纯真，才有希望

哈尔滨工业大学 信息检索实验室
刘挺 教授

我的观点：做科研只看重两个“率”，即论文的“引用率”和技术的“转让率”，其它一概不管。思想被同行所参考，技术被企业所使用，这才是货真价实的成功，什么 SCI、EI，什么鉴定获奖，这些虚幻的目标，常把人引向歧途，或让人空耗光阴。

我们的校园里不乏善于组织的管理者，不乏中规中矩的工程师，但是学者呢，热烈地追求学术的创新、技术的突破，独持己见，不为潮流所左右，超越名利，自由地翱翔在自己兴趣的天空中的学者，我们有吗？

庄子在逍遥游中描述了一个动人的故事，鲲经过“待风”“乘风”“背风”“离风”四个历程，从一条小鱼变成垂翼万里的大鹏。读罢令人神往，也令人叹息。毕竟我们都是凡人，我们等来了大风，等来了时代的潮流，却往往在风中迷失，在潮水中淹没，我们有背离潮流的勇气吗？

系里的一位老师，主动要求降薪，理由是他不想负担太多的开发任务，他要醉心于他所热爱的算法研究。他对自己是真诚的，而真诚是需要勇气的，我羡慕他的勇气，因为我没有。回想我自己，最快乐的好像是读硕士一年级的时候，我单兵独进，写了一个中文自动校对的软件，那时如醉如痴，心无旁骛，每每有所进展，便欣欣然忘乎所以。如今，带着一群弟子，兵精粮足，正该高歌猛进，奋发有为，却时常感到空虚烦闷，杂念纷飞，这是什么原因呢？

细想想，一个很重要的原因是我一直在向现实低头，渴望探寻技术的奥秘，渴望发明创造改变世界的初衷，被日复一日，年复一年的琐事所尘封。本来鲜明的“两个率”评价标准在执行中走了样，我打探着某某会议的论文到底能不能被 EI 全检，我关心着鉴定会请哪些专家比较妥当，我计算着我挣的工分年底能不能达标。在一个个短期目标被完美地实现之后，我发现长期目标离我越来越远了。还依稀记得我的梦想是：发明最好的中文信息处理技术。

求伯君一人写出 WPS 的时代一去不返了，个人英雄主义让位于团队精神，因此我以为离群索居，抛弃环境实不足取。我也认识个别忠于学问而性格古怪的学者，他们从单位里出来成了单干户，没有资源，没有条件，

终究无法成事。然而，我们要不时地追问，我们到底要什么？如果明天就死，我们回首短促的一生，是否会因为随波逐流、得过且过、庸碌平淡而悔恨。能不能，在顺应这个浮躁而喧闹的社会的同时，保留一点纯真，适当地抵御一下名利的诱惑，忘我地去做一些纯粹为了兴趣的研发？能不能，做出一些让自己感动，让用户叫绝的软件，在这个相互之间越来越象的世界里留下一些不同的印记，给人生一点希望？

我认识一位国内著名软件企业的助理总裁，33 岁了，每天写程序超过 6 小时，他亲自编写该公司最新的核心软件平台的代码。国外，很多资深的开发人员在 40-50 岁左右，经验极为丰富。而在中国，超过 30 岁还在第一线写程序会被认为没有前途了。无论是做研究，还是做开发，我们一直在用新手对抗人家的老手，因为我们的老手都去忙管理，忙社交，忙着去应付各种指标了。

读比尔盖茨的《未来之路》，他不断地提到他对某某技术感到“兴奋不已”，我理解这兴奋的背后就是对技术本身的兴趣加上对技术可能带来的巨大社会价值的预期。盖茨如果贪恋哈佛的虚名，就没有今天的微软了。做点自己真正感兴趣的事，做点对社会真正有价值的事。

刘挺博士，哈工大计算机学院教授，信息检索实验室常务主任。主要研究领域为自然语言处理和信息检索。个人主页地址: http://ir.hit.edu.cn/~tliu

纯 C 论坛杂志
2005 年 7 月第 3 期(总第 5 期)

纯 C 论坛
CSDN 电子杂志社区 合作

主　　编：worldguy
编　　辑：
Sun, kylix, iamxiaohan, pineapple, worldguy, xiong, true, hitool, ogg

