基类的代码，下图中我把违规的继承关系用虚线标出。



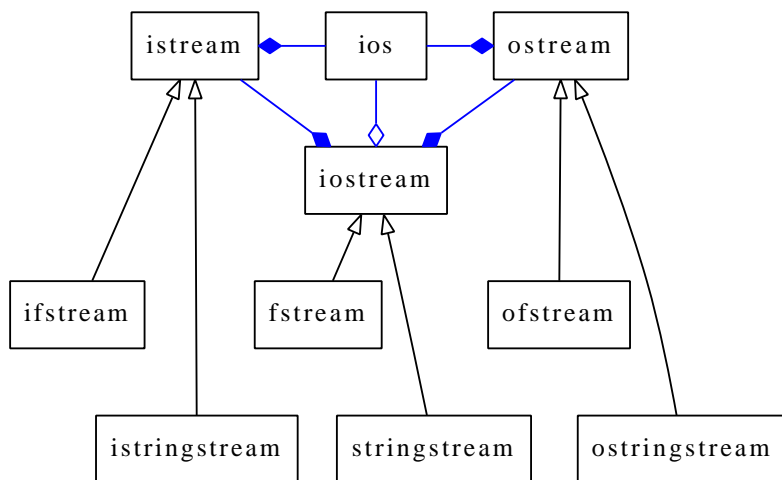
在现有的继承体系中，合理的有：

- `ifstream` **is-a** `istream`
- `istringstream` **is-a** `istream`
- `ofstream` **is-a** `ostream`
- `ostringstream` **is-a** `ostream`
- `fstream` **is-a** `iostream`
- `stringstream` **is-a** `iostream`

我认为不怎么合理的有：

- `ios` 继承 `ios_base`，有没有哪种情况下程序代码期待 `ios_base` 对象，但是客户可以传入一个 `ios` 对象替代之？如果没有，这里用 `public` 继承是不是违反 OO 原则？
- `istream` 继承 `ios`，有没有哪种情况下程序代码期待 `ios` 对象，但是客户可以传入一个 `istream` 对象替代之？如果没有，这里用 `public` 继承是不是违反 OO 原则？
- `ostream` 继承 `ios`，有没有哪种情况下程序代码期待 `ios` 对象，但是客户可以传入一个 `ostream` 对象替代之？如果没有，这里用 `public` 继承是不是违反 OO 原则？
- `iostream` 多重继承 `istream` 和 `ostream`。为什么 `iostream` 要同时继承两个 `non-interface class`？这是接口继承还是实现继承？是不是可以用组合 (`composition`) 来替代？（见《Effective C++ 第 3 版》[2, item 38]：通过组合模塑出 `has-a` 或“以某物实现”。《C++ 编程规范》[4, item 34]：尽可能以组合代替继承。）

用组合替换继承之后的体系：



注意到在新的设计中，只有真正的 **is-a** 关系采用了 **public** 继承，其他均以组合来代替，组合关系以红线表示。新的设计没有用的虚拟继承或多重继承。

其中 **iostream** 的新实现值得一提，代码结构如下：

```
class istream;
class ostream;

class iostream
{
public:
    istream& get_istream();
    ostream& get_ostream();
    virtual ~iostream();

    // ...
};
```

这样一来，在需要 **iostream** 对象表现得像 **istream** 的地方，调用 **get_istream()** 函数返回一个 **istream** 的引用；在需要 **iostream** 对象表现得像 **ostream** 的地方，调用 **get_ostream()** 函数返回一个 **ostream** 的引用。功能不受影响，而且代码更清晰，**istream** 和 **ostream** 也不必使用虚拟继承了。（我非常怀疑 **iostream class** 的真正价值，一个东西既可读又可写，说明是个 **sophisticated IO** 对象，为什么还用这么厚的 **OO** 封装？）

11.5.3 阳春的 locale

iostream 的故事还不止这些，它还包含一套阳春的 locale/facet 实现，这套实践中没人用的东西进一步增加了 iostream 的复杂度，而且不可避免地影响其性能。Nathan Myers 正是始作俑者⁴⁸。

ostream 自身定义的针对整数和浮点数的 operator<< 成员函数的函数体是：

```
ostream& ostream::operator<<(int val) // 或 double val
{
    bool failed =
        use_facet<num_put>(getloc()).put(
            ostreambuf_iterator(*this), *this, fill(), val).failed();
    // ...
}
```

它会调用 num_put::put(), 后者会去调用 num_put::do_put(), 而 do_put() 是个虚函数，没办法 inline。iostream 在性能方面的不足恐怕部分来自于此。这个虚函数白白浪费了把 template 的实现放到头文件应得的好处，编译和运行速度都快不起来。这就是为什么我说 iostream 在泛型方面不讨好。

我没有深入挖掘其中的细节，感兴趣的同学可以移步观看 facet 的继承体系：<http://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00431.html>

据此分析，我不认为以 iostream 为基础的上层程序库（比方说那些克服 iostream 格式化方面的缺点的库）有多大的实用价值。

11.5.4 臆造抽象

孟岩评价“iostream 最大的缺点是臆造抽象”，我非常赞同他老人家的观点。

这个评价同样适用于 Java 那一套叠床架屋的 InputStream、OutputStream、Reader、Writer 继承体系，.NET 也搞了这么一套繁文缛节。

乍看之下，用 input stream 表示一个可以“读”的数据流，用 output stream 表示一个可以“写”的数据流，屏蔽底层细节，面向接口编程，“符合面向对象原则”，似乎是一件美妙的事情。但是，真实的世界要残酷得多。

IO 是个极度复杂的东西，就拿最常见的 memory stream、file stream、socket stream 来说，它们之间的差异极大：

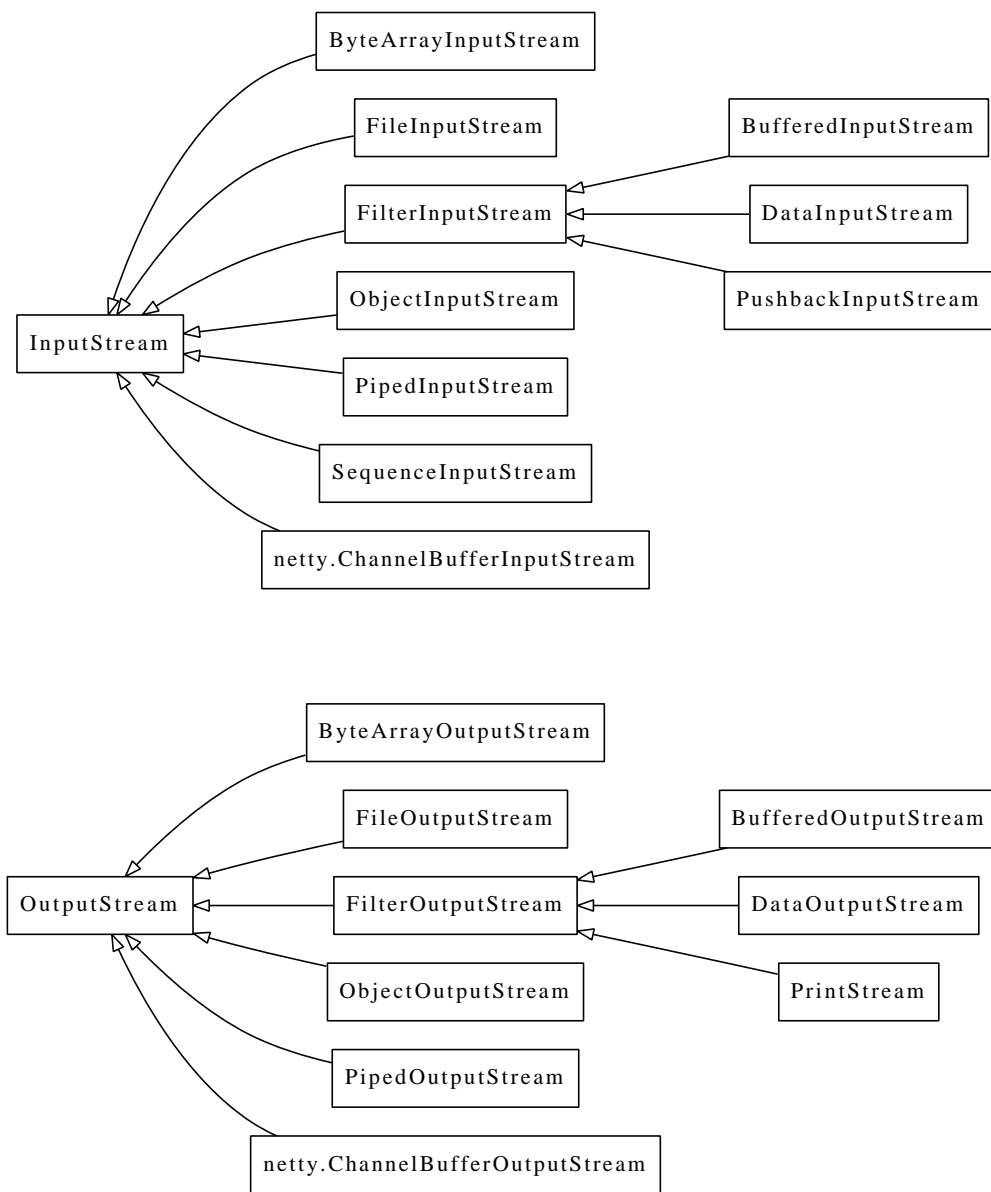
- 是单向 IO 还是双向 IO。只读或者只写？还是既可读又可写？

⁴⁸<http://www.cantrip.org/locale.html>

- 顺序访问还是随机访问。可不可以 `seek`? 可不可以退回 `n` 字节?
- 文本数据还是二进制数据。输入数据格式有误怎么办? 如何编写健壮的处理输入的代码?
- 有无缓冲。`write 500` 字节是否能保证完全写入? 有没有可能只写入了 300 字节? 余下 200 字节怎么办?
- 是否阻塞。会不会返回 `EWOULDBLOCK` 错误?
- 有哪些出错的情况。这是最难的, `memory stream` 几乎不可能出错, `file stream` 和 `socket stream` 的出错情况完全不同。`socket stream` 可能遇到对方断开连接, `file stream` 可能遇到超出磁盘配额。

根据以上列举的初步分析, 我不认为有办法设计一个公共的基类把各方面的情况都考虑周全。各种 IO 设施之间共性太小, 差异太大, 例外太多。如果硬要用面向对象来建模, 基类要么太瘦 (只放共性, 这个基类包含的 `interface functions` 没多大用), 要么太肥 (把各种 IO 设施的特性都包含进来, 这个基类包含的 `interface functions` 很多, 但是不是每一个都能调用)。

一个基类设计得好, 大家才愿意去继承它。比如 `Runnable` 是个很好的抽象, 有不计其数的实现。`InputStream/OutputStream` 好歹也有若干个实现 (见下图)。反观 `istream/ostream`, 只有标准库提供的两套默认实现, 在项目中极少有人会去继承并扩展它, 是不是说明 `istream/ostream` 这一套抽象不怎么灵光呢?



(当然, 假如 Java 有 C++ 那样强大的 `template` 机制, 上面这个继承体系能简化不少。)

若要在 C 语言里解决这个问题, 通常的办法是用一个 `int` 表示 IO 对象 (`file` 或 `PIPE` 或 `socket`), 然后配以 `read()`/`write()`/`lseek()`/`fcntl()` 等一系列全局函数,

程序员自己搭配组合。这个做法我认为比面向对象的方案要简洁高效。

`iostream` 在性能方面没有比 `stdio` 高多少，在健壮性方面多半不如 `stdio`，在灵活性方面受制于本身的复杂设计而难以让使用者自行扩展。目前看起来只适合一些简单的要求不高的应用，但是又不得不为它的复杂设计付出运行时代价，总之其定位有点不上不下。

在实际的项目中，我们可以提炼出一些简单高效的 `strip-down` 版本，在获得便利性的同时避免付出不必要的代价。

11.6 一个 300 行的 `memory buffer output stream`

我认为以 `operator<<` 来输出数据非常适合 `logging`，因此写了一个简单的 `muduo::LogStream` class。代码不到 300 行，完全独立于 `iostream`，代码位于 <https://github.com/chenshuo/recipes/blob/master/logging/>。

这个 `LogStream` 做到了类型安全和类型可扩展，效率也较高。它不支持定制格式化、不支持 `locale/facet`、没有继承、`buffer` 也没有继承与虚函数、没有动态分配内存、`buffer` 大小固定。简单地说，适合 `logging` 以及简单的字符串转换。这基本上是 Bjarne 在 1984 年写的 `ostream` 的翻版。

`LogStream` 的接口定义是

```
class Buffer;

class LogStream : boost::noncopyable
{
    typedef LogStream self;
public:

    self& operator<<(bool);

    self& operator<<(short);
    self& operator<<(unsigned short);
    self& operator<<(int);
    self& operator<<(unsigned int);
    self& operator<<(long);
    self& operator<<(unsigned long);
    self& operator<<(long long);
    self& operator<<(unsigned long long);

    self& operator<<(const void*);

    self& operator<<(float);
    self& operator<<(double);
```

```
// self& operator<<(long double);

self& operator<<(char);
// self& operator<<(signed char);
// self& operator<<(unsigned char);

self& operator<<(const char*);
self& operator<<(const string&);

void append(const char* data, int len);
const Buffer& buffer() const { return buffer_; }
void resetBuffer() { buffer_.reset(); }

private:
    Buffer buffer_;
};
```

LogStream 本身不是线程安全的，它不适合做线程间共享对象。正确的方式是每条 log 消息构造一个 LogStream，用完就扔。LogStream 的成本极低，这么做不会有什么性能损失。

目前这个 logging 库还在开发之中，只完成了 LogStream 这一部分。将来可能改用动态分配的 buffer，这样方便在线程之间传递数据。

11.6.1 整数到字符串的高效转换

muduo::LogStream 的整数转换是自己写的，用的是 Matthew Wilson 的算法，见前面第 8 节“带符号整数的除法与余数”。这个算法比 stdio 和 iostream 都要快。

11.6.2 浮点数到字符串的高效转换

目前 muduo::LogStream 的浮点数格式化采用的是 snprintf()。所以从性能上与 stdio 持平，比 ostream 快一些。

浮点数到字符串的转换是个复杂的话题，这个领域 20 年以来没有什么进展（目前的实现大都基于 David M. Gay 在 1990 年的工作《Correctly Rounded Binary-Decimal and Decimal-Binary Conversions》，代码 <http://netlib.org/fp/>），直到 2010 年才有突破。

Florian Loitsch 发明了新的更快的算法 Grisu3，他的论文《Printing floating-point numbers quickly and accurately with integers》发表在 PLDI 2010，代码见

Google V8 引擎，还有这里 <http://code.google.com/p/double-conversion/>。有兴趣的同学可以阅读这篇博客⁴⁹。

将来 `muduo::LogStream` 可能会改用 `Grisu3` 算法实现浮点数转换。

11.6.3 性能对比

由于 `muduo::LogStream` 抛掉了很多负担，可以预见它的性能好于 `ostringstream` 和 `stdio`。我做了一个简单的性能测试，结果如下表。表中的数字是打印 1 000 000 次的用时，以毫秒为单位，越小越好。

64-bit code			
	<code>snprintf</code>	<code>ostringstream</code>	<code>LogStream</code>
<code>int</code>	499	363	113
<code>double</code>	2315	3835	2338
<code>int64_t</code>	486	347	145
<code>void*</code>	419	330	47
32-bit code			
	<code>snprintf</code>	<code>ostringstream</code>	<code>LogStream</code>
<code>int</code>	544	453	116
<code>double</code>	2241	4030	2267
<code>int64_t</code>	725	958	654
<code>void*</code>	690	425	65

从上表看出，`ostreamstream` 有时候比 `snprintf()` 快，有时候比它慢，`muduo::LogStream` 比它们两个都快得多（`double` 类型除外）。

11.6.4 泛型编程

其他程序库如何使用 `LogStream` 作为输出呢？办法很简单，用模板。

前面我们定义了 `Date class` 针对 `std::ostream` 的 `operator<<`，只要稍作修改就能同时适用于 `std::ostream` 和 `LogStream`。而且 `Date` 的头文件不再需要 `include <ostream>`，降低了耦合。

⁴⁹<http://www.serpentine.com/blog/2011/06/29/here-be-dragons-advances-in-problems-you-didnt-even-know-you-had/>

```

// 不必包含 LogStream 或 ostream 头文件
class Date
{
public:
    Date(int year, int month, int day)
        : year_(year), month_(month), day_(day)
        {
        }

-   void writeTo(std::ostream& os) const
+   template<typename OStream>
+   void writeTo(OStream& os) const
    {
        char buf[32];
        snprintf(buf, sizeof buf, "%d-%02d-%02d", year_, month_, day_);
        os << buf;
    }

private:
    int year_, month_, day_;
};

-std::ostream& operator<<(std::ostream& os, const Date& date)
+template<typename OStream>
+OStream& operator<<(OStream& os, const Date& date)
{
    date.writeTo(os);
    return os;
}

```

11.6.5 格式化

`muduo::LogStream` 本身不支持格式化，不过我们很容易为它做扩展，定义一个简单的 `Fmt class` 就行，而且不影响 `stream` 的状态。

```

class Fmt : boost::noncopyable
{
public:
    template<typename T>
    Fmt(const char* fmt, T val)
    {
        BOOST_STATIC_ASSERT(boost::is_arithmetic<T>::value == true);
        length_ = snprintf(buf_, sizeof buf_, fmt, val);
    }

    const char* data() const { return buf_; }
    int length() const { return length_; }

private:
    char buf_[32];
}

```

```
    int length_;
};

inline LogStream& operator<<(LogStream& os, const Fmt& fmt)
{
    os.append(fmt.data(), fmt.length());
    return s;
}
```

使用方法

```
LogStream os;
double x = 19.82;
int y = 43;
os << Fmt("%.3f", x) << Fmt("%4d", y);
```

11.7 现实的 C++ 程序如何做文件 IO

举三个例子，Google Protobuf Compiler、Google leveldb、Kyoto Cabinet。

11.7.1 Google Protobuf Compiler

Google Protobuf 是一种高效的网络传输格式，它用一种协议描述语言来定义消息格式，并且自动生成序列化代码。Protobuf Compiler 是这种“协议描述语言”的编译器，它读入协议文件 .proto，编译生成 C++、Java、Python 代码。proto 文件是个文本文件，然而 Protobuf Compiler 并没有使用 ifstream 来读取它，而是使用了自己的 FileInputStream 来读取文件。

大致代码流程如下：

1. ZeroCopyInputStream⁵⁰ 是一个抽象基类
2. FileInputStream⁵¹ 继承并实现了 ZeroCopyInputStream
3. Tokenizer⁵² 是词法分析器，它把 proto 文件分解为一个一个字元 (token)。Tokenizer 的构造函数以 ZeroCopyInputStream 为参数，从该 stream 读入文本。

⁵⁰http://code.google.com/p/protobuf/source/browse/trunk/src/google/protobuf/io/zero_copy_stream.h#122

⁵¹http://code.google.com/p/protobuf/source/browse/trunk/src/google/protobuf/io/zero_copy_stream_impl.h#55

⁵²<http://code.google.com/p/protobuf/source/browse/trunk/src/google/protobuf/io/tokenizer.h#75>

4. Parser⁵³ 是语法分析器，它把 proto 文件解析为语法树，以 FileDescriptorProto 表示。Parser 的构造函数以 Tokenizer 为参数，从它读入字元。

由此可见，即便是读取文本文件，C++ 程序也不一定要用 ifstream。

11.7.2 Google leveldb

Google leveldb 是一个高效的持久化 key-value db。⁵⁴ 它定义了三个精简的 interface 用于文件输入输出：

- SequentialFile
- RandomAccessFile
- WritableFile

接口函数如下

```
struct Slice {
    const char* data_;
    size_t size_;
};

// A file abstraction for reading sequentially through a file
class SequentialFile {
public:
    SequentialFile() { }
    virtual ~SequentialFile();

    virtual Status Read(size_t n, Slice* result, char* scratch) = 0;
    virtual Status Skip(uint64_t n) = 0;
};

// A file abstraction for randomly reading the contents of a file.
class RandomAccessFile {
public:
    RandomAccessFile() { }
    virtual ~RandomAccessFile();

    virtual Status Read(uint64_t offset, size_t n, Slice* result,
                        char* scratch) const = 0;
};

// A file abstraction for sequential writing. The implementation
```

⁵³<http://code.google.com/p/protobuf/source/browse/trunk/src/google/protobuf/compiler/parser.h#59>

⁵⁴<http://code.google.com/p/leveldb>

```
// must provide buffering since callers may append small fragments
// at a time to the file.
class WritableFile {
public:
    WritableFile() { }
    virtual ~WritableFile();

    virtual Status Append(const Slice& data) = 0;
    virtual Status Close() = 0;
    virtual Status Flush() = 0;
    virtual Status Sync() = 0;
};
```

leveldb 明确区分 **input** 和 **output**，进一步它又把 **input** 分为 **sequential** 和 **random access**，然后提炼出了三个简单的接口，每个接口只有屈指可数的几个函数。这几个接口在各个平台下的实现也非常简单明了^{55 56}，一看就懂。

注意这三个接口使用了虚函数，我认为这是正当的，因为一次 IO 往往伴随着 **context switch**，虚函数的开销比起 **context switch** 来可以忽略不计。相反，**iostream** 每次 **operator<<()** 就调用虚函数，似乎不太明智。

11.7.3 Kyoto Cabinet

Kyoto Cabinet 也是一个 **key-value db**，是前几年流行的 Tokyo Cabinet 的升级版。它采用了与 leveldb 不同的文件抽象。

KC 定义了一个 **File class**，同时包含了读写操作，这是个 **fat interface**。⁵⁷

在具体实现方面，它没有使用虚函数，而是采用 **#ifdef** 来区分不同的平台⁵⁸，等于把两份独立的代码写到了同一个文件里边。

相比之下，Google leveldb 的做法更高明一些。

11.8 小结

在 C++ 项目里边自己写个 **File class**，把项目用到的文件 IO 功能简单封装一下（以 **RAII** 手法封装 **FILE*** 或者 **file descriptor** 都可以，视情况而定），通常就能满足

⁵⁵http://code.google.com/p/leveldb/source/browse/trunk/util/env_posix.cc#35

⁵⁶http://code.google.com/p/leveldb/source/browse/trunk/util/env_chromium.cc#176

⁵⁷http://fallabs.com/kyotocabinet/api/classkyotocabinet_1_1File.html

⁵⁸<http://code.google.com/p/read-taobao-code/source/browse/trunk/tair/src/storage/kdb/kyotocabinet/kcfile.cc>

需要。记得把拷贝构造和赋值操作符禁用，在析构函数里释放资源，避免泄露内部的 `handle`，这样就能自动避免很多 C 语言文件操作的常见错误。

如果要用 `stream` 方式做 `logging`，可以抛开繁重的 `iostream` 自己写一个简单的 `LogStream`，重载几个 `operator<<` 操作符，用起来一样方便；而且可以用 `stack buffer`，轻松做到线程安全与高效。

12 值语义与数据抽象

本文是第11节《`iostream` 的用途与局限》的后续，在11.3 “`iostream` 与标准库其他组件的交互”这一小节，我简单地提到 `iostream` 的对象和 C++ 标准库中的其他对象（主要是容器和 `string`）具有不同的语义，主要体现在 `iostream` 不能拷贝或赋值。今天全面谈一谈我对这个问题的理解。

本文的“对象”定义较为宽泛，`a region of memory that has a type`，在这个定义下，`int`、`double`、`bool` 变量都是对象。

12.1 什么是值语义

值语义 (value semantics) 指的是对象的拷贝与原对象无关，就像拷贝 `int` 一样。C++ 的内置类型 (`bool/int/double/char`) 都是值语义，标准库里的 `complex<>`、`pair<>`、`vector<>`、`map<>`、`string` 等等类型也都是值语义，拷贝之后就与原对象脱离关系。Java 语言的 `primitive types` 也是值语义。

与值语义对应的是“对象语义/object semantics”，或者叫做引用语义 (reference semantics)，由于“引用”一词在 C++ 里有特殊含义，所以我在本文中使用“对象语义”这个术语。对象语义指的是面向对象意义下的对象，对象拷贝是禁止的。例如 `muduo` 里的 `Thread` 是对象语义，拷贝 `Thread` 是无意义的，也是被禁止的：因为 `Thread` 代表线程，拷贝一个 `Thread` 对象并不能让系统增加一个一模一样的线程。

同样的道理，拷贝一个 `Employee` 对象是没有意义的，一个雇员不会变成两个雇员，他也不会领两份薪水。拷贝 `TcpConnection` 对象也没有意义，系统里边只有一个 TCP 连接，拷贝 `TcpConnection` 对象不会让我们拥有两个连接。`Printer` 也是不能拷贝的，系统只连接了一个打印机，拷贝 `Printer` 并不能凭空增加打印机。此总结，面向对象意义下的“对象”是 `non-copyable`。

Java 里边的 `class` 对象都是对象语义/引用语义。

```
ArrayList<Integer> a = new ArrayList<Integer>();  
ArrayList<Integer> b = a;
```

那么 **a** 和 **b** 指向的是同一个 **ArrayList** 对象，修改 **a** 同时也会影响 **b**。

值语义与 **immutable** 无关。Java 有 **value object** 一说，按 (PoEAA 486) 的定义，它实际上是 **immutable object**，例如 **String**、**Integer**、**BigInteger**、**joda.time.Date-Time** 等等（因为 Java 没有办法实现真正的值语义 **class**，只好用 **immutable object** 来模拟）。尽管 **immutable object** 有其自身的用处，但不是本文的主题。**muduo** 中的 **Date**、**Timestamp** 也都是 **immutable** 的。

C++ 中的值语义对象也可以是 **mutable**，比如 **complex<>**、**pair<>**、**vector<>**、**map<>**、**string** 都是可以修改的。**muduo** 的 **InetAddress** 和 **Buffer** 都具有值语义，它们都是可以修改的。

值语义的对象不一定是 **POD**，例如 **string** 就不是 **POD**，但它是值语义的。

值语义的对象不一定小，例如 **vector<int>** 的元素可多可少，但它始终是值语义的。当然，很多值语义的对象都是小的，例如 **complex<>**、**muduo::Date**、**muduo::Timestamp**。

12.2 值语义与生命期

值语义的一个巨大好处是生命期管理很简单，就跟 **int** 一样——你不需要操心 **int** 的生命期。值语义的对象要么是 **stack object**，或者直接作为其他 **object** 的成员，因此我们不用担心它的生命期（一个函数使用自己 **stack** 上的对象，一个成员函数使用自己的数据成员对象）。相反，对象语义的 **object** 由于不能拷贝，我们只能通过指针或引用来使用它。

一旦使用指针和引用来操作对象，那么就要担心所指的对象是否已被释放，这一度是 C++ 程序 **bug** 的一大来源。此外，由于 C++ 只能通过指针或引用来获得多态性，那么在 C++ 里从事基于继承和多态的面向对象编程有其本质的困难——对象生命期管理（资源管理）。

考虑一个简单的对象建模——家长与子女：**a Parent has a Child, a Child knows his/her Parent**。在 Java 里边很好写，不用担心内存泄漏，也不用担心空悬指针：

```
public class Parent  
{  
    private Child myChild;
```

```
}

public class Child
{
    private Parent myParent;
}
```

只要正确初始化 `myChild` 和 `myParent`，那么 Java 程序员就不用担心出现访问错误。一个 `handle` 是否有效，只需要判断其是否 `non null`。

在 C++ 里边就要为资源管理费一番脑筋：`Parent` 和 `Child` 都代表的是真人，肯定是不能拷贝的，因此具有对象语义。`Parent` 是直接持有 `Child` 吗？抑或 `Parent` 和 `Child` 通过指针互指？`Child` 的生命期由 `Parent` 控制吗？如果还有 `ParentClub` 和 `School` 两个 `class`，分别代表家长俱乐部和学校：`ParentClub has many Parent(s)`，`School has many Child(ren)`，那么如何保证它们始终持有有效的 `Parent` 对象和 `Child` 对象？何时才能安全地释放 `Parent` 和 `Child`？

直接但是易错的写法：

```
class Child;

class Parent : boost::noncopyable
{
private:
    Child* myChild;
};

class Child : boost::noncopyable
{
private:
    Parent* myParent;
};
```

如果直接使用指针作为成员，那么如何确保指针的有效性？如何防止出现空悬指针？`Child` 和 `Parent` 由谁负责释放？在释放某个 `Parent` 对象的时候，如何确保程序中没有指向它的指针？在释放某个 `Child` 对象的时候，如何确保程序中没有指向它的指针？

这一系列问题一度是 C++ 面向对象编程头疼的问题，不过现在有了 `smart pointer`，我们可以借助 `smart pointer` 把对象语义转换为值语义⁵⁹，从而轻松解决对象生命期：让 `Parent` 持有 `Child` 的 `smart pointer`，同时让 `Child` 持有 `Parent` 的 `smart pointer`，这样始终引用对方的时候就不用担心出现空悬指针。当然，其中一个

⁵⁹即像持有 `int` 一样持有对象（的智能指针）

smart pointer 应该是 **weak reference**，否则会出现循环引用，导致内存泄漏。到底哪一个 **weak reference**，则取决于具体应用场景。

如果 **Parent** 拥有 **Child**，**Child** 的生命期由其 **Parent** 控制，**Child** 的生命期小于 **Parent**，那么代码就比较简单：

```
class Parent;
class Child : boost::noncopyable
{
public:
    explicit Child(Parent* myParent_)
        : myParent(myParent_)
    {
    }

private:
    Parent* myParent;
};

class Parent : boost::noncopyable
{
public:
    Parent()
        : myChild(new Child(this))
    {
    }

private:
    boost::scoped_ptr<Child> myChild;
};
```

在上面这个设计中，**Child** 的指针不能泄露给外界，否则仍然有可能出现空悬指针。

如果 **Parent** 与 **Child** 的生命期相互独立，就要麻烦一些：

```
class Parent;
typedef boost::shared_ptr<Parent> ParentPtr;

class Child : boost::noncopyable
{
public:
    explicit Child(const ParentPtr& myParent_)
        : myParent(myParent_)
    {
    }

private:
    boost::weak_ptr<Parent> myParent;
};
typedef boost::shared_ptr<Child> ChildPtr;
```

```

class Parent : public boost::enable_shared_from_this<Parent>,
               private boost::noncopyable
{
public:
    Parent()
    {
    }

    void addChild()
    {
        myChild.reset(new Child(shared_from_this()));
    }

private:
    ChildPtr myChild;
};

int main()
{
    ParentPtr p(new Parent);
    p->addChild();
}

```

上面这个 `shared_ptr+weak_ptr` 的做法似乎有点小题大做。

考虑一个稍微复杂一点的对象模型：a Child has parents: mom and dad; a Parent has one or more Child(ren); a Parent knows his/her spouser. 这个对象模型用 Java 表述一点都不复杂，垃圾收集会帮我们搞定对象生命期。

```

public class Parent
{
    private Parent mySpouser;
    private ArrayList<Child> myChildren;
}

public class Child
{
    private Parent myMom;
    private Parent myDad;
}

```

如果用 C++ 来实现，如何才能避免出现空悬指针，同时避免出现内存泄漏呢？借助 `shared_ptr` 把裸指针转换为值语义，我们就不用担心这两个问题了：

```

class Parent;
typedef boost::shared_ptr<Parent> ParentPtr;

class Child : boost::noncopyable

```

```
{
public:
    explicit Child(const ParentPtr& myMom_,
                  const ParentPtr& myDad_)
        : myMom(myMom_),
          myDad(myDad_)
    {
    }

private:
    boost::weak_ptr<Parent> myMom;
    boost::weak_ptr<Parent> myDad;
};
typedef boost::shared_ptr<Child> ChildPtr;

class Parent : boost::noncopyable
{
public:
    Parent()
    {
    }

    void setSpouser(const ParentPtr& spouser)
    {
        mySpouser = spouser;
    }

    void addChild(const ChildPtr& child)
    {
        myChildren.push_back(child);
    }

private:
    boost::weak_ptr<Parent> mySpouser;
    std::vector<ChildPtr> myChildren;
};

int main()
{
    ParentPtr mom(new Parent);
    ParentPtr dad(new Parent);
    mom->setSpouser(dad);
    dad->setSpouser(mom);
    {
        ChildPtr child(new Child(mom, dad));
        mom->addChild(child);
        dad->addChild(child);
    }
    {
        ChildPtr child(new Child(mom, dad));
        mom->addChild(child);
        dad->addChild(child);
    }
}
```

```
}
```

如果不使用 `smart pointer`，用 C++ 做面向对象编程将会困难重重。

12.3 值语义与标准库

C++ 要求凡是能放入标准容器的类型必须具有值语义。准确地说：`type` 必须是 `SGIAssignable concept` 的 `model`。但是，由于 C++ 编译器会为 `class` 默认提供 `copy constructor` 和 `assignment operator`，因此除非明确禁止，否则 `class` 总是可以作为标准库的元素类型——尽管程序可以编译通过，但是隐藏了资源管理方面的 `bug`。

因此，在写一个 `class` 的时候，先让它继承 `boost::noncopyable`，几乎总是正确的。

在现代 C++ 中，一般不需要自己编写 `copy constructor` 或 `assignment operator`，因为只要每个数据成员都具有值语义的话，编译器自动生成的 `member-wise copying&assigning` 就能正常工作；如果以 `smart ptr` 为成员来持有其他对象，那么就能自动启用或禁用 `copying&assigning`。例外：编写 `HashMap` 这类底层库时还是需要自己实现 `copy control`。

12.4 值语义与 C++ 语言

C++ 的 `class` 本质上是值语义的，这才会出现 `object slicing` 这种语言独有的问题，也才会需要程序员注意 `pass-by-value` 和 `pass-by-const-reference` 的取舍。在其他面向对象编程语言中，这都不需要费脑筋。

值语义是 C++ 语言的三大约束之一，C++ 的设计初衷是让用户定义的类型 (`class`) 能像内置类型 (`int`) 一样工作，具有同等的地位。为此 C++ 做了以下设计（妥协）：

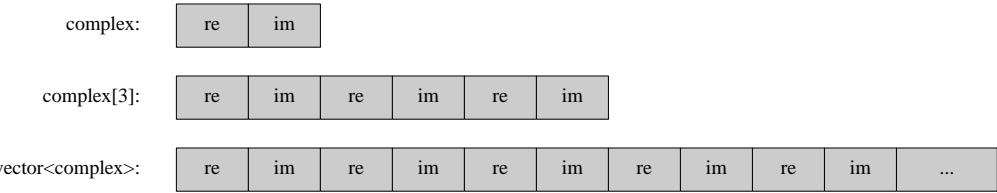
- `class` 的 `layout` 与 C `struct` 一样，没有额外的开销。定义一个“只包含一个 `int` 成员的 `class`”的对象开销和定义一个 `int` 一样。
- 甚至 `class data member` 都默认是 `uninitialized`，因为函数局部的 `int` 是 `uninitialized`。
- `class` 可以在 `stack` 上创建，也可以在 `heap` 上创建。因为 `int` 可以是 `stack variable`。

- `class` 的数组就是一个个 `class` 对象挨着，没有额外的 `indirection`。因为 `int` 数组就是这样。
- 编译器会为 `class` 默认生成 `copy constructor` 和 `assignment operator`。其他语言没有 `copy constructor` 一说，也不允许重载 `assignment operator`。C++ 的对象默认是可以拷贝的，这是一个尴尬的特性。
- 当 `class type` 传入函数时，默认是 `make a copy`（除非参数声明为 `reference`）。因为把 `int` 传入函数时是 `make a copy`。

C++ 的“函数调用”比其他语言复杂之处在于参数传递和返回值传递。C、Java 等语言都是传值，简单地复制几个字节的内存就行了。但是 C++ 对象是值语义，如果以 `pass-by-value` 方式把对象传入函数，会涉及拷贝构造。代码里看到一句简单的函数调用，实际背后发生的可能是一长串对象构造操作，因此减少无谓的临时对象是 C++ 代码优化的关键之一。

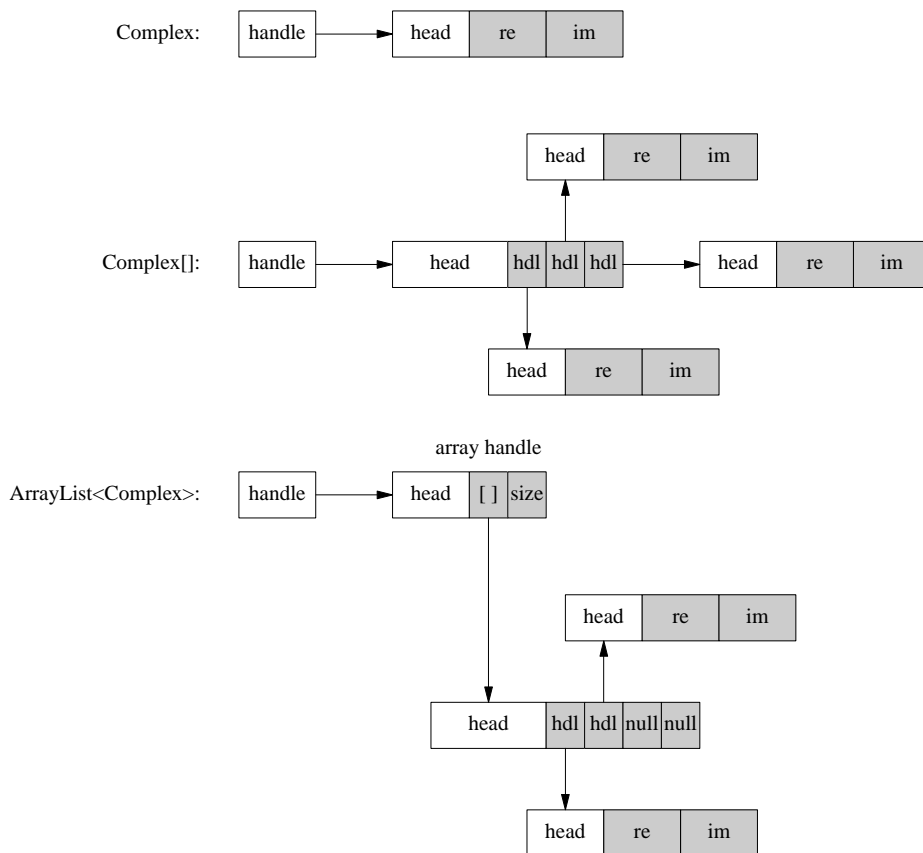
- 当函数返回一个 `class type` 时，只能通过 `make a copy`（C++ 不得不定义 `RVO` 来解决性能问题）。因为函数返回 `int` 时是 `make a copy`。
- 以 `class type` 为成员时，数据成员是嵌入的。例如 `pair<complex<double>, size_t>` 的 `layout` 就是 `complex<double>` 挨着 `size_t`。

这些设计带来了性能上的好处，原因是 `memory locality`。比方说我们在 C++ 里定义 `complex<double>` `class`，`array of complex<double>`，`vector<complex<double>>`，它们的 `layout` 分别是：（`re` 和 `im` 分别是复数的实部和虚部。）



而如果我们在 Java 里干同样的事情，`layout` 大不一样，`memory locality` 也差很多：⁶⁰

⁶⁰图中的 `handle` 是 Java 的 `reference`，为了避免与 C++ 引用混淆，这里换个写法。



Java 里边每个 object 都有 header，在常见的 JVM 中有两个 word 的开销。对比 Java 和 C++，可见 C++ 的对象模型要紧凑得多。

12.5 什么是数据抽象

本节谈一谈与值语义紧密相关的数据抽象 (data abstraction)，解释为什么它是与面向对象并列的一种编程范式，为什么支持面向对象的编程语言不一定支持数据抽象。C++ 在最初的时候是以 data abstraction 为卖点，不过随着时间的流逝，现在似乎很多人只知 Object-Oriented，不知 data abstraction 了。C++ 的强大之处在于“抽象”不以性能损失为代价，下一篇文章我们将看到具体例子。

数据抽象 (data abstraction) 是与面向对象 (object-oriented) 并列的一种编程范式 (programming paradigm)。说“数据抽象”或许显得陌生，它的另外一个名字“抽象数据类型/abstract data type/ADT”想必如雷贯耳。

“支持数据抽象”一直是 C++ 语言的设计目标，Bjarne Stroustrup 在他的《The C++ Programming Language》第二版（1991 年出版）中写道 [2nd]:

The C++ programming language is designed to

- be a better C
- support data abstraction
- support object-oriented programming

这本书第三版（1997 年出版）[3rd] 增加了一条：

C++ is a general-purpose programming language with a bias towards systems programming that

- is a better C,
- supports data abstraction,
- supports object-oriented programming, and
- supports generic programming.

在 C++ 的早期文献⁶¹中中有一篇 Bjarne Stroustrup 在 1984 年写的《Data Abstraction in C++》⁶²。在这个页面还能找到 Bjarne 写的关于 C++ 操作符重载和复数运算的文章，作为数据抽象的详解与范例。可见 C++ 早期是以数据抽象为卖点的，支持数据抽象是 C++ 相对于 C 的一大优势。

作为语言的设计者，Bjarne 把数据抽象作为 C++ 的四个子语言之一。这个观点不是普遍接受的，比如作为语言的使用者，Scott Meyers 在《Effective C++ 第三版》中把 C++ 分为四个子语言：C、Object-Oriented C++、Template C++、STL。在 Scott Meyers 的分类法中，就没有出现数据抽象，而是归入了 object-oriented C++。

那么到底什么是数据抽象？ 简单的说，数据抽象是用来描述 (抽象) 数据结构的。数据抽象就是 ADT。一个 ADT 主要表现为它支持的一些操作，比方说 `stack.push`、`stack.pop`，这些操作应该具有明确的时间和空间复杂度。另外，一个 ADT 可以隐藏其实现细节，比方说 `stack` 既可以用动态数组实现，又可以用链表实现。

⁶¹http://www.softwarepreservation.org/projects/c_plus_plus/index.html#cfont

⁶²http://www.softwarepreservation.org/projects/c_plus_plus/cfront/release_-e/doc/DataAbstraction.pdf

按照这个定义，数据抽象和基于对象 (object-based) 很像，那么它们的区别在哪里？语义不同。ADT 通常是值语义，而 object-based 是对象语义。（这两种语义的定义见前一节《什么是值语义》12.1）。ADT class 是可以拷贝的，拷贝之后的 instance 与原 instance 脱离关系。

比方说

```
stack<int> a;  
a.push(10);  
stack<int> b = a;  
b.pop();
```

这时候 a 里仍然有元素 10。

12.5.1 C++ 标准库中的数据抽象

C++ 标准库里 complex<>、pair<>、vector<>、list<>、map<>、set<>、string、stack、queue 都是数据抽象的例子。vector 是动态数组，它的主要操作有 size()、begin()、end()、push_back() 等等，这些操作不仅含义清晰，而且计算复杂度都是常数。类似的，list 是链表，map 是有序关联数组，set 是有序集合、stack 是 FILO 栈、queue 是 FIFO 队列。“动态数组”、“链表”、“有序集合”、“关联数组”、“栈”、“队列”都是定义明确（操作、复杂度）的抽象数据类型。

12.5.2 数据抽象与面向对象的区别

本文把 data abstraction、object-based、object-oriented 视为三个编程范式。这种细致的分类或许有助于理解区分它们之间的差别。

庸俗地讲，面向对象 (object-oriented) 有三大特征：封装、继承、多态。而基于对象 (object-based) 则只有封装，没有继承和多态，即只有具体类，没有抽象接口。它们两个都是对象语义。

面向对象真正核心的思想是消息传递 (messaging)，“封装继承多态”只是表象。这一点孟岩⁶³和王益⁶⁴都有精彩的论述，陈硕不再赘言。

数据抽象与它们两个的界限在于“语义”，数据抽象不是对象语义，而是值语义。比方说 muduo 里的 TcpConnection 和 Buffer 都是具体类，但前者是基于对象的 (object-based)，而后者是数据抽象。

⁶³<http://blog.csdn.net/myan/article/details/5928531>

⁶⁴<http://cxwangyi.wordpress.com/2011/06/19/杂谈现代高级编程语言/>

类似的, `muduo::Date`、`muduo::Timestamp` 都是数据抽象。尽管这两个 `classes` 简单到只有一个 `int/long` 数据成员, 但是它们各自定义了一套操作 (`operation`), 并隐藏了内部数据, 从而让它从 `data aggregation` 变成了 `data abstraction`。

数据抽象是针对“数据”的, 这意味着 `ADT class` 应该可以拷贝, 只要把数据复制一份就行了。如果一个 `class` 代表了其他资源 (文件、员工、打印机、账号), 那么它就是 `object-based` 或 `object-oriented`, 而不是数据抽象。

`ADT class` 可以作为 `Object-based/object-oriented class` 的成员, 但反过来不成立, 因为这样一来 `ADS class` 的拷贝就失去意义了。

12.6 数据抽象所需的语言设施

不是每个语言都支持数据抽象, 下面简要列出“数据抽象”所需的语言设施。

支持数据聚合 数据聚合 `data aggregation`, 或者 `value aggregates`。即定义 `C-style struct`, 把有关数据放到同一个 `struct` 里。`FORTAN77` 没有这个能力, `FORTAN77` 无法实现 `ADT`。这种数据聚合 `struct` 是 `ADT` 的基础, `struct List`、`struct HashTable` 等能把链表和哈希表结构的数据放到一起, 而不是用几个零散的变量来表示它。

全局函数与重载 例如我定义了 `complex`, 那么我可以同时定义 `complex sin(const complex& x);` 和 `complex exp(const complex& x);` 等等全局函数来实现复数的三角函数和指数运算。`sin()` 和 `exp()` 不是 `complex` 的成员, 而是全局函数 `double sin(double)` 和 `double exp(double)` 的重载。这样能让 `double a = sin(b);` 和 `complex a = sin(b);` 具有相同的代码形式, 而不必写成 `complex a = b.sin();`。

`C` 语言可以定义全局函数, 但是不能与已有的函数重名, 也就没有重载。`Java` 没有全局函数, 而且 `Math class` 是封闭的, 并不能往其中添加 `sin(Complex)`。

成员函数与 `private` 数据 数据也可以声明为 `private`, 防止外界意外修改。不是每个 `ADT` 都适合把数据声明为 `private`, 例如 `complex`、`point`、`pair<>` 这样的 `ADT` 使用 `public data` 更加合理。

要能够在 `struct` 里定义操作, 而不是只能用全局函数来操作 `struct`。比方说 `vector` 有 `push_back()` 操作, `push_back` 是 `vector` 的一部分, 它必须直接修改 `vector` 的 `private data members`, 因此无法定义为全局函数。

这两点其实就是定义 `class`，现在的语言都能直接支持，C 语言除外。

拷贝控制 (copy control) `copy control` 是拷贝 `stack a; stack b = a;` 和赋值 `stack b; b = a;` 的合称。

当拷贝一个 ADT 时会发生什么？比方说拷贝一个 `stack`，是不是应该把它的每个元素按值拷贝到新 `stack`？

如果语言支持显示控制对象的生命期（比方说 C++ 的确定性析构），而 ADT 用到了动态分配的内存，那么 `copy control` 更为重要，不然如何防止访问已经失效的对象？

由于 C++ `class` 是值语义，`copy control` 是实现深拷贝的必要手段。而且 ADT 用到的资源只涉及动态分配的内存，所以深拷贝是可行的。相反，`object-based` 编程风格中的 `class` 往往代表某样真实的事物（`Employee`、`Account`、`File` 等等），深拷贝无意义。

C 语言没有 `copy control`，也没有办法防止拷贝，一切要靠程序员自己小心在意。`FILE*` 可以随意拷贝，但是只要关闭其中一个 `copy`，其他 `copies` 也都失效了，跟空悬指针一般。整个 C 语言对待资源（`malloc` 得到的内存，`open()` 打开的文件，`socket()` 打开的连接）都是这样，用整数或指针来代表（即“句柄”）。而整数和指针类型的“句柄”是可以随意拷贝的，很容易就造成重复释放、遗漏释放、使用已经释放的资源等等常见错误。这方面 C++ 是一个显著的进步，`boost::noncopyable` 是 `boost` 里最值得推广的库。

操作符重载 如果要写动态数组，我们希望能像使用内置数组一样使用它，比如支持下标操作。C++ 可以重载 `operator[]` 来做到这一点。

如果要写复数，我们系统能像使用内置的 `double` 一样使用它，比如支持加减乘除。C++ 可以重载 `operator+` 等操作符来做到这一点。

如果要写日期时间，我们希望它能直接用大于小于号来比较先后，用 `==` 来判断是否相等。C++ 可以重载 `operator<` 等操作符来做到这一点。

这要求语言能重载成员与全局操作符。操作符重载是 C++ 与生俱来的特性，1984 年的 CFront E 就支持操作符重载，并且提供了一个 `complex class`，这个 `class` 与目前标准库的 `complex<>` 在使用上无区别。

如果没有操作符重载，那么用户定义的 ADT 与内置类型用起来就不一样（想想有的语言要区分 `==` 和 `equals`，代码写起来实在很累赘）。Java 里有 `BigInteger`，但是 `BigInteger` 用起来和普通 `int/long` 大不相同：

```
// Java code

public static BigInteger mean(BigInteger x, BigInteger y) {
    BigInteger two = BigInteger.valueOf(2);
    return x.add(y).divide(two);
}

public static long mean(long x, long y) {
    return (x + y) / 2;
}
```

当然，操作符重载容易被滥用，因为这样显得很酷。我认为只在 ADT 表示一个“数值”的时候才适合重载加减乘除，其他情况下用具名函数为好，因此 `muduo::Timestamp` 只重载了关系操作符，没有重载加减操作符。另外一个理由见第3节《采用有利于版本管理的代码格式》。

效率无损 “抽象”不代表低效。在 C++ 中，提高抽象的层次并不会降低效率。不然的话，人们宁可在低层次上编程，而不愿使用更便利的抽象，数据抽象也就失去了市场。后面我们将看到一个具体的例子。

模板与泛型 如果我写了一个 `int vector`，那么我不想为 `double` 和 `string` 再实现一遍同样的代码。我应该把 `vector` 写成 `template`，然后用不同的类型来具现化它，从而得到 `vector<int>`、`vector<double>`、`vector<complex>`、`vector<string>` 等等具体类型。

不是每个 ADT 都需要这种泛型能力，一个 `Date class` 就没必要让用户指定该用哪种类型的整数，`int32_t` 足够了。

根据上面的要求，不是每个面向对象语言都能原生支持数据抽象，也说明数据抽象不是面向对象的子集。

12.7 数据抽象的例子

下面我们看看数值模拟 N-body 问题的两个程序，前一个用 C 语言，后一个是 C++ 的。这个例子来自编程语言的性能对比网站⁶⁵。

⁶⁵<http://shootout.alioth.debian.org/gp4/benchmark.php?test=nbody&lang=all>

两个程序使用了相同的算法。

C 语言版，完整代码见⁶⁶，下面是代码骨干。`planet` 保存与行星位置、速度、质量，位置和速度各有三个分量，程序模拟几大行星在三维空间中受引力支配的运动。

```
// C code

struct planet
{
    double x, y, z;
    double vx, vy, vz;
    double mass;
};

void advance(int nbodies, struct planet *bodies, double dt)
{
    for (int i = 0; i < nbodies; i++)
    {
        struct planet *p1 = &(bodies[i]);
        for (int j = i + 1; j < nbodies; j++)
        {
            struct planet *p2 = &(bodies[j]);
            double dx = p1->x - p2->x;
            double dy = p1->y - p2->y;
            double dz = p1->z - p2->z;
            double distance_squared = dx * dx + dy * dy + dz * dz;
            double distance = sqrt(distance_squared);
            double mag = dt / (distance * distance_squared);
            p1->vx -= dx * p2->mass * mag;
            p1->vy -= dy * p2->mass * mag;
            p1->vz -= dz * p2->mass * mag;
            p2->vx += dx * p1->mass * mag;
            p2->vy += dy * p1->mass * mag;
            p2->vz += dz * p1->mass * mag;
        }
    }
    for (int i = 0; i < nbodies; i++)
    {
        struct planet * p = &(bodies[i]);
        p->x += dt * p->vx;
        p->y += dt * p->vy;
        p->z += dt * p->vz;
    }
}
```

其中最核心的算法是 `advance()` 函数实现的数值积分，它根据各个星球之间的距离和引力，算出加速度，再修正速度，然后更新星球的位置。这个 `naive` 算法的复杂度是 $O(N^2)$ 。

⁶⁶https://gist.github.com/1158889#file_nbody.c

C++ 数据抽象版，完整代码见 [67](#)，下面是代码骨架。

首先定义 `Vector3` 这个抽象，代表三维向量，它既可以是位置，有可以是速度。本处略去了 `Vector3` 的操作符重载，`Vector3` 支持常见的向量加减乘除运算。

然后定义 `Planet` 这个抽象，代表一个行星，它有两个 `Vector3` 成员：位置和速度。

需要说明的是，按照语义，`Vector3` 是数据抽象，而 `Planet` 是 object-based.

```
// C++ code
struct Vector3
{
    Vector3(double x, double y, double z)
        : x(x), y(y), z(z)
    {
    }

    double x;
    double y;
    double z;
};

struct Planet
{
    Planet(const Vector3& position, const Vector3& velocity, double mass)
        : position(position), velocity(velocity), mass(mass)
    {
    }

    Vector3 position;
    Vector3 velocity;
    const double mass;
};
```

相同功能的 `advance()` 代码简短得多，而且更容易验证其正确性。（想想如果把 C 语言版的 `advance()` 中的 `vx`、`vy`、`vz`、`dx`、`dy`、`dz` 写错位了，这种错误较难发现。）

```
// C++ code
void advance(int nbodies, Planet* bodies, double delta_time)
{
    for (Planet* p1 = bodies; p1 != bodies + nbodies; ++p1)
    {
        for (Planet* p2 = p1 + 1; p2 != bodies + nbodies; ++p2)
        {
            Vector3 difference = p1->position - p2->position;
```

⁶⁷https://gist.github.com/1158889#file_nbody.cc

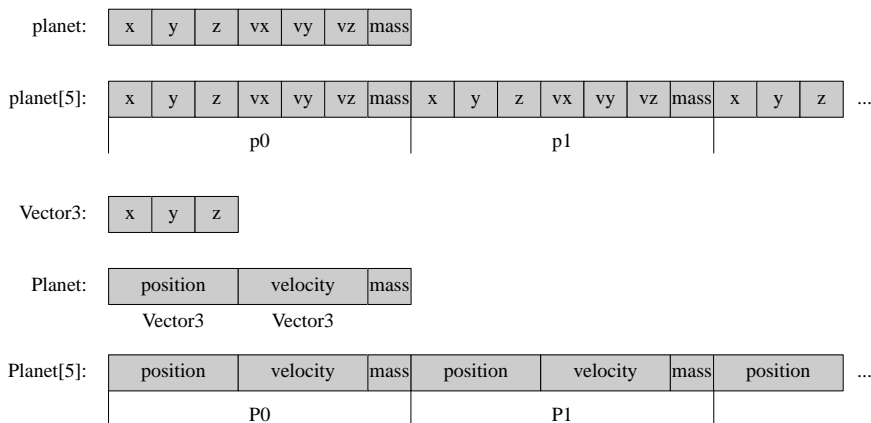

```

    double distance_squared = magnitude_squared(difference);
    double distance = std::sqrt(distance_squared);
    double magnitude = delta_time / (distance * distance_squared);
    p1->velocity -= difference * p2->mass * magnitude;
    p2->velocity += difference * p1->mass * magnitude;
}
}
for (Planet* p = bodies; p != bodies + nbodies; ++p)
{
    p->position += delta_time * p->velocity;
}
}

```

性能上，尽管 C++ 使用了更高层的抽象 `Vector3`，但它的性能和 C 语言一样快。看看 `memory layout` 就会明白：

C `struct` 的成员是连续存储的，`struct` 数组也是连续的。

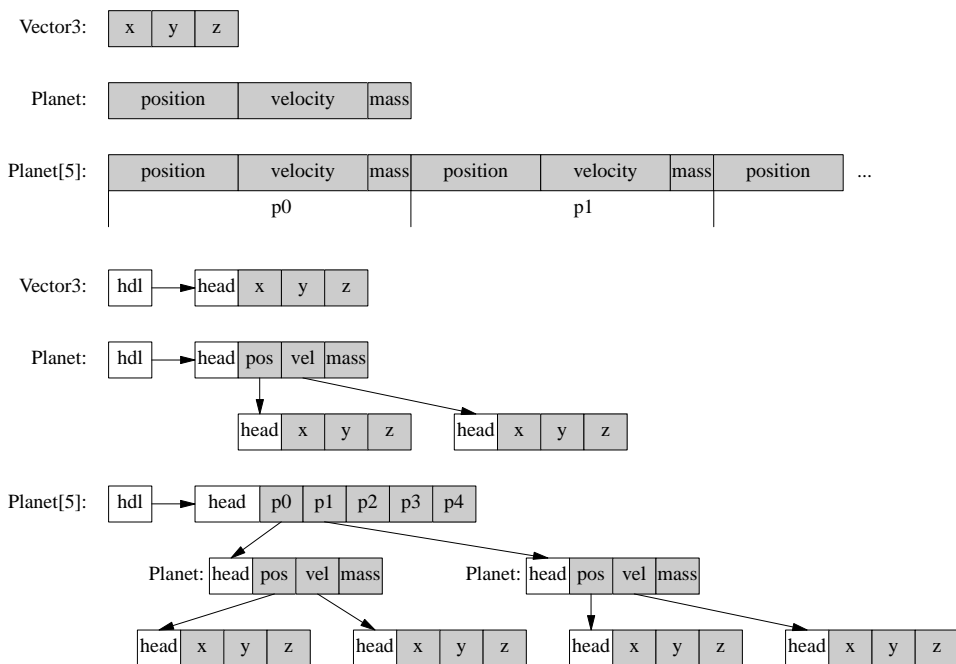


C++ 尽管定义了了 `Vector3` 这个抽象，它的内存布局并没有改变，`Planet` 的布局 和 C `planet` 一模一样，`Planet[]` 的布局也和 C 数组一样。

另一方面，C++ 的 `inline` 函数在这里也起了巨大作用，我们可以放心地调用 `Vector3::operator+=()` 等操作符，编译器会生成和 C 一样高效的代码。

不是每个编程语言都能做到在提升抽象的时候不影响性能，来看看 Java 的内存 布局。

如果我们用 `class Vector3`、`class Planet`、`Planet[]` 的方式写一个 Java 版的 N-body 程序，内存布局将会是：



这样大大降低了 **memory locality**，有兴趣的读者可以对比 **Java** 和 **C++** 的实现效率。

注：这里的 **N-body** 算法只为比较语言之间的性能与编程的便利性，真正科研中用到的 **N-body** 算法会使用更高级和底层的优化，复杂度是 $O(N \log N)$ ，在大规模模拟时其运行速度也比本 **naive** 算法快得多。

12.7.1 更多的例子

- **Date** 与 **Timestamp**，这两个 **class** 的“数据”都是整数，各定义了一套操作，用于表达日期与时间这两个概念。
- **BigInteger**，它本身就是一个“数”。如果用 **C++** 实现 **BigInteger**，那么阶乘函数写出来十分自然，下面第二个函数是 **Java** 语言的版本。

```
// C++ code
BigInteger factorial(int n)
{
    BigInteger result(1);
    for (int i = 1; i <= n; ++i) {
        result *= i;
    }
}
```

```
        return result;
    }

    // Java code
    public static BigInteger factorial(int n) {
        BigInteger result = BigInteger.ONE;
        for (int i = 1; i <= n; ++i) {
            result = result.multiply(BigInteger.valueOf(i));
        }
        return result;
    }
}
```

高精度运算库 `gmp` 有一套高质量的 C++ 封装⁶⁸

- 图形学中的三维齐次坐标 `Vector4` 和对应的 `4x4` 变换矩阵 `Matrix4`，例如⁶⁹
- 金融领域中经常成对出现的“买入价/卖出价”，可以封装为 `BidOffer struct`，这个 `struct` 的成员可以有 `mid()` “中间价”，`spread()` “买卖差价”，加减操作符，等等。

12.8 小结

数据抽象是 C++ 的重要抽象手段，适合封装“数据”，它的语义简单，容易使用。数据抽象能简化代码书写，减少偶然错误。

13 再探 `std::string`

Scott Meyers 在《Effective STL》[3] 第 15 条提到 `std::string` 有多种实现方式，归纳起来有三类，而每类又有多种变化。

1. 无特殊处理 (eager copy)，采用类似 `std::vector` 的数据结构。现在很少有实现采用这种方式。
2. Copy-on-Write (COW)。g++ 的 `std::string` 一直采用这种方式实现⁷⁰。
3. 短字符串优化 (SSO)，利用 `string` 对象本身的空间来存储短字符串。Visual C++ 用的是这种实现方式。

⁶⁸http://gmplib.org/manual/C_002b_002b-Interface-General.html#C_002b_002b-Interface-General

⁶⁹http://www.ogre3d.org/docs/api/html/classOgre_1_1Matrix4.html

⁷⁰`libstdc++` 的 `std::string` 是 Nathan Myers 的手笔。

下表总结了我知道的各个库的 string 实现方式和 string 对象分别在 32-bit/64-bit x86 系统中的大小。

库	32-bit	64-bit	实现方式
g++ std::string	4	8	COW
__gnu_cxx::__sso_string	24	32	SSO
__gnu_cxx::__rc_string	4	8	COW
clang libc++	12	24	SSO
SGI STL	12	24	eager copy
STLPort	24	48	SSO
Apache libstdcxx	4	8	COW
Visual C++ 2010	28/32	40/48	SSO

Visual C++ 的 std::string 的大小跟编译模式有关，表中小的那个数字是 release 编译，大的是 debug 编译。因此 debug 库和 release 库不能混用。除此之外，其他库的 string 大小是固定的。

以下分别介绍这几种实现方式的代码骨架和数据结构示意图，无论哪种实现方式都要保存三个数据：1. 字符串本身 (char*)，2. 字符串的长度 (size)，3. 字符串的容量 (capacity)。

13.1 直接拷贝 (eager copy)

类似 std::vector 的“三指针”结构。代码骨架 (省略模板):

_____ eager copy string 1

```
// http://www.sgi.com/tech/stl/string

// Class invariants:
// (1) [start, finish) is a valid range.
// (2) Each iterator in [start, finish) points to a valid object
//     of type value_type.
// (3) *finish is a valid object of type value_type; in particular,
//     it is value_type().
// (4) [finish + 1, end_of_storage) is a valid range.
// (5) Each iterator in [finish + 1, end_of_storage) points to
//     uninitialized memory.

// Note one important consequence: a string of length n must manage
// a block of memory whose size is at least n + 1.

class string
{
```

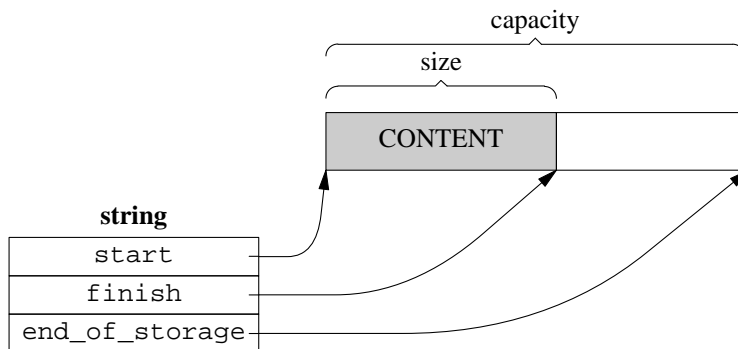
```

public:
    const_pointer data() const { return start; }
    iterator begin()          { return start; }
    iterator end()            { return finish; }
    size_type size() const    { return finish - start; }
    size_type capacity() const { return end_of_storage - start; }

private:
    char* start;
    char* finish;
    char* end_of_storage;
};

```

eager copy string 1



对象的大小是 3 个指针，在 32-bit 中是 12 字节，在 64-bit 中是 24 字节。

Eager copy string 的另一种实现方式是把后两个成员变量替换成整数，表示字符串的长度和容量，即：

```

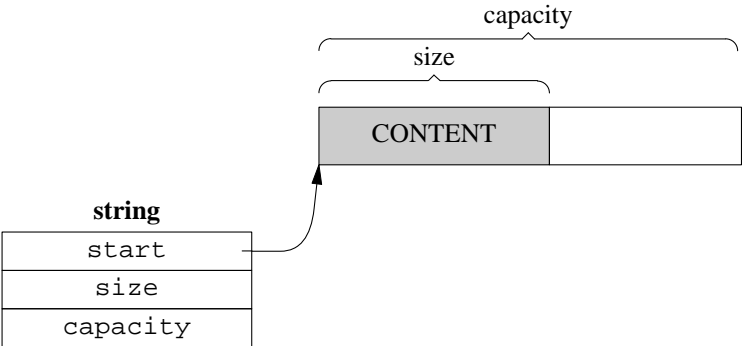
class string
{
public:
    const_pointer data() const { return start; }
    iterator begin()          { return start; }
    iterator end()            { return start + size_; }
    size_type size() const    { return size_; }
    size_type capacity() const { return capacity_; }

private:
    char* start;
    size_t size_;
    size_t capacity_;
};

```

eager copy string 2

eager copy string 2



这种做法并没有多大的改变，因为 `size_t` 和 `char*` 是一样大的。但是，我们通常用不到单个几百兆字节的字符串⁷¹，那么可以再改变一下长度和容量的类型（从 64-bit 整数改成 32-bit 整数），这样在 64-bit 下可以减小对象的大小。

class string

{

private:

char* start;

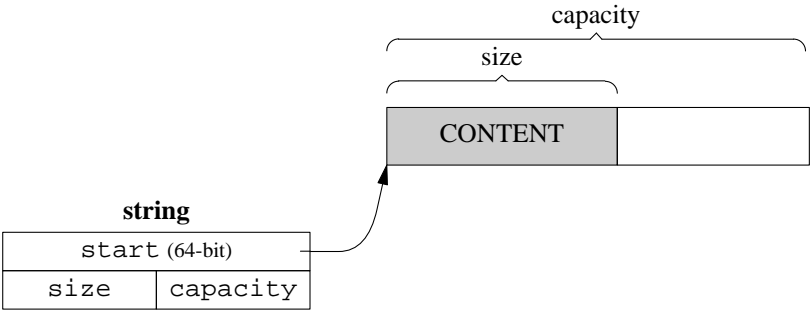
uint32_t size;

uint32_t capacity;

};

eager copy string 3

eager copy string 3



新的 `string` 结构在 64-bit 中是 16 字节，比原来的 24 字节小了一些。

13.2 Copy-on-Write

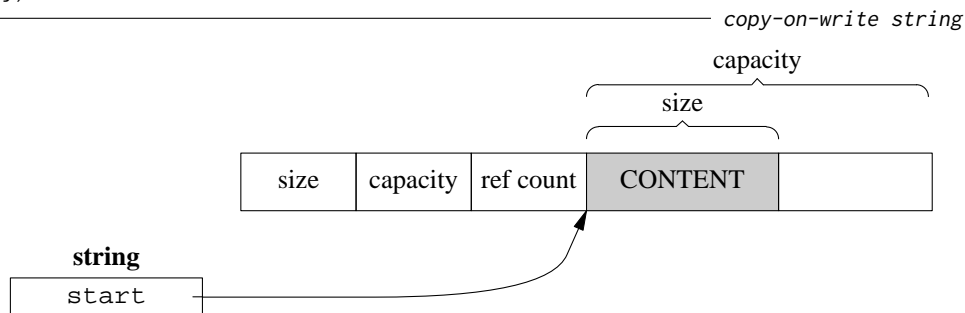
`string` 对象里只放一个指针。值得一提的是 COW 对多线程不友好，Andrei Alexandrescu 提倡在多核时代应该改用 `eager copy string`。

⁷¹如果真的用到了，就继续使用 `std::string` 或 `std::vector<char>` 好了。

```

class cow_string // libstdc++-v3
{
    struct Rep
    {
        size_t size;
        size_t capacity;
        size_t refcount;
        char* data[1]; // variable length
    };
    char* start;
};

```



这种数据结构没啥好说的，在 64-bit 中似乎也没有优化空间。另外 COW 的操作复杂度不一定符合直觉，它拷贝字符串是 $O(1)$ 时间，但是拷贝之后的第一次 `operator[]` 有可能是 $O(N)$ 时间。⁷²

13.3 短字符串优化 (SSO)

`string` 对象比前面两个都大，因为有本地缓冲区 (local buffer)。

```

class sso_string // __gnu_ext::__sso_string
{
    char* start;
    size_t size;

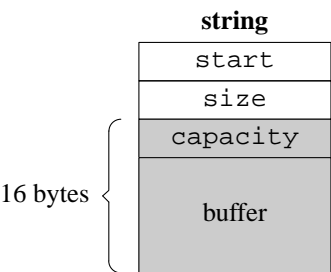
    static const int kLocalSize = 15;

    union
    {
        char buffer[kLocalSize+1];
        size_t capacity;
    } data;
};

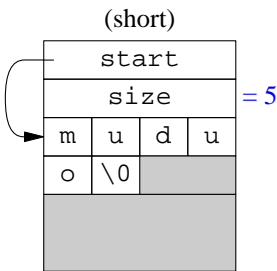
```

short-string-optimized string

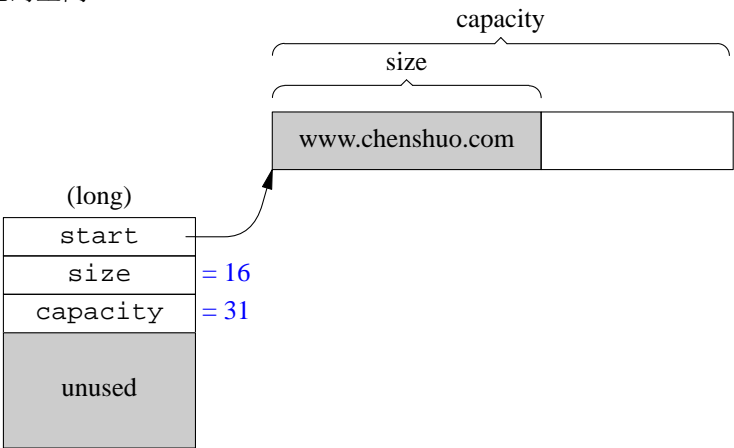
⁷²<http://coolshell.cn/articles/1443.html>



如果字符串比较短（通常的阈值是 15 字节），那么直接存放在对象的 `buffer` 里。
`start` 指向 `data.buffer`。



如果字符串超过 15 字节，那么就变成第118页的 `eager copy 2` 那种结构，`start` 指向堆上分配的空间。



短字符串优化的实现方式不止一种，主要区别是把那三个指针/整数中的哪一个与本地缓冲重合。例如《Effective STL》[3] 第 15 条展现的“实现 D”是将 `buffer` 与 `start` 指针重合，这正是 Visual C++ 的做法。而 STLPort 的 `string` 是将 `buffer` 与 `end_of_storage` 指针重合。

SSO string 在 64-bit 中有一个小小的优化空间：如果允许字符串 `max_size()` 不大于 4G 的话，我们可以用 32-bit 整数来表示长度和容量，这样同样是 32 字节的 `string` 对象，`local buffer` 可以增大至 19 字节。

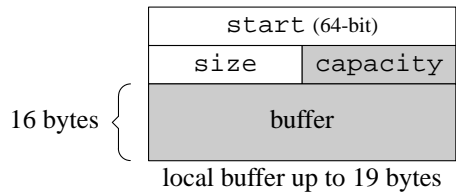
short-string-optimized string 2

```
class sso_string // optimized for 64-bit
{
    char* start;
    uint32_t size;

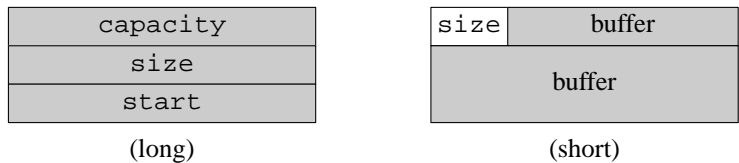
    static const int kLocalSize = sizeof(void*) == 8 ? 19 : 15;

    union
    {
        char buffer[kLocalSize+1];
        uint32_t capacity;
    } data;
};
```

short-string-optimized string 2

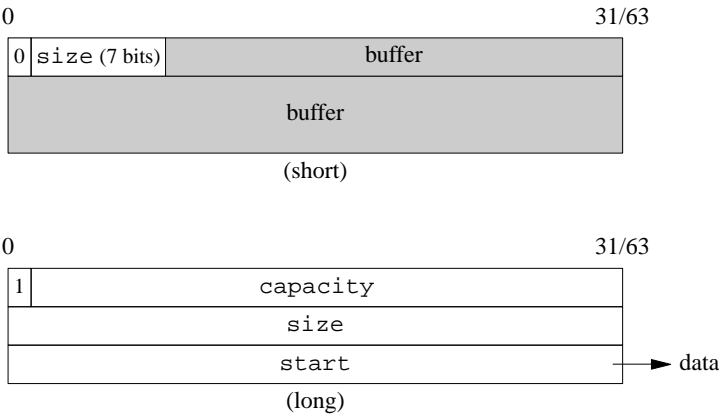


llvm/clang/libc++ 采用了与众不同的 SSO 实现，空间利用率最高，`local buffer` 几乎与三个指针/整数完全重合，在 64-bit 上对象大小是 24 字节，本地缓冲区可达 22 字节。



它用一个 `bit` 来区分是长字符还是短字符，然后用位操作和掩码 (`mask`) 来取重叠部分的数据，因此实现是 SSO 里最复杂的⁷³。

⁷³<http://llvm.org/viewvc/llvm-project/libcxx/trunk/include/string?view=markup>



Andrei Alexandrescu 建议^[8] 针对不同的应用负载选用不同的 `string`，对于短字符串，用 SSO `string`；对于中等长度的字符串，用 `eager copy`；对于长字符串，用 COW。具体分界点需要靠 `profiling` 来确定，选用合适的字符串可能提高 10% 的整体性能。

从实现的复杂度上看，`eager copy` 是最简单的，SSO 稍微复杂一些，COW 最难。性能也各有千秋，见 Petr Ovtchenkov 写的《Comparison of Strings Implementations in C++ language》⁷⁴。

我准备自己写一个 non-standard⁷⁵ non-template⁷⁶ 的 `string` 库⁷⁷作为练手，计划采用 `eager copy 3` 和 `sso 2` 的数据结构。

注脚：C++03/98 标准没有规定 `string` 中的字符是连续存储的，但是《Generic Programming and the STL》的作者 Matthew Austern 指出⁷⁸：现在所有的 `std::string` 实现都是连续存储的，因此建议在新标准中明确规定下来⁷⁹。

⁷⁴<http://complement.sourceforge.net/compare.pdf>
⁷⁵C++ 标准库 `string` 有很多设计缺陷，见 Herb Sutter 的《Exceptional C++ Style》^[5] 第 37~40 条。
⁷⁶云风：C++ 里 `string` 用 `template` 实现，纯粹是没事找事。同时区分宽字符和单字符极少在同一份代码里同时出现。结果把好好一个结实的库变成了一个源代码级的模板。另外见 Steve Donovan 写的《Overdoing C++ Templates》<http://blog.csdn.net/myan/article/details/1915>
⁷⁷<https://github.com/chenshuo/recipes/tree/master/string>
⁷⁸http://code.google.com/p/protobuf/source/browse/tags/2.4.1/src/google/protobuf/stubs/stl_util-inl.h#78
⁷⁹<http://www.open-std.org/JTC1/SC22/WG21/docs/lwg-defects.html#530>

14 用 STL algorithm 秒杀几道算法面试题

C++ STL 的 algorithm 配合自定义的 functor (仿函数、函数对象) 可以秒杀不少面试题, 代码简洁, 正确性也容易验证。本节仍旧采用 C++03 的 functor 写法, 没有采用 C++11 的 Lambda 表达式写法, 尽管后者会简洁得多。本节代码在 Debian Linux 6.0/g++ 4.4 下测试通过, 完整代码及测试用例见 ⁸⁰。

14.1 用 next_permutation() 生成排列与组合

本小节内容源自十年前我写的一篇博客⁸¹, 这篇博客还找到了 Visual C++ 7.0 的 STL 的一个疑似 bug (或者叫 feature)。生成排列、组合、整数划分的具体算法见 Donald Knuth 的《The Art of Computer Programming, Volume 4A》⁸²第 7.2.1 节。本处只给出使用 STL 的实现代码。

14.1.1 生成 N 个不同元素全排列

这是 next_permutation() 的基本用法, 把元素从小到大放好 (即字典序最小的排列), 然后反复调用 next_permutation() 就行了。

```

1  #include <algorithm>
2  #include <iostream>
3  #include <iterator>
4  #include <vector>
5
6  int main()
7  {
8      int elements[] = { 1, 2, 3, 4 };
9      const size_t N = sizeof(elements)/sizeof(elements[0]);
10     std::vector<int> vec(elements, elements + N);
11
12     int count = 0;
13     do
14     {
15         std::cout << ++count << ": ";
16         std::copy(vec.begin(), vec.end(),
17                 std::ostream_iterator<int>(std::cout, ", "));
18         std::cout << std::endl;
19     } while (next_permutation(vec.begin(), vec.end()));
20 }
```

recipes/algorithm/permutation.cc

⁸⁰<https://github.com/chenshuo/recipes/>

⁸¹<http://blog.csdn.net/solstice/article/details/2059>

⁸²<http://cs.utsa.edu/~wagner/knuth/>

整个程序最关键的就是第 19 行。输出的前几行是

```
1: 1, 2, 3, 4,
2: 1, 2, 4, 3,
3: 1, 3, 2, 4,
4: 1, 3, 4, 2,
5: 1, 4, 2, 3,
6: 1, 4, 3, 2,
7: 2, 1, 3, 4,
8: 2, 1, 4, 3,
9: 2, 3, 1, 4,
// 一共 24 行
```

类似的代码还能生成多重排列，比如 2 个 a、3 个 b 的全部排列，代码见 permutation2.cc。输出是

```
1: a, a, b, b, b,
2: a, b, a, b, b,
3: a, b, b, a, b,
4: a, b, b, b, a,
5: b, a, a, b, b,
6: b, a, b, a, b,
7: b, a, b, b, a,
8: b, b, a, a, b,
9: b, b, a, b, a,
10: b, b, b, a, a,
```

注: $\frac{5!}{2! \times 3!} = 10$

思考：能不能把 do {} while() 循环换成 while () {} 循环？

14.1.2 生成从 N 个元素中取出 M 个的所有组合

题目 输出从 7 个不同元素中取出 3 个元素的所有组合。

思路：对序列 { 1, 1, 1, 0, 0, 0 } 做全排列，对于每个排列，输出数字 1 对应的位置上的元素。代码如下

```

7  int main()
8  {
9      int values[] = { 1, 2, 3, 4, 5, 6, 7 };
10     int elements[] = { 1, 1, 1, 0, 0, 0, 0 };
11     const size_t N = sizeof(elements)/sizeof(elements[0]);
12     assert(N == sizeof(values)/sizeof(values[0]));
13     std::vector<int> selectors(elements, elements + N);
14

```

recipes/algorithm/combination.cc

```

15     int count = 0;
16     do
17     {
18         std::cout << ++count << ": ";
19         for (size_t i = 0; i < selectors.size(); ++i)
20         {
21             if (selectors[i])
22             {
23                 std::cout << values[i] << ", ";
24             }
25         }
26         std::cout << std::endl;
27     } while (prev_permutation(selectors.begin(), selectors.end()));
28 }

```

recipes/algorithm/combination.cc

注意，为了照顾输出顺序，第 27 行用的是 `prev_permutation()`。程序输出是

```

1: 1, 2, 3,
2: 1, 2, 4,
3: 1, 2, 5,
4: 1, 2, 6,
// 省略若干行
33: 4, 5, 7,
34: 4, 6, 7,
35: 5, 6, 7,

```

可见完整地输出了 $C(7, 4) = 35$ 种组合。

14.2 用 {make,push,pop}_heap() 实现多路归并

题目 用一台 4G 内存的机器对磁盘上的单个 100G 文件排序。

这种单机外部排序题目的标准思路是先分块排序，然后多路归并成输出文件。多路归并很容易用 `heap` 排序实现，比方说要归并已经按从小到大的顺序排好序的 32 个文件，我们可以构造一个 32 元素的 `min heap`，每个元素是 `std::pair<Record, FILE*>`。然后每次取出堆顶的元素，将其 `Record` 写入输出文件；如果 `FILE*` 还可读，就读入一条 `Record`，再向 `heap` 中添加 `std::pair<Record, FILE*>`。这样当 `heap` 为空的时候，多路归并就完成了。注意在这个过程中 `heap` 的大小通常会慢慢变小，因为有可能某个输入文件已经全部读完了。

这种方法比传统的二路归并要节省很多遍磁盘读写，假如用教科书上的二路归并来做外部排序⁸³，那么我们要先读一遍这 32 个文件，两两归并输出 16 个稍大的已排

⁸³这种教科书有可能是在大型机还在使用磁带外存的时候写成的。

序中间文件；然后再读一遍这 16 个中间文件，两两归并输出 8 个更大的中间文件；如此往复，最后归并两个已经排好序的大文件，输出最终的结果。读者可以算算这比直接多路归并要多读写多少遍磁盘。

完整的外部排序代码见 `recipes/esort/sort02.cc` 及其改进版 `sort{03,04}.cc`。这里展示一个内存里的多路归并，以说明基本思路。

```
File mergeN(const std::vector<File>& files)  recipes/algorithm/mergeN.cc
{
    File output;
    std::vector<Input> inputs;

    for (size_t i = 0; i < files.size(); ++i) {
        Input input(&files[i]);
        if (input.next()) {
            inputs.push_back(input);
        }
    }

    std::make_heap(inputs.begin(), inputs.end());
    while (!inputs.empty()) {
        std::pop_heap(inputs.begin(), inputs.end());
        output.push_back(inputs.back().value);

        if (inputs.back().next()) {
            std::push_heap(inputs.begin(), inputs.end());
        } else {
            inputs.pop_back();
        }
    }

    return output;
}
```

第 44~51 行构造一个 heap，第 52 行开始的 while 循环反复取出堆顶元素（第 53 行 `std::pop_heap()` 会把堆顶元素放到序列末尾，即 `inputs.back()` 处），第 54 行把取出的元素（当前最小值）输出。第 56~60 行从堆顶元素所属的文件读入下一条记录，如果成功，就把它放回堆中（第 57 行）。当循环结束的时候，堆为空，说明每个文件都读完了。

其中用到的 `Input` 类型定义如下。

```
typedef int Record;
typedef std::vector<Record> File;  recipes/algorithm/mergeN.cc
```

```
struct Input
{
    Record value;
    const File* file;

    explicit Input(const File* f);
    bool next();

    bool operator<(const Input& rhs) const
    {
        // make_heap to build min-heap, for merging
        return value > rhs.value;
    }
};
```

recipes/algorithm/mergeN.cc

类似的题目：有 a、b 两个文件，大小各是 100G 左右，每行长度不超过 1k，这两个文件有少量（几百个）重复的行，要求用一台 4G 内存的机器找出这些重复行。

解这道题目有两个方向，一是 hash，把 a、b 两个文件按行的 hash 取模分成几百个小文件，每个小文件都在 1G 以内，然后对 a1、b1 求交集 c1，对 a2、b2 求交集 c2，这样就能在内存里解决了。

第二个思路是外部排序，但是跟前面完整的外部排序不同，我们并不需要得到 a'、b' 两个已排序的文件再求交集，只需要把 a 分块排序成 100 个小文件，再把 b 分块排序成 100 个小文件，剩下的工作就是一边读这些小文件，一边在内存中同时归并出 a' 和 b'，一边求出交集。内存中的两个多路归并需要两个 heap，分别对应 a 和 b 的小文件 s。

代码写起来估计比单个 heap 归并要复杂一些，特别是 C++ 不支持类似 C# 的 yield 关键字来方便地实现迭代。假如 C++ 有 yield，那么“求交集”这一步我们直接调用 `std::set_intersection()` 并配合适当的迭代器就行了，但是在没有 yield 的情况下要实现这样的迭代器恐怕要费事得多，因为每个迭代器要维护更多的状态。这算是 coroutine 的一个使用场景。

14.3 用 unique() 去除连续重复空白

孟岩在谈《C++ 程序设计原理与实践》⁸⁴时曾说“比如对我来说，C++ 这个语言最强的地方在于它的模板技术提供了足够复杂的程序库开发机制，可以把复杂性高度

⁸⁴<http://blog.csdn.net/hzbooks/article/details/5767169>

集中在程序库里。做得好的话，在应用代码部分我连一个 for 循环都不用写，犯错误的机会就少，效率还不打折扣，关键是看着代码心里爽。”这几小节可算是他这番话的一个注脚。C++11 有了 Lambda 表达式，Scott Meyers 倡导的“Prefer algorithm calls to hand-written loops”就更容易落实了⁸⁵。

题目 给你一个字符串，要求原地 (in-place) 把相邻的多个空格替换为一个⁸⁶。例如，输入 "a_b"，输出 "a_b"；输入 "aaa_bbb_"，输出 "aaa_bbb_"。

这道题目不难，手写的话也就是单重循环，复杂度是 $O(N)$ 时间和 $O(1)$ 空间。这里展示用 `std::unique()` 的解法，思路很简单：`std::unique()` 的作用是去除相邻的重复元素，我们只要把“重复元素”定义为“两个元素都是空格”即可。注意所有针对区间的 STL algorithm 都只能调换区间内元素的顺序，不能真正删除容器内的元素，因此需要第 17 行。关键代码如下

```

5 struct AreBothSpaces
6 {
7     bool operator()(char x, char y) const
8     {
9         return x == ' ' && y == ' ';
10    }
11 };
12
13 void removeContinuousSpaces(std::string& str)
14 {
15     std::string::iterator last
16         = std::unique(str.begin(), str.end(), AreBothSpaces());
17     str.erase(last, str.end());
18 }

```

recipes/algorithm/removeContinuousSpaces.cc

14.4 用 partition() 实现“调整数组顺序使得奇数位于偶数前面”

`std::partition()` 的作用是把符合条件的元素放到区间首部，不符合条件的元素放到区间后部，我们只需把“符合条件”定义为“元素是奇数”就能解决这道题。复杂度是 $O(N)$ 时间 $O(1)$ 空间。为节省篇幅，`isOdd()` 直接做成了函数，而不是函数对象，缺点是有可能阻碍编译器做 inlining。

⁸⁵<http://drdobbs.com/184401446>

⁸⁶<https://gist.github.com/2227226>

```

5  bool isOdd(int x)
6  {
7      return x % 2 != 0; // x % 2 == 1 is WRONG
8  }
9
10 void moveOddsBeforeEvens()
11 {
12     int oddeven[] = { 1, 2, 3, 4, 5, 6 };
13     std::partition(oddeven, oddeven+6, &isOdd);
14     std::copy(oddeven, oddeven+6, std::ostream_iterator<int>(std::cout, ", "));
15     std::cout << std::endl;
16 }

```

recipes/algorithm/partition.cc

输出如下，注意确实满足“奇数位于偶数之前”，但奇数元素之间的相对位置有变化，偶数元素亦是如此。

1, 5, 3, 4, 2, 6,

如果题目要求改成“调整数组顺序使奇数位于偶数前面，并且保持奇数的先后顺序不变，偶数的先后顺序不变”，解决办法也一样简单，改用 `stable_partition()` 即可，代码及输出如下：

```

int oddeven[] = { 1, 2, 3, 4, 5, 6 };
std::stable_partition(oddeven, oddeven+6, &isOdd);
std::copy(oddeven, oddeven+6, std::ostream_iterator<int>(std::cout, ", "));
std::cout << std::endl;
// 输出 1, 3, 5, 2, 4, 6,

```

注意，`stable_partition()` 的复杂度较特殊：在内存充足的情况下，开辟原数组一样大的空间，复杂度是 $O(N)$ 时间和 $O(N)$ 空间；在内存不足的情况下，要做 in-place 位置调换，复杂度是 $O(N \log N)$ 时间和 $O(1)$ 空间。

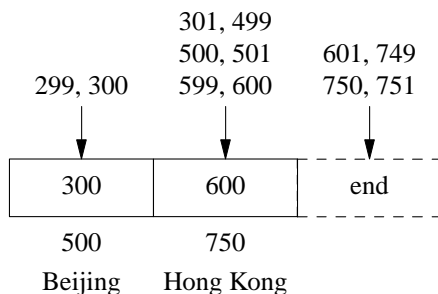
类似的题目还有“调整数组顺序使负数位于非负数前面”，读者自能举一反三。

14.5 用 `lower_bound()` 查找 IP 地址所属的城市

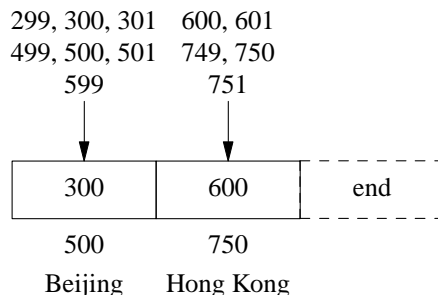
题目 已知 N 个 IP 地址区间和它们对应的城市名称，写一个程序，能从 IP 地址找到它所在的城市。注意这些 IP 地址区间互不重叠。

这道题目的 naïve 解法是 $O(N)$ ，借助 `std::lower_bound()` 可以轻易做到 $O(\log N)$ 查找，代价是事先做一遍 $O(N \log N)$ 的排序，如果区间相对固定而查找很频繁，这么做是值得的。

基本思路是按 IP 区间的首地址排好序，再进行二分查找。比如说有两个区间 [300, 500]、[600, 750]，分别对应北京和香港两个城市，那么 `std::lower_bound()` 查找 299, 300, 301, 499, 500, 501, 599, 600, 601, 749, 750, 751 等“IP 地址”返回的迭代器如下。



我们需要对返回的结果微调（第 28~32 行），使得迭代器 `it` 所指的区间是惟一有可能包含该 IP 地址的区间。



最后判断一下 IP 地址是否位于这个区间就行了（第 34 行）。完整代码如下，为了简化，“城市”用整数表示，-1 表示未找到。另外这个实现对于整个 IP 地址空间都是正确的，即便区间中包括 [255.255.255.0, 255.255.255.255] 这种边界条件。

```

7 struct IPRange
8 {
9     uint32_t startIp; // inclusive
10    uint32_t endIp;    // inclusive
11    int value;         // >= 0
12
13    bool operator<(const IPRange& rhs) const
14    {
15        return startIp < rhs.startIp;
16    }
17 };
18
19 // REQUIRE: ranges is sorted.

```

recipes/algorithm/iprange.cc

```
20 int findIpValue(const std::vector<IPrange>& ranges, uint32_t ip)
21 {
22     int result = -1;
23
24     if (!ranges.empty()) {
25         IPrange needle = { ip, 0, 0 };
26         std::vector<IPrange>::const_iterator it
27             = std::lower_bound(ranges.begin(), ranges.end(), needle);
28         if (it == ranges.end()) {
29             --it;
30         } else if (it != ranges.begin() && it->startIp > ip) {
31             --it;
32         }
33
34         if (it->startIp <= ip && it->endIp >= ip) {
35             result = it->value;
36         }
37     }
38
39     return result;
40 }
```

recipes/algorithm/iprange.cc

说明：如果 IP 地址区间有重复，那么我们通常要用线段树⁸⁷来实现高效的查询。另外，在真实的场景中，IP 地址区间通常适用专门的 **longest prefix match** 算法，这会比本节的通用算法更快。

14.6 小结

想到正确的思路是一码事，写出正确的经得起推敲的代码是另一码事。例如第 14.4 节用 $(x \% 2 != 0)$ 来判断 `int x` 是否为奇数，如果写成 $(x \% 2 == 1)$ 就是错的，因为 `x` 可能是负数，负数的取模运算的关窍见第 8 节。常见的错误还包括误用 `char` 的值作为数组下标（题目：统计文件中每个字符出现的次数），但是没有考虑 `char` 可能是负数，造成访问越界。有的人考虑到了 `char` 可能是负数，因此先强制转型为 `unsigned int` 再用作下标，这仍然是错的。正确的做法是强制转型为 `unsigned char` 再用作下标，这涉及到 C/C++ 整形提升的规则，就不详述了。这些细节往往是面试官的考察点⁸⁸。本节给出的解法在正确性方面应该是没问题的，效率方面，可以说在 Big-O 意义下是最优的，但不一定是运行最快的。

另外，面试题的目的可能就是让你动手实现一些 STL 算法，例如求两个有序集合的交集 (`set_intersection`)、洗牌 (`random_shuffle`) 等等，这就不属于本文所讨论的

⁸⁷http://en.wikipedia.org/wiki/Segment_tree

⁸⁸工作 5 年以来，我面试过近百人，因此这番话是从面试官的角度说的。

范围了。从“算法”本身的难度上看，我个人把 STL algorithm 分为三类，面试时要求手写的往往是第二类算法。

- 容易，即闭着眼睛一想就知道是如何实现的，自己手写一遍的难度跟 `strlen()` 和 `strcpy()` 差不多。这类算法基本上就是遍历一遍输入区间，对每个元素做些判断或操作，一个 `for` 循环就解决战斗。一半左右的 STL algorithm 属于此类，例如 `for_each()`、`transform()`、`accumulate()` 等等。
- 较难，知道思路，但是要写出正确的实现要考虑清楚各种边界条件。例如 `merge()`、`unique()`、`remove()`、`random_shuffle()`、`partition()`、`lower_bound()` 等等，三成左右的 STL algorithm 属于此类。
- 难，要在一个小时内写出正确的健壮的实现基本不现实，例如 `sort()`⁸⁹、`nth_element()`、`next_permutation()`、`inplace_merge()` 等等，约有两成 STL algorithm 属于此类。

注意，“容易”级别的算法是指写出正确的实现很容易，不一定意味着写出高效的实现也同样容易，例如 `std::copy()` 拷贝 POD 类型的效率可媲美 `memcpy()`，这需要用一点模板技巧。

以上分类纯属个人主观看法，或许换个人来分类结果就大不一样，例如把 `remove()` 归入简单，把 `next_permutation()` 归入较难，把 `lower_bound()` 归入难等等。

15 C++ 编译链接模型精要

C++ 从 C 语言⁹⁰ 继承了一种古老的编译模型，引发了其他语言中根本不存在的的一些编译方面的问题（比方说“一次定义原则 (ODR)⁹¹”）。理解这些问题有助于在实际开发中规避各种古怪的错误。

C++ 语言的三大约束是：与 C 兼容、零开销 (zero overhead) 原则、值语义。值语义的话题前文第 12 节已经讲过，这里谈谈第一点“与 C 兼容”。

⁸⁹快速排序是本科生物数据结构课上就有的内容，但是工业强度的实现是足以在顶级期刊上发论文的。

⁹⁰本文谈的 C 语言和 C++ 语言指的是现代的常见的实现（没有特别指明时，可认为是 Linux x86-64 的 GCC），并不限于 C 标准或 C++ 标准，因为标准里根本就没有提到“程序库/library”这个概念。另外本文所提的 C 语言库函数不仅包括 C 标准中的函数，也包括 POSIX 里的常用函数，因为在 Linux 下二者是不分家的，都位于 `libc.so`。

⁹¹http://en.wikipedia.org/wiki/One_Definition_Rule

“与 C 兼容”的含义很丰富，不仅仅是兼容 C 的语法⁹²，更重要的是兼容 C 语言的编译模型与运行模型，也就是说能直接使用 C 语言的头文件和库。比方说对于 `connect(2)` 这个系统函数，它的头文件和原型如下。

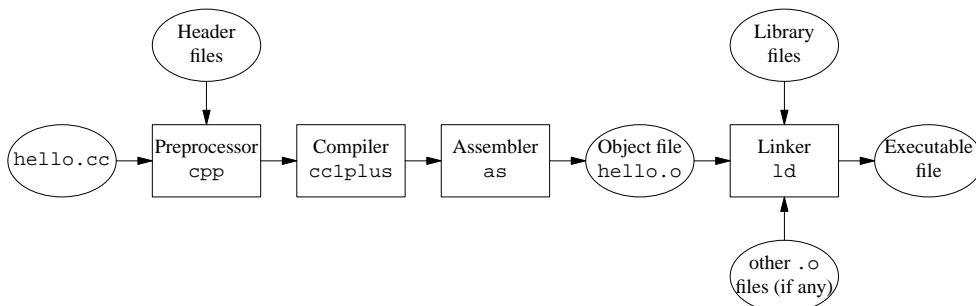
```
#include <sys/socket.h>
```

```
int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

C++ 的基本类型的长度和表示 (representation) 必须和 C 语言⁹³ 一样 (int、指针等等)，C++ 编译器必须能理解头文件 `sys/socket.h` 中 `struct sockaddr` 的定义，生成与 C 编译器完全相同的 layout (包括采用相同的对齐 (alignment) 算法)，并且遵循 C 语言的函数调用约定 (参数传递，返回值传递，栈帧管理等等)，才能直接调用这个 C 语言库函数。

由于现代操作系统暴露出的原生接口往往是 C 语言描述的⁹⁴。C++ 兼容 C，从而能在编译的时候直接使用这些头文件，并链接到相应的库上。并在运行的时候直接调用 C 语言的函数库，这省了一道中间层的手续，可算作是 C++ 高效的原因之一。

下图表明了 Linux 上编译一个 C++ 程序的典型过程。其中最耗时间的是 `cc1plus` 这一步，在一台正在编译 C++ 项目的机器上运行 `top(1)`，排在首位的往往就是这个进程。



值得指出的是，上图中各个阶段的界线并不是铁定的。通常 `cpp` 和 `cc1plus` 会合并成一个进程；而 `cc1plus` 和 `as` 之间既可以以临时文件 (*.s) 为中介，也可以以管道 (pipe) 为中介；对于单一源文件的小程序，往往不必生成 .o 文件。另外，linker 还有一个名字叫做 `link editor`。

⁹²从兼容语法的角度，Java 和 C# 都可以算是“与 C 兼容”，例如它们的 for 循环写出来都是：

```
for (int i = 0; i < 100; ++i) { /* do something */ }
```

⁹³准确地说是和编译系统库的 C 语言编译器保持一致。

⁹⁴Windows 的原生 API 接口是 `Windows.h` 头文件，POSIX 是一堆 C 语言头文件，本文不区分系统调用与用户态库函数，统称为“系统函数”。

在不同的语境下，“编译”一词有不同的含义。如果笼统地说把 .cc 文件“编译”为可执行文件，那么指的是 `preprocessor/compiler/assembler/linker` 这四个步骤。如果区分“编译”和“链接”，那么“编译”通常指的是从源文件生成目标文件这几步（即 `g++ -c`）。如果进一步区分预处理、编译（代码转换）、汇编，那么编译器实际看到的是预处理器完成头文件替换和宏展开之后的源代码⁹⁵。

C++ 至今（C++11）没有模块机制，不能像其他现代编程语言那样用 `import` 或 `using` 来引入当前源文件用到的库（含其他 `package/module` 里的函数或类），而必须用 `include` 头文件的方式来机械地将库的接口声明以文本替换的方式载入，再重新 `parse` 一遍。这么做一方面让编译效率奇低，编译器动辄要 `parse` 几万行预处理之后的源码，哪怕源文件只有几百行；另一方面，也留下了巨大的隐患。部分原因是头文件包含具有传递性，引入不必要的依赖；另一个原因是头文件是在编译时使用，动态库文件是在运行时使用，二者的时间差可能带来不匹配，导致二进制兼容性方面的问题（见第 4 节）。C++ 的设计者 Bjarne Stroustrup 自己很清楚这一点⁹⁶，但这是在“与 C 兼容”的大前提下不得不做出的妥协。

比如有一个简单的小程序，只用了 `printf(3)`，却不得不包含 `stdio.h`，把其他不相关的函数、`struct` 定义、宏、`typedef`、全局变量等等也统统引入到当前命名空间。在预处理的时候会读取近 20 个头文件，预处理之后供编译器 `parse` 的源码有近千行，这还算是短的。

```
$ cat hello.cc          # 一个简单的源文件
#include <stdio.h>      # 只包含了一个头文件

int main()
{
    printf("hello preprocessor\n");
}

$ gcc -E hello.cc |wc # 预处理之后有 942 行
  942   2164   17304

$ strace -f -e open cpp hello.cc -o /dev/null 2>&1 |grep -v ENOENT|awk '{print $3}'
# 省略无关内容。另外我不知道 cpp 有没有直接输出以下内容的命令行选项，只好用笨办法。
open("hello.cc",
open("/usr/include/stdio.h",
open("/usr/include/features.h",
open("/usr/include/bits/predefs.h",
open("/usr/include/sys/cdefs.h",
```

⁹⁵前面提到，现代编译器通常把预处理和代码转换合并起来，从而让编译器获得更多的信息，调试信息也更丰富。现在的编译器能获知包括宏常量的名字、宏函数等传统上编译器看不到的内容，有的开发环境甚至能单步跟踪宏函数。

⁹⁶《C++ 语言的设计和演化》第 18 章“C 语言预处理器”——C++ 必须被摧毁。

```

open("/usr/include/bits/wordsize.h",
open("/usr/include/gnu/stubs.h",
open("/usr/include/bits/wordsize.h",
open("/usr/include/gnu/stubs-64.h",
open("/usr/lib/gcc/x86_64-linux-gnu/4.4.5/include/stddef.h",
open("/usr/include/bits/types.h",
open("/usr/include/bits/wordsize.h",
open("/usr/include/bits/typesizes.h",
open("/usr/include/libio.h",
open("/usr/include/_G_config.h",
open("/usr/lib/gcc/x86_64-linux-gnu/4.4.5/include/stddef.h",
open("/usr/include/wchar.h",
open("/usr/lib/gcc/x86_64-linux-gnu/4.4.5/include/stdarg.h",
open("/usr/include/bits/stdio_lim.h",
open("/usr/include/bits/sys_errlist.h",

```

读者有兴趣的话可以把其中的 `stdio.h` 替换为 C++ 标准库的头文件 `complex`，看看预处理之后的源代码有多少行，额外包含了哪些头文件。（在我的机器上测试，预处理之后有 21879 行，包含了近 150 个头文件，包括 `string`、`sstream` 等大块头。）

值得一提的是，为了兼容 C 语言，C++ 付出了很大的代价。例如要兼容 C 语言的隐式类型转换规则（例如整数类型提升），在让 C++ 的函数重载决议 (`overload resolution`) 规则无比复杂⁹⁷。另外 `class` 定义式后面那个分号也不晓得谋杀了多少初学者的时间⁹⁸。Bjarne Stroustrup 自己也说“我又不是不懂如何设计出比 C++ 更漂亮的语言⁹⁹。”（由于 C 语言没函数重载，也就不存在重载决议，所以隐式类型转换的危害没有体现在这一方面。）

15.1 C 语言的编译模型及其成因

要想了解 C 语言的编译模型的成因，我们需要略微回顾一下 Unix 的早期历史¹⁰⁰ 1969 年 Ken Thompson 用汇编语言在一台闲置的 PDP-7 上写出了 Unix 的史前版本¹⁰¹。值得一提的是，PDP-7 的字长是 18-bit¹⁰²，只能按字 (word) 寻址，不支持今日常见的按 8-bit 字节寻址。假如 C 语言诞生在 PDP-7 上，计算机软硬件的发展史恐怕要改写。

⁹⁷C++ 复杂的作用域机制也大大增加了函数重载决议的难度，基本上只有 C++ 编译器才弄得清楚。

http://en.wikipedia.org/wiki/C%2B%2B#Parsing_and_processing_C.2B.2B_source_code

⁹⁸这是为了与 C `struct` 语法兼容，因为 C 允许在函数返回类型处定义新 `struct` 类型，因此分号是必需的。

⁹⁹*Even I knew how to design a prettier language than C++.* — Bjarne Stroustrup

¹⁰⁰The Evolution of the Unix Time-sharing System. Dennis M. Ritchie.

<http://cm.bell-labs.com/cm/cs/who/dmr/hist.pdf>

¹⁰¹Unix 历史一般从 1970 年算起 (Unix Epoch 是 1970-01-01 00:00:00 UTC)，因此这个只能算“史前”。

¹⁰²<http://en.wikipedia.org/wiki/PDP-7>

1970 年 5 月, Ken Thompson 和 Dennis Ritchie 所在的贝尔实验室下订单购买了一台 PDP-11 小型机¹⁰³, 这是 1970 年 1 月刚刚上市的新机型。PDP-11 的字长是 16-bit, 可以按 8-bit 字节寻址, 这可谓一举奠定了今后 C 语言及硬件的发展道路¹⁰⁴。这台机器的主机 (处理器和内存) 当年夏天就到货了, 但是硬盘直到 1970 年 12 月才到货。

1971 年, Ken Thompson 把原来运行在 PDP-7 上的 Unix 用 PDP-11 汇编人肉重写了一遍, 运行在这台 PDP-11/20 机器上。这台机器一共只有 24kB 内存¹⁰⁵, 其中 16kB 运行操作系统, 8kB 运行用户代码¹⁰⁶; 硬盘一共只有 512kB, 文件大小限制为 64kB。然后实现了一个文本处理器, 用于排版贝尔实验室的专利申请, 这是购买这台计算机的正经用途。

下面的 Unix 历史多半发生在另外一台内存和硬盘都更大的 PDP-11 机器上, 型号可能是 PDP-11/40 或 PDP-11/45。(不同的权威文献说法不一, 可能不止一台。)

1972 年是 C 语言历史上最为关键的一年¹⁰⁷, 这一年 C 语言加入了预处理, 具备了编写大型程序的能力 (理由见下文)。到了 1973 年初, C 语言基本定型, 主要新特性是支持结构体。此时 C 语言的编译模型已经基本定型, 即分为预处理、编译、汇编、链接这四个步骤, 沿用至今。

1973 年是 Unix 历史上关键的一年, 这一年夏天, 二神把 Unix 的内核用 C 语言重写了一遍, 完成了用高级语言编写操作系统的伟大创举。(Thompson 在 1972 年就尝试用过 C 重写 Unix 内核, 但是当时的 C 语言不支持结构体, 因此他放弃了。)

随后, 1974 年, Dennis Ritchie 和 Ken Thompson 发表了经典论文《The UNIX Time-Sharing System》¹⁰⁸。除了没有函数原型声明外, 1974 年的 C 代码¹⁰⁹读起来跟现在的 C 程序基本无区别。

这里的 Unix 早期版本历史还参考了 <http://minnie.tuhs.org/cgi-bin/utree.pl>

¹⁰³<http://en.wikipedia.org/wiki/PDP-11>

¹⁰⁴在 C 语言 70 年代开始流行之后, 高效支持 C 语言就成了 CPU 指令集的设计目标之一, 否则这种 CPU 很难推广。另外, C/Unix/Arpanet 还规范了字节的长度, 在此之前, 字节可以是 6、7、8、9、12 比特, 之后都是 8-bit, 否则就不能与其他系统联网通信。 <http://herbsutter.com/2011/10/12/dennis-ritchie/>

¹⁰⁵用的是磁芯存储器 http://en.wikipedia.org/wiki/Magnetic-core_memory, 因此早期文献常以 core 指代内存。

¹⁰⁶PDP-11/20 是 PDP-11 系列的第一个型号, 甚至没有内存保护机制, 也就没法区分核心态和用户态。

¹⁰⁷The Development of the C Language. Dennis M. Ritchie.

<http://cm.bell-labs.com/cm/cs/who/dmr/chist.pdf>

¹⁰⁸<http://cm.bell-labs.com/cm/cs/who/dmr/cacm.pdf> 这篇文章至少有三个版本, 第一版以单页摘要的形式发表于 1973 年 10 月的第四届 ACM SOSP 会议上, 第二版发表于 1974 年 7 月的 CACM 期刊上, 第三版发表于 1978 年七-八月的 BSTJ 上。此处连接是第三版, 内容与 CACM 的原始版本略有出入。

¹⁰⁹Unix V5 的 C 编译器源码: <http://minnie.tuhs.org/cgi-bin/utree.pl?file=V5/usr/c>

15.1.1 为什么 C 语言需要预处理？

了解了 C 语言的诞生背景，我们可以归纳 PDP-11 上的第一代 C 编译器的硬性约束：内存地址空间只有 16-bit，程序和数据必须挤在这狭小的 64kB 空间里，可谓捉襟见肘¹¹⁰。注意，本节提到的 C 语言甚至早于 1978 年的 K&R C，是 1970 年代最初几年的原始 C 语言。

编译器没办法在内存里完整地表示单个源文件的抽象语法树¹¹¹，更不可能把整个程序（由多个源文件组成）放到内存里，以完成交叉引用（不同源文件的函数之间相互调用，使用外部变量等等）。由于内存限制，编译器必须要能分别编译多个源文件，生成多个目标文件，再设法把这些目标文件组合（链接¹¹²）为一个可执行文件。

在今天看来，C 语言这种支持把一个程序分成多个源文件的“功能”几乎是顺理成章的。但是在当时而言，并不是每个语言都有意做到这一点。我们以同一时期（1968 ~ 1974）Niklaus Wirth 设计的 Pascal 语言为对照。Pascal 语言可以定义函数和结构体，也支持指针，语法也比当时的 C 语言更优美。但是它长期没有官方规定¹¹³的多源文件模块化机制，它要求每个程序 (program) 必须位于同一个源文件¹¹⁴，这其实大大限制了它在系统编程方面的用途¹¹⁵。如果 Pascal 一早就克服这些缺点¹¹⁶，“那么我们今天很可能要把 `begin` 和 `end` 直接映射到键盘上。¹¹⁷”

或许是受内存限制，一个可执行程序不能太大，Dennis Ritchie 编写的 PDP-11 C 编译器不是一个可执行文件，而是 7 个可执行文件¹¹⁸：`cc`、`cpp`、`as`、`ld`、`c0`、`c1`、`c2`¹¹⁹。其中 `cc` 是个 `driver`，用于调用另外几个程序。`cpp` 是预处理器（Unix V7 从 `c0` 分离出来），当时叫做 `compiler control line expander`。`c0`、`c1`、`c2` 是 C 编译器的三个阶段 (phase)¹²⁰，`c0` 的作用是把源程序编译为两个中间文件，`c1` 把中间文件编

¹¹⁰PDP-11 的物理内存可以有几百 kB，但是每个进程只能看到 16-bit 的地址空间。PDP-11/45 支持将代码空间和数据空间分离（即哈佛结构，而非冯诺依曼结构），各自有 64kB。但是直到 1979 年的 Unix V7 才用上这个功能，而此时 C 语言早已定型。http://en.wikipedia.org/wiki/PDP-11_architecture

¹¹¹我怀疑当时的 C 编译器恐怕连整个函数都无法放到内存里，只能放下当前的表达式。

¹¹²其实链接器的历史比编译器还长，在没有高级语言编译器只有汇编器的时代，链接器就已经存在。我们可以把多个汇编源文件 `assemble` 成目标文件，再让链接器来处理外部符号的地址与函数重定位。

¹¹³PASCAL - User Manual and Report. Springer-Verlag, 1974.

¹¹⁴Donald Knuth 写的 `TEX` 就是一个 20000 多行的单源文件 Pascal 大程序。

¹¹⁵Niklaus Wirth 最初的设计目的是让 Pascal 成为结构化编程的教学语言。

¹¹⁶Why Pascal is Not My Favorite Programming Language. Brian W. Kernighan.

<http://www.lysator.liu.se/c/bwk-on-pascal.html>

¹¹⁷孟岩《C++ 开源程序库评话》<http://blog.csdn.net/myan/article/details/679007>

¹¹⁸Regenerating System Software http://minnie.tuhs.org/PUPS/Setup/v7_regen.html

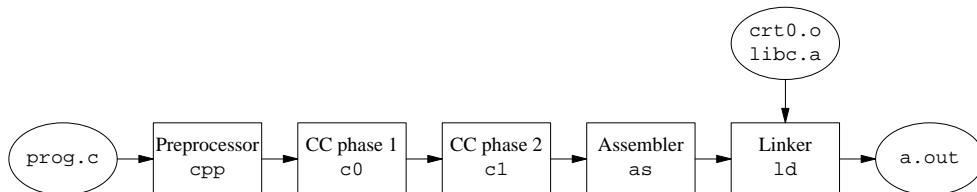
¹¹⁹在 Unix V5 中 `c[012]` 的源代码一共有 6100 行，在 Unix V6 中一共有 8000 行。

¹²⁰A Tour through the UNIX C Compiler. Dennis M. Ritchie.

<http://plan9.bell-labs.com/v7vol2b.pdf>

译为汇编源代码，c2 是可选的，用于对生成汇编代码做窥孔优化。as 是汇编器，把汇编代码转换为目标文件。ld 是链接器，把目标文件和库文件链接成可执行文件。不用 cc，手工编译一个简单程序 prog.c 的过程是：

```
/lib/cpp prog.c > prog.i      # prog.i 是预处理之后的源代码
/lib/c0 prog.i temp1 temp2    # c0 生成 temp1 和 temp2 这两个中间文件
/lib/c1 temp1 temp2 prog.s    # c1 读入 temp1 和 temp2，生成汇编代码 prog.s
as - prog.s                  # 把 prog.s 汇编为目标文件 a.out。猜猜 a.out 的原意？
ld -n /lib/crt0.o a.out -lc   # 把 a.out 链接为可执行文件
当时的链接器是单向查找未决符号，因此要把 crt0.o 放到 a.out 之前，-lc 必须放到末尾
```



为了能在尽量减少内存使用的情况下实现分离编译，C 语言采用了“隐式函数声明 (implicit declaration of function)”的做法。代码在使用前文未定义的函数时，编译器不需要也不检查函数原型¹²¹。既不检查参数个数，也不检查参数类型与返回值类型。编译器认为未声明的函数都返回 int，并且能接受任意个数的 int 型参数。而且早期的 C 语言甚至不严格区分指针和 int，而是认为二者可以相互赋值转换。在 C++ 程序员看来，这是毫无安全保障的做法，但是 C 语言就是如此地相信程序员。

举例解释一下什么是“隐式函数声明”。

```
$ cat hello.c
int main()                # 这个程序没有引用任何头文件
{
    printf("hello C.\n"); # 隐式声明 int printf(...);
    return 0;
}

$ gcc hello.c -Wall        # 用 gcc 可以编译运行通过
hello.c: In function 'main':
hello.c:3: warning: implicit declaration of function 'printf'
hello.c:3: warning: incompatible implicit declaration of built-in function 'printf'

$ g++ hello.c -Wall        # 用 g++ 则会报错
hello.c: In function 'int main()':
hello.c:3: error: 'printf' was not declared in this scope
```

如果 C 程序用到了某个没有定义的函数（可能错误拼写了函数名），那么实际造成的是链接错误 (undefined reference)，而非编译错误。例如

¹²¹C 语言的函数原型是 1980 年代才从 C++ 借用过来的，算是 C++ 对 C 的反哺。

```
$ cat undefined.c
int main()
{
    helloworld();          # 隐式声明 helloworld
    return 0;
}

$ gcc undefined.c -Wall
undefined.c: In function 'main':
undefined.c:3: warning: implicit declaration of function 'helloworld'
/tmp/ccHUCGat.o: In function 'main':
undefined.c:(.text+0xa): undefined reference to 'helloworld'
collect2: ld returned 1 exit status    # 真正报错的是 ld, 不是 cc1
```

其实，有了隐式函数声明，我们已经能分别编译多个源文件，然后把它们链接为一个大的可执行文件（此处指的是编译出来有几十 kB 的程序）。那么为什么还需要头文件和预处理呢？

根据 Eric S. Raymond 在《The Art of Unix Programming》第 17.1.1 节¹²²引用 Steve Johnson 的话，最早的 Unix 是把内核数据结构（例如 struct dirent）打印在手册上，然后每个程序自己在代码中定义 struct。例如 Unix V5 的 ls(1) 源码¹²³中就自行定义了表示目录的结构体。有了预处理和头文件，这些公共信息就可以做成头文件放到 /usr/include，然后程序包含用到的头文件即可。减少无谓错误，提高代码的可移植性。

最早的预处理只有两项功能：#include 和 #define。#include 完成文件内容替换，#define 只支持定义宏常量，不支持定义宏函数。早期的头文件里只放三样东西：struct 定义，外部变量¹²⁴的声明，宏常量。这样可以减少各个源文件里的重复代码。

到目前为止，头文件的预处理的作用都还是正面的。在谈头文件与预处理的害处之前，让我把 PDP-11 的 16-bit 地址空间对 C 语言及其编译模型的影响讲完。

15.1.2 C 语言的编译模型

由于不能将整个源文件的语法树保存在内存中，C 语言其实是按“单遍编译 (one pass)¹²⁵”来设计的。所谓单遍编译，指的是从头到尾扫描一遍源码，一边解析 (parse) 代码，一边即刻生成目标代码。在单遍编译时，编译器只能看到目前（当前语句/符号之前）已经 parse 过的代码，看不到之后的代码，而且过眼即忘。意味着

¹²²http://www.faqs.org/docs/artu/c_evolution.html

¹²³<http://minnie.tuhs.org/cgi-bin/utree.pl?file=V5/usr/source/s1/lsc> 注意 readdir() 函数

¹²⁴或者叫全局变量，如果不那么学究的话。

¹²⁵http://en.wikipedia.org/wiki/One-pass_compiler

- C 语言要求结构体必须先定义，才能访问其成员，否则编译器不知道结构体成员的类型和偏移量，就无法立刻生成目标代码。
- 局部变量也必须先定义再使用，因为如果把定义放到后面，编译器在第一次看到一个局部变量时并不知道它的类型和在 `stack` 中的位置，也就无法立刻生成代码，只能报错退出。
- 为了方便编译器分配 `stack` 空间，C 语言要求局部变量只能在语句块的开始处定义。
- 对于外部变量，编译器只需要知道它的类型和名字，不需要知道它的地址，因此需要先声明后使用。在生成的目标代码中，外部变量的地址是个空白，留给链接器去填上。
- 当编译器看到一个函数调用，按隐式函数声明规则，编译器可以立刻生成调用函数的汇编代码（函数参数入栈、调用、获取返回值），这里惟一尚不能确定的是函数的实际地址，编译器可以留下一个空白给链接器去填。

对 C 编译器来说，只需要记住 `struct` 的成员和偏移，知道外部变量的类型，就足以一边解析源代码，一边生成目标代码。因此早期的头文件和预处理恰好满足了编译器的需求。外部符号（函数或变量）的决议 (**resolution**) 可以留给链接器去做¹²⁶。

从上面的编译过程可以发现，C 编译器可以做得很小，只使用很少的内存。据我观察，Unix V5 的 C 编译器甚至没有使用动态分配内存，而是用一些全局的栈和数组来帮助处理复杂表达式和语句嵌套，整个编译器的内存消耗是固定的¹²⁷。（我推测 C 语言不支持在函数内部嵌套定义函数也是受此影响，因为这样一来意味着要必须用递归才能解析函数体，编译器的内存消耗就不是一个定值。）

受“不能嵌套”的影响，整个 C 语言的命名空间是平坦的 (**flat**)，函数和 `struct` 都处于全局命名空间。这其实给 C 程序员带来不少麻烦，因为每个库都要设法避免自己的函数和 `struct` 与其他库冲突。早期 C 语言甚至不允许在不同 `struct` 中使用相同的成员名称¹²⁸，因此我们看到一些 `struct` 的名字有前缀，例如 `struct timeval` 的成员是 `tv_sec` 和 `tv_usec`，`struct sockaddr_in` 的成员是 `sin_family`、`sin_port`、`sin_addr`。

讲清楚了 C 语言的编译模型，我们再来看看它对 C++ 的影响（和伤害）。

¹²⁶链接器的主要作用之一其实就是填空，见 [6] 和 [9] 等书有关章节。

¹²⁷这意味着 Unix V5 的 C 编译器不能处理太复杂的表达式，编译器也确实有对“Expression overflow”的错误处理。

¹²⁸即 `struct` 的成员名称是全局的。其实不是不允许，而是相同名字的成员的类型和其在各自 `struct` 内的偏移必须也相同。

15.2 C++ 的编译模型

由于要保持与 C 兼容，原本很多在 C 语言中顺理成章或者危害不大的东西继承到了 C++ 里就成了大祸害¹²⁹。

15.2.1 单遍编译

C++ 也继承了单遍编译。在单遍编译时，编译器只能根据目前看到的代码做出决策，读到后面的代码也不会影响前面做出的决定。这特别影响了名字查找 (name lookup) 和函数重载决议。

先说名字查找，C++ 中的名字包括：类型名、函数名、变量名、typedef 名、template 名等等。比方说对下面这行代码

```
Foo<T> a;    # Foo、T、a 这三个名字都不是 macro
```

如果不知道 Foo、T、a 这三个名字分别代表什么，编译器无法进行语法分析。根据之前出现的代码不同，上面这行语句至少有三种可能性：

1. Foo 是个 `template<typename X> class Foo;`，T 是 type，那么这句话以 T 为模板类型参数类型具现化了 `Foo<T>` 类型，并定义了变量 a。
2. Foo 是个 `template<int X> class Foo;`，T 是 `const int`，那么这句话以 T 为非类型模板参数具现化了 `Foo<T>` 类型，并定义了变量 a。
3. Foo、T、a 都是 `int`，这句话是个没啥用的表达式语句。

别忘了 `operator<` 是可以重载的，这又大大增加了这句简单代码的可能性^{130 131}。另外一个经典的例子是 `AA BB(CC);`，这句话既可以声明函数，也可以定义变量。

C++ 只能通过解析源码来了解名字的含义，不能像其他语言那样通过直接读取目标代码中的元数据来获得所需信息（函数原型、class 类型定义等等）。这意味着要想准确理解一行 C++ 代码的含义，我们需要通读这行代码之前的所有代码，并理解每个符号（包括操作符）的定义。而头文件的存在使得肉眼观察几乎是不可能的。完全有可能出现一种情况：某人不经意改变了头文件，或者仅仅是改变了源文件中头文件的包含顺序，就改变了代码的含义，破坏了代码的功能。这时能造成编译错误已经是谢天谢地了。

¹²⁹前面已经讲过隐式类型转换对函数重载决议的影响。

¹³⁰有兴趣的话可以读一读陈皓写的《恐怖的 C++ 语言》一文 <http://coolshell.cn/articles/1724.html>

¹³¹更多的例子见 [11] 的 6.3.1 节。

C++ 编译器的符号表至少要保存目前已看到的每个名字的含义，包括 `class` 的成员定义、已声明的变量、已知的函数原型等等¹³²，才能正确解析源代码。编译器还要正确处理作用域嵌套引发的名字的含义变化：内层作用域中的名字有可能遮住 (shadow) 外层作用域中的名字。有些其他语言会对此发出警告，对此我建议用 `g++` 的 `-Wshadow` 选项来编译代码。（插一句题外话：`muduo` 的代码都是 `-Wall -Wextra -Werror -Wconversion -Wshadow` 编译的。）

再说**函数重载决议**，当 C++ 编译器读到一个函数调用语句时，它必须（也只能）从目前已看到的同名函数中选出最佳函数。哪怕后面的代码中出现了更合适的匹配，也不能影响当前的决定¹³³。这意味着如果我们交换两个 `namespace` 级的函数定义在源代码中的位置，那么有可能改变程序的行为。

比方说有如下一段代码：

```
void foo(int)
{
    printf("foo(int);\n");
}

void bar()
{
    foo('a'); // 调用 foo(int)
}

void foo(char)
{
    printf("foo(char);\n");
}

int main()
{
    bar();
}
```

如果有人在重构的时候把 `void bar()` 的定义挪到 `void foo(char)` 之后，程序的输出就不一样了。

这个例子充分说明实现 C++ 重构工具的难度：重构器对代码的理解必须达到编译器的水准，才能在修改代码时不改变原意。函数的参数可以是复杂表达式，重构器必须能正确解析表达式的类型才能完成重载决议。比方说 `foo(str[0])` 应该调用哪个

¹³²这还没有考虑 `template`，编译 `template` 的难度超乎想象。

¹³³对于 `class` 成员函数有一个例外，编译器总是先扫描一遍 `class` 定义，再来处理其中的成员函数，因此全部同名成员函数都参与重载决议。这条规则还有一个例外的例外：对于 `class` 里的 `typedef` 不成立，`typedef` 的名字只在定义点之后可见。

foo() 跟 str[0] 的类型有关, 而 str 可能是个 std::string, 这就要求重构器能正确理解 template 并具现化之。C++ 至今没有像样的重构工具, 恐怕正是这个原因。

C++ 编译器必须在内存中保存函数级的语法树, 才能正确实施返回值优化 (RVO)¹³⁴, 否则遇到 return 语句的时候编译器无法判断被返回的这个对象是不是那个可以被优化的 named object¹³⁵。

其实由于 C++ 新增了不少语言特性, C++ 编译器并不能真正做到像 C 那样过眼即忘的单遍编译。但是 C++ 必须兼容 C 的语意, 因此编译器不得不装得好像是单遍编译 (准确说是单遍 parse) 一样, 哪怕它内部是 multiple pass 的¹³⁶。

15.2.2 前向声明

几乎每份 C++ 编码规范^{137 138 139}都会建议尽量使用前向声明来减少编译期依赖, 这里我用“单向编译”来解释一下为什么是可行的, 很多时候甚至是必须的。

如果代码里调用了函数 foo(), C++ 编译器 parse 此处函数调用时, 需要生成函数调用的目标代码。为了完成语法检查并生成调用函数的目标代码, 编译器需要知道函数的参数个数和类型以及函数的返回值类型, 它并不需要知道函数体的实现 (除非要做 inline 展开)。因此我们通常把函数原型放到头文件里, 这样每个包含了此头文件的源文件都可以使用这个函数。这是每个 C/C++ 程序员都明白的事情。

当然, 光有函数原型是不够的, 程序其中某一个源文件应该定义这个函数, 否则会造成链接错误 (未定义的符号)。这个定义 foo() 函数的源文件通常也会包含 foo() 的头文件。但是, 假设在定义 foo() 函数时把参数类型写错了, 会出现什么情况?

```
// in foo.h
void foo(int);           // 原型声明

// in foo.cc
#include "foo.h"

void foo(int, bool)      // 在定义的时候必须把参数列表和返回类型抄一遍。
{                          // 有抄错的可能, 也可能将来改了一处忘了改另一处
```

¹³⁴http://en.wikipedia.org/wiki/Return_value_optimization

¹³⁵Visual C++ 直到 2005 年才实现 RVO。

[http://msdn.microsoft.com/en-us/library/ms364057\(VS.80\).aspx](http://msdn.microsoft.com/en-us/library/ms364057(VS.80).aspx)

¹³⁶C++ 允许 forward reference, 因此几乎肯定做不到 one pass。

http://en.wikipedia.org/wiki/Forward_declaration

¹³⁷[2] 第 31 条, [4] 第 22 条。

¹³⁸LLVM 编程规范 http://llvm.org/docs/CodingStandards.html#hl_dontinclude

¹³⁹http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml#Header_File_Dependencies


```
// do something  
}
```

编译 `foo.cc` 会有错吗？不会，因为编译器会认为 `foo` 有两个重载。但是链接整个程序会报错：找不到 `void foo(int)` 的定义。你有没有遇到过类似的问题？

这是 C++ 的一种典型缺陷，即一样东西区分声明和定义，代码放到不同的文件中，这就会出现不一致的可能性。C/C++ 里很多稀奇古怪的错误就源自于此，比如 [9] 举的一个经典例子：在一个源文件里声明 `extern char* name`，在另一个源文件里却定义成 `char name[] = "Shuo Chen";`。

对于函数的原型声明和函数体定义而言，这种不一致表现在参数列表和返回类型上，编译器通常能查出参数列表不同，但不一定能查出返回类型不同，见后文第 148 页。也可能参数类型相同，但是顺序调换。例如原型声明为 `draw(int height, int width)`，定义的时候写成 `draw(int width, int height)`，编译器无法查出此类错误，因为原型声明中的变量名是无用的。

其他语言似乎没有这个问题。例如我们不需要在 Java 里使用函数原型声明，一个成员函数的参数列表只需要在代码里出现一次，不存在不一致的可能。Java 编译器也不受“单遍编译”的约束，调整成员函数的顺序不会影响代码语义。Java 也没有笨重过时的头文件包含机制，而是有一套基于 `package` 的模块化机制，陷阱少得多。

如果要写一个库给别人用，那么通常要把接口函数的原型声明放到头文件里。但是在写库的内部实现的时候，如果没有出现函数相互调用¹⁴⁰的情况，那么我们可以适当组织函数定义的顺序，让基础函数出现在代码的前面，这样就不必前向声明函数原型了。见云风的一篇博客¹⁴¹。

函数原型声明可以看作是对函数的前向声明 (forward declaration)，除此之外我们还常常用到 `class` 的前向声明。

有些时候 `class` 的前向声明是必须的，例如第 99 页出现的 `Child` 和 `Parent class` 相互指涉的情况¹⁴²。有些时候 `class` 的完整定义是必须的^[4, item 22]，例如要访问 `class` 的成员，或者要知道 `class` 的大小以便分配空间。其他时候，有 `class` 的前向声明就足够了，编译器只需要知道有这么个名字的 `class`。

对于 `class Foo`，以下几种使用不需要看见其完整定义：

¹⁴⁰http://en.wikipedia.org/wiki/Mutual_recursion

¹⁴¹<http://blog.codingnow.com/2007/06/kiss.html>

¹⁴²同一页出现的 Java 代码也没有前向声明，说明 Java 编译器能同时看到多个源文件的代码。

- 定义或声明 `Foo*` 和 `Foo&`，包括用于函数参数、返回类型、局部变量、类成员变量等等。这是因为 C++ 的内存模型是 flat 的，`Foo` 的定义无法改变 `Foo` 的指针或引用的含义。
- 声明一个以 `Foo` 为参数或返回类型的函数，如 `Foo bar()` 或 `void bar(Foo f)`，但是，如果代码里调用这个函数就需要知道 `Foo` 的定义，因为编译器要使用 `Foo` 的拷贝构造函数和析构函数，因此至少要看到它们的声明（虽然构造函数没有参数，但是有可能位于 `private` 区）。

`muduo` 代码中大量使用前向声明来减少 `include`，并且避免把内部 `class` 的定义暴露给用户代码。

[4] 第 30 条规定不能重载 `&&`、`||`、`,` (逗号) 这三个操作符，Google 的 C++ 编程规范补充规定¹⁴³ 不能重载一元 `operator&`（取址操作符），因为一旦重载 `operator&`，这个 `class` 的就不能用前向声明了。例如

```
class Foo; // 前向声明

void bar(Foo& foo)
{
    Foo* p = &foo; // 这句话是取 foo 的地址，但是如果重载了 &，意思就变了。
}
```

代码的行为跟是否 `include Foo` 的完整定义有关，等于埋了定时炸弹。

15.3 C++ 链接 (linking)

链接 (linking) 这个话题可以单独写一本书 [6]，这本书讲“C++ 链接”的有第 4.4 节“静态链接/C++ 相关问题”和第 9.4 节“C++ 与动态链接¹⁴⁴”等章节。

本节重点介绍与 C++ 日常开发相关的链接方面的问题，先以手工编制一本书的目录和交叉索引为例，介绍链接器的基本工作原理^{145 146 147}。假设一个作者写完了十多个章节，你的任务是把这些章节编辑为一本书。每个章节的篇幅不等，从 30 页到 80 页都有，都已经分别排好版打印出来。（已经从源文件编译成了目标文件。）

¹⁴³http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml#Operator_Overloading

¹⁴⁴不过我对用 C++ 编写动态链接库有自己的看法，见本文第 5、6 两节。

¹⁴⁵[9] 第 5 章，[10] 第 7 章，[6] 第 4 章。

¹⁴⁶Linkers and Loaders. <http://www.linuxjournal.com/article/6463>

¹⁴⁷https://events.linuxfoundation.org/images/stories/pdf/lfcs2012_ccoutant.pdf

章节之间有交叉引用，即正文里会出现“请参考 XXX 页的第 YYY 节”等字样。作者在撰写每个章节的时候并不知道当前文字的章节号，当然也不知道当前文字将来会在出现哪一页上。因为他可以随时调整章节顺序、增减文字内容，这些举动会影响最终的章节编号和页码。为了引用其他章节的内容，作者会在文字中放 `anchor` (\LaTeX 是 `\label`)，给需要被引用的文字命名。比方说本节“C++ 编译连接模型精要”的名字是 `sec.cppCompilation`。（这就好比给全局函数或全局变量起个独一无二的名字。）在引用其他章节的编号或页码时，作者在正文中留下一个适当的空白，并注明这里应该填上的某个 `anchor` 的页码或章节编号 (\LaTeX 是 `\ref{sec.cppCompilation}`)。

现在你拿到了这十几摞打印的文稿，怎么把它们编辑成一本书呢？你可能会想到下面这两个步骤：先编排页码和章节编号，再解决交叉引用。第一步：

1a 把这些文稿按章的先后顺序叠好，这样就可以统一编制页码。

1b 在编制页码的同时，章节号也可以一并确定下来。

在进行 1a 和 1b 这个步骤时，你可以同时顺序记录两张纸：

- 章节的编号、标题和它出现的页码，用于编制目录。
- 遇到 `anchor` 时，记下它的名字和出现的页码、章节号，用于解决交叉引用。

如果按上面的办法来操作，解决交叉引用就不难了。第二步：

- 2 再从头翻一遍书稿，遇到空白的交叉应用，就到 `anchor` 索引表里查出它的页码和章节编号，填上空白。

至此，如果一切顺利的话，书籍编辑任务完成。

在这项工作中最容易出现两种意外情况：

- 正文中交叉应用找不到对应的 `anchor`，空白填不上咋办？
- 某个 `anchor` 多次定义，该选哪一个填到交叉引用的空白处呢？

这正是最常见的两种链接错误。

上面描述的办法要至少翻两遍全文，有没有办法从头到尾只翻一遍书就完成交叉引用呢？如果作者在写书的时候只从前面的章节引用后面的章节，那么是可以做到

的。我们在编排页码和章节号的时候顺便阅读全文，遇到新的交叉引用空白就记到一张之上。这张纸记录交叉引用的名字和空白出现的页码。我们知道后面肯定能遇到对应的 **anchor**。在遇到一个 **anchor** 时，去那张纸上看看有没有交叉引用用到它，如果有，就回翻到空白的页码，把空白填上，回头再继续编制页码和章节号。这样一遍扫下来，章节编号、页码、交叉引用全部搞定了。

这正是传统 **one-pass** 链接器的工作方式，在使用这种链接器的时候要注意参数顺序，越基础的库越放到后面。如果程序用到了多个 **library**，这些 **library** 之间有依赖（假设不存在循环依赖），那么链接器的参数顺序应该是依赖图的拓扑排序。这样保证每个未决符号都可以在后面出现的库中找到。比如 **A**、**B** 两个彼此独立的库同时依赖 **C** 库，那么链接的顺序是 **ABC** 或 **BAC**。

为什么这个规定不是反过来，先列出基础库，再列出应用库呢？原因是前一种做法的内存消耗要小得多。如果先处理基础库，链接器不知道库里哪些符号会被后面代码的用到，因此只能每一个都记住，链接器的内存消耗跟所有库的大小之和成正比。反过来，如果先处理应用库，那么只需要记住目前尚未查到定义的符号就行了。链接器的内存消耗跟程序中外部符号的多少成正比（而且一旦填上空白，就可以忘掉它）。

以上简要介绍了 C 语言的链接模型，C++ 与之相比主要增加了两项内容：

- 函数重载，需要类型安全的链接^[11, 第 11.3 节]，即 **name mangling**¹⁴⁸
- **vague linkage**^{149 150}，即同一个符号有多份互不冲突的定义。

name mangling 的事情一般不需要程序员操心，只要掌握 **extern "C"** 的用法，能和 C 程序库 **interoperate** 就行。何况现在一般的 C 语言库的头文件都会适当使用 **extern "C"**，使之也能用于 C++ 程序。

C 语言通常一个符号在程序中只能有一处定义，否则就会造成重复定义。C++ 则不同，编译器在处理单个源文件的时候并不知道某些符号是否应该在本编译单元定义。为了保险起见¹⁵¹，只能每个目标文件生成一份“弱定义”，而依赖链接器去选择一份作为最终的定义，这就是 **vague linkage**。为了让这种机制能正确运作，C++ 要求代码满足一次定义原则 (ODR)，否则代码的行为是随机的，视 **linker** 心情好坏而定。

以下分别简要谈谈这两方面对编程的影响。

¹⁴⁸http://en.wikipedia.org/wiki/Name_mangling

¹⁴⁹<http://gcc.gnu.org/onlinedocs/gcc/Vague-Linkage.html>

¹⁵⁰<http://sourcery.mentor.com/public/cxx-abi/abi.html#vague>

¹⁵¹不这么做的话就会出现未定义的符号错误，因为链接器通常不会聪明到反过来调用编译器去生成未定义的符号。

15.3.1 函数重载

我们知道，为了实现函数重载，C++ 编译器普遍采用名字改编 (name mangling)¹⁵² 的办法，为每个重载函数生成独一无二的名字，这样在链接的时候就能找到正确的重载版本。比如 `foo.cc` 里定义了两个 `foo()` 重载函数。

```
$ cat foo.cc
int foo(bool x)
{
    return 42;
}

int foo(int x)
{
    return 100;
}

$ g++ -c foo.cc
$ nm foo.o                # foo.o 定义了两个 external linkage 函数
0000000000000000 T _Z3foob
0000000000000010 T _Z3fooi

$ c++filt _Z3foob _Z3fooi  # unmangle 这两个函数名
foo(bool)                 # 注意, mangled name 里没有返回类型
foo(int)
```

注意普通 `non-template` 函数的 `mangled name` 不包含返回类型。记得吗，返回类型不参与函数重载。

这其实有一个小小的隐患，也是“C++ 典型缺陷”的一个体现。如果一个源文件用到了重载函数，但它看到的函数原型声明的返回类型是错的（违反了 ODR），链接器无法捕捉这样的错误。

```
$ cat main.cc
void foo(bool);           # 返回类型错误地写成了 void

int main()
{
    foo(true);
}

$ g++ -c main.cc
$ nm main.o               # 目标文件依赖 _Z3foob 这个符号
U _Z3foob
0000000000000000 T main

$ g++ main.o foo.o        # 能正常生成 ./a.out
```

¹⁵²此处以 `g++` 为例，规则见 <http://sourcery.mentor.com/public/cxx-abi/abi.html#mangling>

对于内置类型，这应该不会造成实际的影响。但是如果返回类型是 `class`，那么就天晓得会发生什么了。

15.3.2 inline 函数

`inline` 函数的方方面面见 [2] 第 30 条。

由于 `inline` 函数的关系，C++ 源代码里调用一个函数并不意味着生成的目标代码里也会做一次真正的函数调用 (可能看不到 `call` 指令)。现在的编译器聪明到可以自动判断一个函数是否适合 `inline`，因此 `inline` 关键字在源文件中往往不是必须的。当然，在头文件里 `inline` 还是要的，为了防止链接器抱怨重复定义 (`multiple definition`)。现在的 C++ 编译器采用重复代码消除¹⁵³的办法来避免重复定义。也就是说，如果编译器无法 `inline` 展开的话，每个编译单元都会生成 `inline` 函数的目标代码，然后链接器会从多份实现中任选一份保留，其余的则丢弃 (`vague linkage`)。如果编译器能够展开 `inline` 函数，那就不必单独为之生成目标代码了 (除非使用函数指针指向它)。

如何判断一个 C++ 可执行文件是 `debug build` 还是 `release build`? 换言之，如何判断一个可执行文件是 `-O0` 编译还是 `-O2` 编译? 我通常的做法是看 `class template` 的短成员函数有没有被 `inline` 展开。例如

```
$ cat vec.cc
#include <vector>
#include <stdio.h>

int main()
{
    std::vector<int> vi;
    printf("%zd\n", vi.size()); #
}

$ g++ -Wall vec.cc      # non-optimized build
$ nm ./a.out |grep size|c++filt
00000000004007ac W std::vector<int, std::allocator<int> >::size() const
// vector<int>::size() 没有 inline 展开，目标文件中出现了函数 (弱) 定义。

$ g++ -Wall -O2 vec.cc # optimized build
$ nm ./a.out |grep size|c++filt
// 没有输出，因为 vector<int>::size() 被 inline 展开了。
```

注意，编译器为我们自动生成的 `class` 析构函数也是 `inline` 函数，有时候我们要故意 `out-line`，防止代码膨胀或出现编译错误。以下 `Printer` 是依据第 6 节介绍的

¹⁵³ 见 [6] 第 4.4.1 节。

pimpl 手法实现的公开 class。这个 class 的头文件完全没有暴露 Impl class 的任何细节，只用到了前向声明。并且有意地把构造函数和析构函数也显式声明了。

```
#include <boost/scoped_ptr.hpp> printer.h

class Printer // : boost::noncopyable
{
public:
    Printer();
    ~Printer(); // make it out-line
    // other member functions

private:
    class Impl; // forward declaration only
    boost::scoped_ptr<Impl> impl_;
};
```

printer.h

在源文件中，我们可以从容地先定义 Printer::Impl，然后再定义 Printer 的构造函数和析构函数。

```
#include "printer.h" printer.cc

class Printer::Impl
{
    // members
};

Printer::Printer()
    : impl_(new Impl) // 现在编译器看到了 Impl 的定义，这句话能编译通过。
{
}

Printer::~~Printer() // 尽管析构函数是空的，也必须放到这里来定义。
{
    // 否则编译器在将隐式声明的 ~Printer() inline 展开
    // 的时候无法看到 Impl::~Impl() 的声明，会报错。
}
```

printer.cc

在现代的 C++ 系统中，编译和链接的界限更加模糊了。传统 C++ 教材告诉我们，要想编译器能够 inline 一个函数，那么这个函数体必须在当前编译单元可见。因此我们通常把公共 inline 函数放到头文件中。现在有了 link time code generation¹⁵⁴，编译器不需要看到 inline 函数的定义，inline 展开可以留给链接器去做。

¹⁵⁴2010 年发布的 gcc4.5 开始支持 -fllto 选项，落后 Visual C++ .NET 好多年。

除了 `inline` 函数，g++ 还有大量的内置函数 (built-in function) ¹⁵⁵，因此源代码中出现 `memcpy`、`memset`、`strlen`、`sin`、`exp` 之类的“函数调用”不一定真的会调用 `libc` 里的库函数。另外，由于编译器知道这些函数的功能，因此优化起来更充分。

有意思的是，编译器如何处理 `inline` 函数中的 `static` 变量？这个留给读者去探究吧。

15.3.3 template

C++ 模板包括函数模板和类模板，在涉及与链接相关的话题时，我们一般会考察：

- 函数定义，包括具现化后的函数模板、类模板的成员函数、类模板的静态成员函数等等。
- 变量定义，包括函数模板的静态数据变量、类模板的静态数据成员、类模板的全局对象等等。

模板编译链接的不同之处在于，以上具有 `external linkage` 的对象通常会在多个编译单元被定义。链接器必须进行重复代码消除 ¹⁵⁶，才能正确生成可执行文件。

`template` 和 `inline` 函数会不会导致代码膨胀？

假设有一个定长 `Buffer` 类，其内置 `buffer` 长度是在编译期确定的，我们可以把它实现为非类型类模板：

```
template<int Size>
class Buffer
{
public:
    Buffer() : index_(0) {}

    void append(const void* data, int len)
    {
        ::memcpy(buffer_+index_, data, len);
        index_ += len;
    }

    void clear() { index_ = 0; }
    // other members
```

¹⁵⁵<http://gcc.gnu.org/onlinedocs/gcc-4.4.4/gcc/Other-Builtins.html>

¹⁵⁶见 [6] 第 4.4.1 节。

```
private:
    char buffer_[Size]; // Size 是模板参数
    int index_;
};
```

在代码中使用了 `Buffer<256>` 和 `Buffer<1024>` 两份具体体:

```
int main()
{
    Buffer<256> b1;
    b1.append("hello", 5); // Buffer<256>::append()
    b1.clear();           // Buffer<256>::clear()

    Buffer<1024> b2;
    b2.append("template", 8); // Buffer<1024>::append()
    b2.clear();             // Buffer<1024>::clear()
}
```

按照 C++ 模板的具现化规则, 编译器会为每一个用到的类模板函数具现化一份实体。

```
$ g++ buffer.cc
$ nm a.out
00400748 W _ZN6BufferILi1024EE5clearEv      # Buffer<1024>::clear()
004006f2 W _ZN6BufferILi1024EE6appendEPKvi  # Buffer<1024>::append(void const*, int)
004006da W _ZN6BufferILi1024EEC1Ev          # Buffer<1024>::Buffer()
004006c2 W _ZN6BufferILi256EE5clearEv      # Buffer<256>::clear()
0040066c W _ZN6BufferILi256EE6appendEPKvi  # Buffer<256>::append(void const*, int)
00400654 W _ZN6BufferILi256EEC1Ev          # Buffer<256>::Buffer()
```

这样看来真的造成了代码膨胀, 但实际情况并不一定如此, 如果我们用 `-O2` 编译一下, 会发现编译器把这些短函数都 `inline` 展开了。

```
$ g++ -O2 buffer.cc
$ nm a.out |c++filt |grep Buffer
# 没有输出, Buffer 的成员函数都被 inline 展开了, 没有生成函数定义。
```

如果我们想限制模板的具现化, 比方说限制 `Buffer` 只能有 64、256、1024、4096 这几个长度, 除了可以用 `static_assert` 来制造编译期错误, 还可以用下面这个只声明不定义的办法来制造链接错误。

一般的 C++ 教材会告诉你, 模板的定义要放到头文件中, 否则会有编译错误。如果读者足够细心, 会发现其实所谓的“编译错误”是链接错误。例如

```
$ cat main.cc
template<typename T>
```



```

void foo(const T&);    # 声明而没有定义

template<typename T>
T bar(const T&);

int main()
{
    foo(0);
    foo(1.0);
    bar('c');
}

$ g++ main.cc          # 注意是链接器报错，不是编译器报错
/tmp/cc5SKd58.o: In function 'main':
main.cc:(.text+0x17): undefined reference to 'void foo<int>(int const&)'
main.cc:(.text+0x31): undefined reference to 'void foo<double>(double const&)'
main.cc:(.text+0x41): undefined reference to 'char bar<char>(char const&)'
collect2: ld returned 1 exit status

```

那么有办法把模板的实现放到库里，头文件里只放声明吗？其实是可以的，前提是你知道模板会有哪些具现化类型，并事先显式（或隐式）具现化出来。

```

$ g++ -c main.cc      # 可以单独编译为目标文件
$ nm main.o           # 目标文件里引用了未定义的模板函数，
                     U _Z3barIcET_RKS0_      # char bar<char>(char const&)
                     U _Z3fooIdEvRKT_        # void foo<double>(double const&)
                     U _Z3fooIiEvRKT_        # void foo<int>(int const&)
0000000000000000 T main      # 注意这次函数 mangled name 包含返回类型

$ cat foobar.cc
template<typename T>
void foo(const T&)
{
}

template<typename T>
T bar(const T& x)
{
    return x;
}

template void foo(const int&);      # 显式具现化
template void foo(const double&);  # 如果漏了这几行，仍然会有链接错误。
template char bar(const char&);

$ g++ -c foobar.cc
$ nm foobar.o           # foobar.o 包含模板函数的定义
0000000000000000 W _Z3barIcET_RKS0_
0000000000000000 W _Z3fooIdEvRKT_
0000000000000000 W _Z3fooIiEvRKT_

$ g++ main.o foobar.o      # 可以成功生成 a.out

```

对于通用 (universal) 的模板库，这个办法是行不通的，因为你不可能实现知道客户会用哪些参数类型来具现化你的模板（比方说 `vector<T>` 和 `shared_ptr<T>`）。但是对于某些特殊情况，这可以减少代码膨胀，比方说把 `Buffer<int>` 的构造函数从头文件移到某个源文件，并且只具现化几个固定的长度，这样防止客户代码任意具现化 `Buffer` 模板。

对于 `private` 成员函数模板，我们也不用在头文件中给出定义，因为用户代码不能调用它也就无法随意具现化它，不会造成链接错误。考虑下面这个多功能打印机的例子，`Printer` 既能打印，也能扫描。`PrintRequest` 和 `ScanRequest` 都是由代码生成器生成的 `class`，它们有一些共同的成员，但是没有共同的基类。

```
class PrintRequest                                     Request.h
{
public:
    int getUserId() const { return userId_; }
    // other members
private:
    int userId_;
};

class ScanRequest
{
public:
    int getUserId() const { return userId_; }
    // other members
private:
    int userId_;
};
```

Request.h

我们写一个 `Printer class`，能同时处理这两种请求，为了避免代码重复，我们打算用一个函数模板来解析 `request` 的公共部分。

```
class PrintRequest;
class ScanRequest;

class Printer : boost::noncopyable // 注意 Printer 不是模板
{
public:
    void onRequest(const PrintRequest&);
    void onRequest(const ScanRequest&);

private:
    template<typename REQ>
    void decodeRequest(const REQ&);
```

Printer.h

```
void processRequest();

int currentRequestUserId_;
};
```

Printer.h

这个 `decodeRequest` 是模板，但不必把实现暴露在头文件中，因为只有 `onRequest` 会调用它。我们可以把这个成员函数模板的实现放到源文件中。

```
#include "Printer.h"
#include "Request.h"

template<typename REQ>
void Printer::decodeRequest(const REQ& req)
{
    currentRequestUserId_ = req.getUserId();
    // decode other parts
}
```

现在编译器能看到 `decodeRequest` 的定义，也就能自动具现化它

```
void Printer::onRequest(const PrintRequest& req)
{
    decodeRequest(req);
    processRequest();
}
```

```
void Printer::onRequest(const ScanRequest& req)
{
    decodeRequest(req);
    processRequest();
}
```

Printer.cc

前面展示的几种 `template` 用法一般不会用在通用的模板库中，因此很少有书籍或文章谈到它们。在编写应用程序的时候适当使用模板能减少重复劳动，降低出错的可能，值得了解一下。

另外，C++11 新增了 `extern template` 特性，可以阻止隐式模板具现化。g++ 很早就支持这个特性，g++ 的 C++ 标准库就使用了这个办法，使得使用 `std::string` 和 `std::iostream` 的代码不受代码膨胀之苦。

```
$ cat ios.cc
#include <iostream>
#include <string>
```

```

using namespace std;
int main()           // 用到了 iostream 和 string 两个大模板
{
    string name;
    cin >> name;
    cout << "Hello, " << name << "\n";
}

$ g++ ios.cc
$ size a.out          # 生成的可执行文件很小
   text    data     bss     dec     hex filename
   2900     648     584    4132    1024 a.out

$ nm a.out |grep ' [TW] ' # 仔细看目标文件，并没有具现化那些巨大的类模板
0000000000400c20 T __libc_csu_fini
0000000000400c30 T __libc_csu_init
0000000000400cf8 T _fini
0000000000400958 T _init
0000000000400a50 T _start
0000000000601288 W data_start
0000000000400b34 T main

$ nm a.out |grep -o ' U .*' # 而是引用了标准库中的实现
U _ZNSsC1Ev@@GLIBCXX_3.4 # 这两个是 string 的构造函数与析构函数
U _ZNSsD1Ev@@GLIBCXX_3.4
U _ZNSt8ios_base4InitC1Ev@@GLIBCXX_3.4
U _ZNSt8ios_base4InitD1Ev@@GLIBCXX_3.4
U _ZStlsISt11char_traitsIcESaIcEERSt13basic_ostreamIT_0ES7_RKSbIS4_S5_T1_E
U _ZStlsIcSt11char_traitsIcESaIcEERSt13basic_ostreamIT_0ES7_RKSbIS4_S5_T1_E
U _ZStrsIcSt11char_traitsIcESaIcEERSt13basic_istreamIT_0ES7_RSbIS4_S5_T1_E

```

这或许能帮助消除一定模板恐惧吧。

15.3.4 虚函数

在现在的 C++ 实现中，虚函数的动态调用（动态绑定、运行期决议）是透过虚函数表 (vtable) 进行的，每个多态 class 都应该有一份 vtable。定义或继承了虚函数的对象中会有一个隐含成员：指向 vtable 的指针，即 vptr。在构造和析构对象的时候，编译器生成的代码会修改这个 vptr 成员，这就要用到 vtable 的定义（使用其地址）。因此我们我们有时看到的链接错误不是抱怨找不到某个虚函数的定义，而是找不到虚函数表的定义。例如

```

$ cat virt.cc
class Base
{
public:
    virtual ~Base();

```

```
    virtual void doIt();
};

int main()
{
    Base* b = new Base;
    b->doIt();
}

$ g++ virt.cc
/tmp/cc8Q7qKi.o: In function 'Base::Base()':
virt.cc:(.text._ZN4BaseC1Ev[Base::Base()]+0xf):
        undefined reference to 'vtable for Base'
collect2: ld returned 1 exit status
```

出现这种错误的根本原因是程序中某个虚函数没有定义，知道了这个方向，查找问题就不难了。

另外，按道理说，一个多态 class 的 `vtable` 应该恰好被某一个目标文件定义，这样链接就不会有错。但是 C++ 编译器有时无法判断是否应该在当前编译单元生成 `vtable` 定义¹⁵⁷，为了保险起见，只能每个编译单元都生成 `vtable`，交给链接器去消除重复数据¹⁵⁸。有时我们不希望 `vtable` 导致目标文件膨胀，可以在头文件的 class 定义中声明 `out-line` 虚函数¹⁵⁹。

15.4 工程项目中头文件的使用规则

既然短时间内 C++ 还无法摆脱头文件和预处理，我们要深入理解可能存在的陷阱。在实际项目中，有必要规范头文件和预处理的用法，避免它们的危害。

15.4.1 头文件的害处

我认为头文件的害处主要体现在以下几方面：

- 传递性，头文件可以再包含其他头文件。前面已经举过例子，一个简单的 `#include <complex>` 展开之后有两万多行代码，一方面造成编译缓慢，另一方面，任何一个头文件改动一点点代码都会需要重新编译所有直接或间接包含它的源文件。因为 `build tool` 无法有效判断这个改动是否会影响程序语义，保守

¹⁵⁷就跟“无法 `inline` 的 `inline` 函数”一个道理

¹⁵⁸<http://sourcery.mentor.com/public/cxx-abi/abi.html#vague-vtable>

¹⁵⁹http://llvm.org/docs/CodingStandards.html#ll_virtual_anch

起见只能把受影响的源文件全部重新编译一遍。因此合理组织源代码，减少开发时 **rebuild** 的成本是每个稍具规模的项目必做功课。

- 顺序性，一个源文件可以包含多个头文件。如果头文件内容组织不当，会造成程序的语义跟头文件包含的顺序有关，也跟是否包含某一个头文件有关¹⁶⁰。通常的做法是把头文件分为几类¹⁶¹，然后分别按顺序包含这几类头文件¹⁶²，相同类的头文件按文件名的字母排序。这样一方面源代码比较整洁，另一方面如果两个人同时修改源码，各自想多包含一个头文件，那么造成冲突的可能性较小。一般应该避免每次在 **#include** 列表的末尾添加新的头文件，这样很快代码的依赖关系就无法管理了。
- 差异性。内容差异造成不同源文件看到的头文件不一致，时间差异造成头文件与库文件内容不一致。例如第 13 节提到不同的编译选项会造成 **Visual C++** **std::string** 的大小不一样。也就是说 **<string>** 头文件的内容经过预处理后会有变化，如果两个源文件编译时的宏定义选项不一致，可能造成二进制代码不兼容。这说明整个程序应该用统一的编译选项¹⁶³。如果程序用到了第三方静态库或者动态库，除了拿到头文件和库文件，我们还要拿到当时编译这个库的编译选项，才能安全无误地使用这个程序库。如果程序用到了两个库，但是它们的编译选项有冲突，那就麻烦大了，后面谈库文件组织的时候再来说这个问题和时间差异的问题。

反观现代的编程语言，它们比 C++ 的历史包袱轻多了，模块化做得也比较好。模块化的做法主要有两种：

- 对于解释型语言，**import** 的时候直接把对应模块的源文件解析 (**parse**) 一遍（不再是简单地把源文件包含进来）。
- 对于编译型语言，编译出来的目标文件（例如 **Java** 的 **.class** 文件）里直接包含了足够的元数据，**import** 的时候只需要读目标文件的内容，不需要读源文件。

这两种做法都不会造成声明与定义不一致的问题，因为在这些语言里声明也定义是不分家的。同时这种 **import** 手法也不会引入不想要的名字，大大简化了名字查找的负担（无论是人脑还是编译器），也不用担心 **import** 的顺序不同造成代码功能变化。

¹⁶⁰假设有两个源文件，一个包含了 **foo.h**，一个没有，**foo.h** 里定义了特殊的宏或者模板特化或者 **struct** 对齐指令，那么这两个源文件中相同代码的行为就可能不一致了。而且这种不一致很难追查。

¹⁶¹例如分为 C 语言系统头文件、C++ 标准库头文件、C++ 第三方库头文件、本公司的基础库头文件、本项目的头文件。

¹⁶²不同的编程规范的建议不一致，我个人是按从特殊到一般的顺序包含头文件，见 **muduo** 源码。

¹⁶³另外一个例子是 **g++** 的 **-malign-double** 选项会影响 32-bit 下 **double** 类型的地址对齐，如果两个源码的编译选项不同，造成同一个 **struct/class** 的 **layout** 不同，那么程序的行为就很飘逸了。

15.4.2 头文件的使用规则

几乎每个 C++ 编程规范都会涉及头文件的组织。归纳起来观点如下：

- “将文件间的编译依赖降至最小。” [2, item31]
- “将定义式之间依赖关系降至最小。避免循环依赖。” [4, item22]
- “让 class 名字、头文件名字、源文件名字直接相关。”¹⁶⁴ 这样方便源代码的定位。muduo 源码遵循这一原则，例如 TcpClient class 的头文件是 TcpClient.h，其成员函数定义在 TcpClient.cc。
- “令头文件自给自足。” [4, item23] 例如要使用 muduo 的 TcpServer，可以直接包含 TcpServer.h。为了验证 TcpServer.h 的自足性，TcpServer.cc 第一个包含的头文件就是它。
- “总是在头文件内写内部 #include guard（护套），不要在源文件写外部护套。” [4, item24] 这是因为现在的预处理对这种通用做法有特别的优化，GNU cpp 在第二次 #include 同一个头文件时甚至不会去读这个文件，而是直接跳过¹⁶⁵。
- #include guard 用的宏的名字应该包含文件的路径全名（从版本管理器的角度），必要的话还要加上项目名称（如果每个项目有自己的代码仓库）¹⁶⁶。
- 如果编写程序库，那么公开的头文件应该表达模块的接口¹⁶⁷，必要的时候可以把实现细节放到内部头文件中。muduo 的头文件满足这条规则，见第 ?? 节。

遵循以上规则，作为应用程序的作者，一般就不会遇到跟头文件和预处理相关的诡异问题。下面我们谈谈在编写和使用库的时候应该注意些什么。

15.5 工程项目中库文件的组织原则

考虑一个稍具规模的公司，有一个基础库团队，开发并维护公司的公共 C++ 网络库 net；还有一个团队开发了一套消息中间件，并提供了一个客户端库 hub，hub 是基于 net 构建的；最近公司又有另外一个团队开发了一套存储系统，并提供了一个客户端库 cab，cab 也是基于 net 构建的。公司内部开发的的服务端程序可能会用到这些库的一个或几个，本节主要讨论如何组织这些由不同团队开发的库与应用程序。

¹⁶⁴http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml#Header_Files

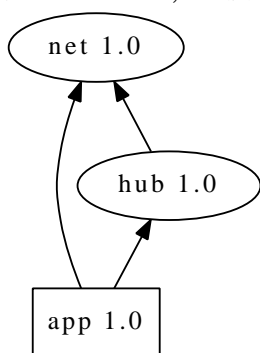
¹⁶⁵http://gcc.gnu.org/onlinedocs/gcc-4.4.4/cpp/Once_002dOnly-Headers.html

¹⁶⁶http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml#The__define_Guard

¹⁶⁷http://llvm.org/docs/CodingStandards.html#hl_module

在谈具体的 C++ 库文件的组织之前，先谈一谈更基本的话题：依赖管理。

假设你负责实现并维护一个关键的网络服务程序 **app**，经过充分测试之后，**app 1.0** 上线运行，一切顺利。**app 1.0** 用到了网络库 (**net 1.0**) 和消息中间件的客户端库 (**hub 1.0**)，并且 **hub 1.0** 本身也用到了 **net 1.0**，依赖关系如下：



尽管在发布之前 QA 人员 sign-off 的是 **app 1.0**，但是我们应该认为他们 sign-off 的是 **app 1.0** 和它依赖的所有库构成的 **bundle**。因为 **app** 的行为跟它用到的库有关，如果改变其中任何一个库，**app** 的行为都可能发生变化（尽管 **app** 的源码和可执行文件一个字节都没动），也就可能跟当时充分测试通过的“**app 1.0**”行为不一致。

周伟明老师在《软件测试实践》第 1.7.2 节“COM 的可测试性分析”中明确表示，COM “违反了软件设计的基本原理”，其理由是：

我们假设一个软件包含有 n 个不同的 COM 组件，按照 COM 的设计思想，每个组件都是可以替换的。假设每个组件都有若干个不同的版本，记为分别有 M_1, M_2, \dots, M_n 个不同的版本，那么组成整个软件的所有组件的组合关系有 $M_1 \times M_2 \times \dots \times M_n$ 种，等于这个软件共有 $\prod_{i=1}^n M_i$ 种二进制版本。如果要将测试做得充分，这些组合全部都需要进行测试，否则很难保证没测试到的组合不会有问题。

这至少从理论上说明，改动程序本身或它依赖的库之后应该重新测试，否则测试通过的版本和实际运行的版本根本就是两个东西。一旦出了问题，责任就难理清了。

这个问题对于 C++ 之外的语言也同样存在，我认为凡是可以在编译之后替换库的语言都需要考虑类似的问题¹⁶⁸。对于脚本语言来说，除了库之外，解释器的版本 (Python 2.5/2.6/2.7) 也会影响程序的行为，因此有 Python virtualenv 和 Ruby

¹⁶⁸http://en.wikipedia.org/wiki/Dependency_hell

`rbenv` 这样的工具，允许一台机器同时安装多个解释器版本。`Java` 程序的行为除了跟 `class path` 里的那些 `jar` 文件有关，也跟 `JVM` 的版本有关，通常我们不能在没有充分测试的情况下升级 `JVM` 的大版本（从 1.5 到 1.6）。

除了库和运行环境，还有一种依赖是对外部进程的依赖，例如 `app` 程序依赖某些数据源（运行在别的机器上的进程），会在运行的时候通过某种网络协议从这些数据源定期或不定期读取数据。数据源可能会升级，其行为也可能变化，如何管理这种依赖就超出本文的范围了。

回到 C++，首先谈编译器版本之间的兼容性。截至 `g++ 4.4`，`Linux` 目前已有四个互不兼容的 `ABI`¹⁶⁹ 版本，编译出来的库互不通用：

1. `gcc 3.0` 之前的版本，例如 2.95.3。

2. `gcc 3.0/3.1`。¹⁷⁰

3. `gcc 3.2/3.3`。¹⁷¹

4. `gcc 3.4 ~ 4.4`。^{172 173}

现在看来，这其实影响不大，因为估计没有谁还在用 `g++ 3.x` 来编译新的代码。

另外一个需要考虑的是 C++ 标准库 (`libstdc++`) 的版本与 C 标准库 (`glibc`) 的版本。C++ 标准库的版本跟 C++ 编译器直接关联¹⁷⁴，我想一般不会有人去替换系统的 `libstdc++`。C 标准库的版本跟 `Linux` 操作系统的版本直接相关，见下表。一般也不会有人单独升级 `glibc`，因为这基本上意味着需要重新编译用户态的所有代码。另外，为了稳妥起见，通常建议用 `Linux` 发行版自带的那个 `gcc` 版本来编译你的代码。因为这个版本的 `gcc` 是 `Linux` 发行版主要支持的编译器版本，当前 `kernel` 和用户态的其他程序也基本是它编译的，如果它有什么问题的话早就被人发现了。

根据以上分析，一旦选定了生产环境中操作系统的版本，另外三样东西的版本就确定了。我们暂且认为生产环境中运行 `app 1.0` 的机器的 `Linux` 操作系统版本、`libstdc++` 版本、`glibc` 版本是统一的¹⁷⁵，而且 C++ 应用程序和库的代码都是用操作系统原生的 `g++` 来编译。

¹⁶⁹这个指的是编译器的 `ABI`，跟本章前面提到的库的 `ABI` 不是一回事。

<http://gcc.gnu.org/onlinedocs/gcc/Compatibility.html>

¹⁷⁰Getting the Best From G++. Nathan Sidwell. 2003.

¹⁷¹<http://gcc.gnu.org/gcc-3.2/c++-abi.html>

¹⁷²<http://lwn.net/Articles/142828/>

¹⁷³http://gcc.gnu.org/onlinedocs/gcc/C_002b_002b-Dialect-Options.html

¹⁷⁴<http://gcc.gnu.org/onlinedocs/libstdc++/manual/abi.html>

¹⁷⁵升级操作系统时这三个都会一起变，那时候程序几乎肯定要重新测试并重新部署上线。

Distro	kernel	gcc	glibc
RHEL 6	2.6.32	4.4.6	2.12
RHEL 5	2.6.18	4.1.2	2.5
RHEL 4	2.6.9	3.4.6	2.3.4
Debian 6.0	2.6.32	4.4.5	2.11.2
Debian 5.0	2.6.26	4.3.2	2.7
Debian 4.0	2.6.18	4.1.1	2.3.6
Ubuntu 10.04 LTS	2.6.32	4.4.3	2.11.1
Ubuntu 8.04 LTS	2.6.24	4.2.3	2.7
Ubuntu 6.04 LTS	2.6.15	4.0.3	2.3.6

这样一来，我们就可以在 C++ 编译器版本、C++ 标准库版本、C 标准库版本均固定的情况下讨论应用程序与库的组织¹⁷⁶。进一步说，这里讨论的是公司内部实现的库，而不是操作系统自带的编译好的库（libz、libssl、libpthread 等等）。后面这些库可以通过操作系统的 package 管理机制来统一部署，确保每台机器的环境相同。

Linux 的共享库 (shared library) 比 Windows 的动态链接库在 C++ 编程方面要好用得多，对应用程序来说基本可算是透明的，跟使用静态库无区别。主要体现在：

- 一致的内存管理。Linux 动态库与应用程序共享同一个 heap，因此动态库分配的内存可以交给应用程序去释放¹⁷⁷，反之亦可。
- 一致的初始化。动态库里的静态对象（全局对象、namespace 级的对象等等）的初始化和程序其他地方的静态对象一样，不用特别区分对象的位置。
- 在动态库的接口中可以放心地使用 class、STL、boost（如果版本相同）。
- 没有 dllimport/dllexport 的累赘。直接 include 头文件就能使用。
- DLL Hell^{178 179} 的问题也小得多，因为 Linux 允许多个版本的动态库并存，而且每个符号可以有多个版本¹⁸⁰。

¹⁷⁶注意几大主流 Linux 发行版不约而同地在最新稳定版中选择了 Kernel 2.6.32、g++4.4，这也是 muduo 选择 kernel 2.6.32、g++4.4 为首要支持平台的原因。

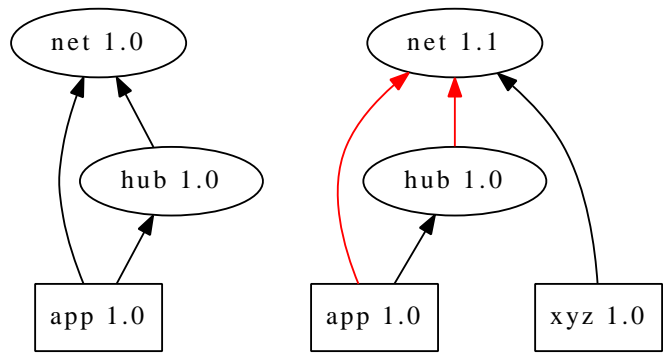
¹⁷⁷例如 libreadline 的 readline(3) 返回的 char* 指针必须由调用方用 free(3) 释放。

¹⁷⁸http://en.wikipedia.org/wiki/DLL_Hell

¹⁷⁹<http://www.desaware.com/tech/dllhell.aspx>

¹⁸⁰见 [6] 第 8 章“Linux 共享库的组织”

DLL hell 指的是安装新的软件的时候更新了某个公用的 DLL，破坏了其他已有软件的功能。例如安装 xyz 1.0 会把 net 库升级为 1.1 版，覆盖了原来 app 1.0 和 hub 1.0 依赖的 net 1.0，这有潜在的风险。



现在 Windows 7 里有 side-by-side assembly，基本解决了 DLL hell 问题，代价是系统里有一个巨大的且不断增长的 WinSxS 目录。

一个 C++ 库的发布方式有三种：动态库 (.so)、静态库 (.a)、源码库 (.cc) ¹⁸¹。下表简单总结了一些基本特性。

	动态库	静态库	源码库
库的发布方式	头文件 + .so 文件	头文件 + .a 文件	头文件 + .cc 文件
程序编译时间	短	短	长
查询依赖	ldd 查询	编译期信息	编译期信息
部署	可执行文件 + 动态库	单一可执行文件	单一可执行文件
主要时间差	编译时 ⇔ 运行时	编译库 ⇔ 编译应用程序	无

本文谈动态库只包括编译时就链接动态库的那种常规用法，不包括运行期动态加载 (dlopen) 的用法。

作为应用程序的作者，如果要在多台 Linux 机器上运行这个程序，我们先要把它部署 (deploy) 到那些机器上 ¹⁸²。如果程序只依赖操作系统本身提供的库（包括可以通过 package 管理软件安装的第三方库），那么只要把可执行文件拷贝到目标机器上就能运行。这是静态库和源码库在分布式环境下的突出优点之一。

相反，如果依赖公司内部实现的动态库，这些库必须事先（或者同时）部署到这些机器上，应用程序才能正常运行。这立刻就会面临运维方面的挑战：部署动态库的

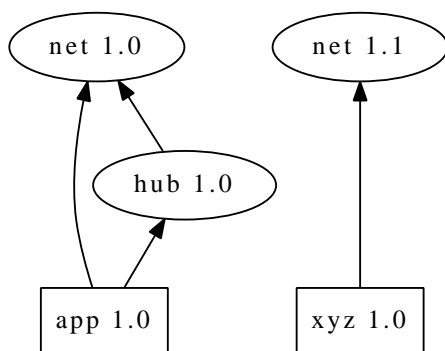
¹⁸¹ header-only 的库也算是源码库

¹⁸²如果处于测试目的，多台机器可以从某个网络文件系统启动可执行文件。但是在生产环境中，一般要把可执行文件放到本地文件系统，以减少依赖、增强可用性。

工作由谁（库的作者还是应用程序的作者）来做呢？另外一个相关的是，如果动态库的作者修正了 **bug**，他可以自主更新所有机器上的库吗？

我们暂且认为库的作者可以独立地部署并更新动态库，并且影响到使用这个库的应用程序¹⁸³。否则的话，如果每个程序都把自己用到的动态库和应用程序一起打包发布，库的作者不负责库的更新，那么这和使用静态库就没有区别了，还不如直接静态链接多好。

无论哪种方式，我们都必须保证应用程序之间的独立性，也就是让动态库的多个大版本能够并存。例如部署 **app 1.0** 和 **xyz 1.0** 之后的变成：



按照传统的观点，动态库比静态库节省磁盘空间和内存空间¹⁸⁴，并且具备动态更新的能力（可以 **hot fix bug**¹⁸⁵），似乎动态库应该是目前的首选¹⁸⁶。但是正是这种动态更新¹⁸⁷的能力让动态库成了烫手的山芋。

15.5.1 动态库是有害的

Jeffrey Richter 对动态库的本质问题有精辟的论述¹⁸⁸：

¹⁸³具体地说这跟动态库的版本规划有关，比如 **net 1.1.1** 升级到 **net 1.1.2** 只会影响原来使用 **net 1.1** 系列的应用程序，不影响使用 **net 1.0** 的应用程序。

¹⁸⁴对于系统库或许真的如此，但是对于我们自己写的业务库则不一定有多大实际的效果。假设一台服务器上运行 10 个不同的服务程序，每个程序的可执行文件大小是 100MB（当然这是非常夸张的估算），那么一共用了 1GB 的内存来装载代码，相比服务器动辄几十 GB 的内存来说简直是九牛一毛。

¹⁸⁵当然头文件里 **inline** 函数的 **bug** 不能通过发布新的库文件来修正，而必须重新编译可执行文件。

¹⁸⁶1990 年代出版的《C 专家编程》[9]就大力推崇动态库。

¹⁸⁷当然我们不能原地 (**in-place**) 覆盖更新正在使用的动态库或可执行文件，这会让进程在一段时间之后因 **SIGBUS** 而崩溃。

¹⁸⁸《Applied Microsoft .NET Framework Programming》第 3 章，李建忠译。

一旦替换了某个应用程序用到的动态库，先前运行正常的这个程序使用的将不再是当初 build 和测试时的代码。结果是程序的行为变得不可预期。

怎样在 fix bug 和增加 feature 的同时，还能保证不会损坏现有的应用程序？我 (Jeffrey Richter) 曾经对这个问题思考了很久，并且得出了一个结论——那就是这是不可能的。

作为库的作者，你肯定不希望更新部署一个看似有益无害的 bug fix 之后，星期一早上被应用程序的维护者的电话吵醒，说程序不能启动（新的库破坏了二进制兼容性）或者出现了不符合预期的行为。

作为应用程序的作者，你也肯定不希望星期一大早上被运维的同事吵醒，说你负责的某个服务进程无法启动或者行为异常。经排查，发现只有某一个动态库的版本与上星期不同。你该朝谁发火呢？

既然双方都不想过这种提心吊胆的日子，那为什么还要用动态库呢？

那么有没有可能在发布动态库的 bug fix 之前充分测试所有受影响的应用程序呢？这会遇到一个二难命题：一个动态库的使用面窄，只有两三个程序用到它，测试的成本较低，那么它作为动态库的优势就不明显。相反，一个动态库的使用面宽，有几十个程序用到它，动态库的“优势”明显，测试和更新的成本也相应很高（或许高到足以抵消它的“优势”）。有一种做法是把动态库的更新先发布到 QA 环境，正常运行一段时间之后再发布到生产环境，这么做也有另外的问题：你在测试下一版 app 1.1 的时候，该用 QA 环境的动态库版本还是用生产环境的动态库版本？如果程序在编译测试之后行为还会改变，这是不是在让 QA 白费力气？

总之，一旦动态库可能频繁更新¹⁸⁹，我没有发现一个完美的使用动态库的办法。

在决定使用动态库之前，我建议至少要熟悉它的各种陷阱。参考资料如下：

- <http://harmful.cat-v.org/software/dynamic-linking/>
- A Quick Tour of Compiling, Linking, Loading, and Handling Libraries on Unix.¹⁹⁰
- How to write shared libraries. Ulrich Drepper.¹⁹¹
- Good Practices in Library Design, Implementation, and Maintenance. ____.¹⁹²

¹⁸⁹ “频繁”指的是一两个月一次，因此如果一个应用程序使用了 5 个独立更新的这种动态库，那么几乎每周都会有库更新。

¹⁹⁰<http://ref.web.cern.ch/ref/CERN/CNL/2001/003/shared-lib/Pr/>

¹⁹¹<http://www.akkadia.org/drepper/dsohowto.pdf>

¹⁹²<http://www.akkadia.org/drepper/goodpractice.pdf>

- *Solaris Linker and Libraries Guide* ¹⁹³
- *Shared Libraries in SunOS* ¹⁹⁴

15.5.2 静态库也好不到哪儿去

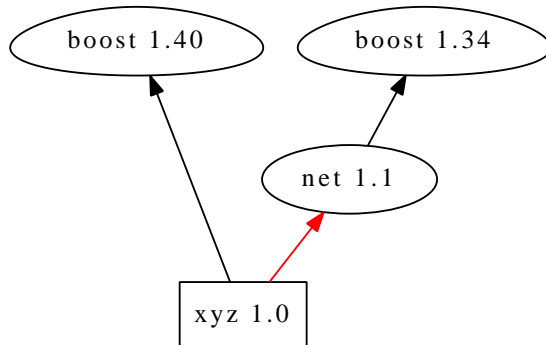
静态库相比动态库主要有几点好处：¹⁹⁵

- 依赖管理在编译期决定，不用担心日后它用的库会变。同理，调试 `core dump` 不会遇到库不一致而 `debug` 符号失效的情况。
- 运行速度可能更快，因为没有 PLT(过程查找表)，函数调用的开销更小。
- 发布方便，只要把单个可执行文件拷到模板机器上。

静态库的一个小缺点是链接比动态库慢，有的公司甚至专门开发了针对大型 C++ 程序的链接器 ¹⁹⁶。

静态库的作者把源文件编译成 `.a` 库文件，连同头文件一起打包发布。应用程序的作者用库的头文件编译自己的代码，并链接到 `.a` 库文件，得到可执行文件。这里有一个编译的时间差：编译库文件比编译可执行文件要早，这就可能造成编译应用程序时看到的头文件与编译静态库时不一样。

比方说编译 `net 1.1` 时用的是 `boost 1.34`，但是编译 `xyz` 这个应用程序的时候用的是 `boost 1.40`，见下图。这种不一致有可能导致编译错误，或者更糟糕地导致不可预期的运行错误。比方说 `net` 库以 `boost::function` 提供回调，但是 `boost 1.36` 去掉了一个模板类型参数 ¹⁹⁷，造成 `xyz 1.0` 用 `boost 1.40` 的话就与 `net 1.1` 不兼容。



¹⁹³<http://docs.oracle.com/cd/E19963-01/html/819-0690/index.html>

¹⁹⁴<http://www.cs.cornell.edu/courses/cs414/2004fa/sharedlib.pdf>

¹⁹⁵http://en.wikipedia.org/wiki/Static_library

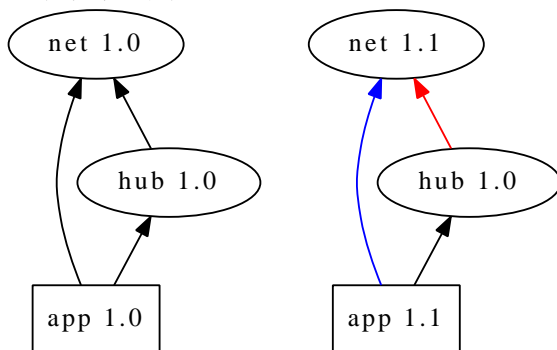
¹⁹⁶<http://research.google.com/pubs/pub34417.html>

¹⁹⁷<http://www.boost.org/doc/html/function/history.html>

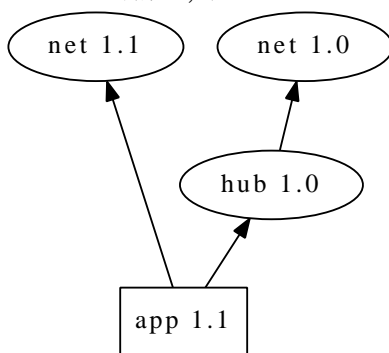
这说明应用程序在使用静态库的时候必须要采用完全相同的开发环境（更底层的库、编译器版本、编译器选项）。但是万一两个静态库的依赖有冲突怎么办？

静态库把库之间的版本依赖完全放倒编译期，这比动态库要省心得多，但是仍然不是一件容易的事情。下面略举几种可能遇到的情况。

- 迫使升级高版本。假设一开始应用程序 `app 1.0` 依赖 `net 1.0` 和 `hub 1.0`，一切正常。在开发 `app 1.1` 的时候，我们要用到 `net 1.1` 的功能。但是 `hub 1.0` 仍然依赖 `net 1.0`，`hub` 库的作者暂时没有升级到 `net 1.1` 的打算。如果不小心的话，就会造成 `hub 1.0` 链接到 `net 1.1`，这就跟编译 `hub 1.0` 的环境不同了，`hub 1.0` 的行为不再是经过充分测试的。



- 重复链接。如果 `makefile` 编写不当，有可能出现 `hub 1.0` 继续链接到 `net 1.0`，而应用程序则链接到 `net 1.1` 的情况，如下图。

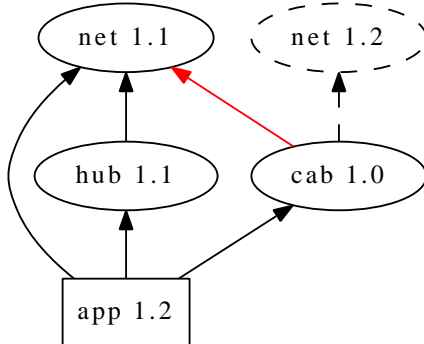


这时如果 `net` 库里有 `internal linkage` 的静态变量，可能造成奇怪的行为，因为同一个变量现在有了两个实体，违背了 ODR。一个具体的例子见云风的博客

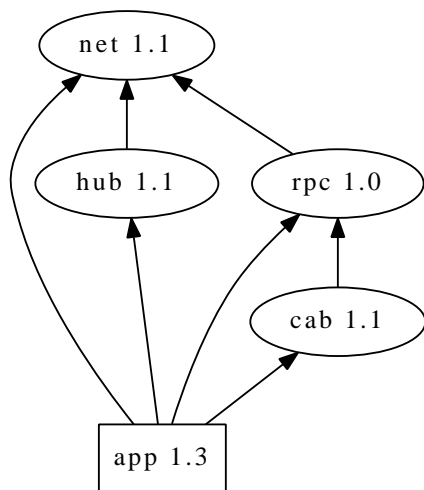
198

¹⁹⁸http://blog.codingnow.com/2012/01/lua_link_bug.html

- 版本冲突。比方说 **app** 升级到 1.2 版，想加入一个库 **cab** 1.0，但是 **cab** 1.0 依赖 **net** 1.2，如下图。这时我们的问题是，如果用 **net** 1.1，则不满足 **cab** 1.0 的需求，如果用 **net** 1.2，则不满足 **hub** 1.1 的需求。那该怎么办？



可见静态库的版本管理的并不如想象中那么简单。如果一个应用程序用到了三四个公司内部的静态库（见下图），那么协调库之间的版本要花一番脑筋，单独升级任何一个库都可能破坏它原本的依赖。



静态库的演化也比较费事。

到目前为止我们认为公司没有历史包袱，所有的机器都是 2009 年前后买的，运行的是 **Ubuntu 8.04 LTS**，软件版本是 **g++ 4.2**、**glibc 2.7**、**boost 1.34** 等等，C++ 程序和库也都是在这个统一的环境下开发的。现在到了 2012 年，线上服务器已服役满 3 年，进入换代周期。新购买的机器打算升级到 **Ubuntu 10.04 LTS**，因为新内核的驱

动程序对新硬件支持更好，而且 8.04 到还有一年多就停止支持了。这样同时升级了内核、gcc 4.4、glibc 2.11、boost 1.40。

这就要求静态库的作者得为新系统重新编译并发布新的库文件，为了避免混淆，我们不得不为库加上后缀名，以标明环境和依赖。假设目前有 `net 1.0`、`net 1.1`、`net 1.2`、`hub 1.0`、`hub 1.1`、`cab 1.0` 等现役的库，那么需要发布多个版本的静态库：

- `net1.0_boost1.34_gcc42`
- `net1.0_boost1.40_gcc44`
- `net1.1_boost1.34_gcc42`
- `net1.1_boost1.40_gcc44`
- `net1.2_boost1.34_gcc42`
- `net1.2_boost1.40_gcc44`
- `hub1.0_net1.0_boost1.34_gcc42`
- `hub1.0_net1.0_boost1.40_gcc44`
- `hub1.1_net1.1_boost1.34_gcc42`
- `hub1.1_net1.1_boost1.40_gcc44`
- `cab1.0_net1.2_boost1.34_gcc42`
- `cab1.0_net1.2_boost1.40_gcc44`

这种组合爆炸式的增长让人措手不及，因为任何一个底层库新增一个变体 (variant)，所有依赖它的高层库都要为之编译一个版本。

要想摆脱这个困境，我目前能想到的办法是使用源码库，即每个应用程序都从头编译所需的库，把时间差减到最小。

15.5.3 源码编译是王道

每个应用程序自己选择要用到的库，并自行编译为单个可执行文件。彻底避免头文件与库文件之间的时间差，也不用为库的版本搭配操心。这么做的缺点是编译时间很长，因为把各个库的编译任务从库文件的作者转嫁到了每个应用程序的作者。

另外，最好能和源码版本工具配合，让应用程序只需指定用哪个库，`build` 工具能自动帮我们 `check out` 库的源码。这样库的作者只需要维护少数几个 `branch`，发布

库的时候不需要把头文件和库文件打包供人下载，只要 `push` 到特定的 `branch` 就行。而且这个 `build` 工具最好还能解析库的 `makefile`（或等价的 `build script`），自动帮我们解决库的传递性依赖，就像 `Apache Ivy` 能做的那样。在目前看到的 `build` 工具里，最接近这一点的是 `Chromium` 的 `gyp`，其他如 `SCons`、`CMake`、`Premake`、`Waf` 等等工具仍然是以库的思路来搭建项目。

总结：由于 `C++` 的头文件与源文件分离，并且目标文件里没有足够的元数据供编译器使用，因此必须同时提供库文件和头文件。也就是说要想使用一个已经编译好的 `C/C++` 库（无论是静态库还是动态库），我们需要两样东西，一是头文件（`.h`），二是库文件（`.a` 或 `.so`），这就存在了两样东西不匹配的可能。这是造就 `C++` 简陋脆弱的模块机制的根本原因。`C++` 库之间的依赖管理远比其他现代语言复杂，在编写程序库和应用程序时，要熟悉各种机制的优缺点，采用开发及维护成本较低的方式来组织和发布库。

16 Zero overhead 原则

<http://www.open-std.org/jtc1/sc22/wg21/docs/18015.html>

<http://www.aristeia.com/c++-in-embedded.html>

Bibliography

- [1] 侯捷. 《池内春秋——Memory Pool 的设计哲学与无痛运用》. 《程序员》2002 年第 9 期。
- [2] Scott Meyers 著, 侯捷译. 《Effective C++ 中文版》(第 3 版). 电子工业出版社, 2006.
- [3] Scott Meyers 著. 《Effective STL》. Addison-Wesley, 2001.
- [4] Herb Sutter and Andrei Alexandrescu 著, 侯捷 陈硕译. *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*. 《C++ 编程规范》. 碁峰出版社, 2008.
- [5] Herb Sutter 著, 刘未鹏译. 《Exceptional C++ Style 中文版》. 人民邮电出版社, 2006.
- [6] 俞甲子 石凡 潘爱民著. 《程序员的自我修养——链接、装载与库》. 电子工业出版社, 2009.
- [7] Michael Feathers 著, 刘未鹏译. *Working Effectively with Legacy Code*. 《修改代码的艺术》. 人民邮电出版社, 2007.
- [8] Andrei Alexandrescu. Scalable Use of the STL. C++ and Beyond 2010. http://www.artima.com/shop/cpp_and_beyond_2010
- [9] Peter van der Linden 著. 《Expert C Programming: Deep C Secrets》. Prentice Hall Professional, 1994.
- [10] Randal E. Bryant and David R. O'Hallaron 著. 龚奕利 雷迎春 译. 《深入理解计算机系统 (第 2 版)》. 机械工业出版社, 2011.
- [11] Bjarne Stroustrup 著, 裘宗燕 译. 《C++ 语言的设计和演化》. 机械工业出版社, 2002.
- [12] 陈硕. 从《C++ Primer 第四版》入手学习 C++. 《C++ Primer 第 4 版 评注版》前言. 电子工业出版社, 2012.
<https://github.com/downloads/chenshuo/documents/LearnCpp.pdf>