论坛：
http://purec.binghua.com

投稿信箱：
purec@126.com

目录

卷首语

纯真，才有希望	刘挺	i

C/C++

C Develops Endless Future	孙志岗	1
Boost 源码笔记：Boost::multi_array	谢轩	5

编译原理

词法分析自动生成器 CScanGenner 的设计与实现	谢煜波	17
基于 Flex 的 c/c++代码加亮工具	Tinyfool	28

计算机病毒

解读经典样本 EICAR	SwordLea	32

技术资料

AS86 的 MAN	hold(译)	34

代码分析

一段代码的分析	谢煜波	39

编辑部通讯

投稿指南	编辑部	I

C Develops Endless Future

哈尔滨工业大学 计算机语言基础教研室
孙志岗（sun@hit.edu.cn）

题记：本来题目想叫做“从 C 到无穷大”，太晦涩了。后来又想用“C、D、E、F……”，太不知所云了。于是，用了这个题目，它的缩写恰好就是 CDEF。不管用什么样的题目，我所想表达的都一个意思，那就是 C 语言课程结束以后，我们该干什么。

我们都学过很多课程，每门课程都有大纲，有教材。严格按照大纲学习，把教材烂熟于胸，这是一贯的学习策略，并靠此斩落考试无数，一路杀入大学。如果你学习 C 语言的目的仅仅是为了期末考试，至多再加上一个“二级”，那么请不要再继续阅读本文，因为后面的文字只会给你带来负面作用。一门真正的大学课程的学习应该是永无止境的，没有任何机构可以给它划个框框说哪些该学哪些不用学，也不会有任何教材能解答你未来会遇到的所有疑问。本文仅希望帮助那些热爱编程，并把编程当作未来旅程中不可缺少的元素的同学。帮助大家把 C 作为一个起点，去开拓无尽的未来。

一、怎样学会 C？

C 是永远学不会的！

仅从语法上说，C 可能是所有高级语言中最简单的，最常用的关键字不会超过 30 个，语法规则也不复杂，没有需要死记硬背的“习惯用法”。如果 C 语言就像英语一样只是单词、语法，那么课程结束后所有人都可以说学会 C 了。每种自然语言都能用简单的文字与语法来描述复杂的大观世界，但并不是每个人都可以像爱因斯坦、霍金、马克思、李白、金庸等等那样用语言表达出令人景仰的内容。即便与你身边的人相比较，同样说汉语的你们也很难对同一个事物说出完全一样的观点。C 语言亦然。它是一个语言工具，通过这个工具，我们表达我们对世界的理解，或者具体说，对程序的看法。眼界与思维直接影响着我们写下代码的漂亮程度。如果以总能写下漂亮代码作为学会 C 语言的判断标准的话，那么因为眼界与思维的锻炼是无止境的，所以 C 是永远学不会的，就像我们都还没有真正学会汉语一样。

既然 C 有那么多可学的内容，那么我们该学什么呢？

二、学 C 学什么？

学 C 的目的并不在于要学会 C，否则就太狭隘了。我们要利用学习 C 的机会，拓展自己的眼界与思维，锻炼能力，成为世界（包括计算机世界、编程世界）的主宰，而不是 C 语言的奴隶。

第一学学习。“学习”是一个主动创造的过程而非被动接受。真正的学习需要主动去寻求广泛的知识，跟踪最新进展，综合思考判断各方观点，动手实践检验，进而形成自己的观点，再将其传播出去。C 语言的学习也符合这个规律。

经典书籍要读，手册和在线文档要经常查，Internet 更是提供了接触最新知识的机会。这里推荐两个网站：http://csdn.net 和 http://purec.binghua.com。后者是一名哈工大的学生在大四时创办的，以钻研深层技术为目标，被一位微软的员工评论为“牛人处处有，PureC 特别多”。但真正能把全世界的牛人集中在一起的地方是

Newsgroup：news://comp.lang.c。如果你不知道 Newsgroup 是什么，就马上去搜索引擎找答案吧（http://groups-beta.google.com/是一个适合新手的 Newsgroup 入口）。搜索引擎是最重要得知识来源，首推 google.com。可惜工大校园内的机房基本都不能出国访问，所以 baidu.com 也勉为其难地不得不用一下，尽管它搜出信息的可用性和可信性都差一些。在此特别说明一下，英文阅读能力十分重要，别回避英文信息，因为总会有你不能回避的那一天。硬着头皮冲上去，很快你就会发现读英文比读中文快乐很多。

如果对学到的知识一概接受，那就无趣了。当阅读范围扩张，会发现即便是权威的观点也会有很多矛盾，也会看到总有所谓“真理”被推翻。所以，带着“怀疑”的态度去阅读，可能更有利于习惯背书的中国学生开拓思维。C 语言中就有很多历史悬案，喋喋不休地争论至今没有结果，比如缩进该用空格还是 Tab，“{”是否该单列一行……。一定要有自己的观点，“尽信书不如无书”吗。

学编程，上机的重要性远远大于看书。亲自动手编程序的学习效果比干啃书本要好上 0xFFFFFFFF 倍。上机不要就是敲书上的例子，那只能锻炼出打字员。干点儿有挑战性的事情，比如编个小游戏，做个恶作剧程序，更可以考虑开发共享软件。如果你感到无从着手，那就下载别人的源代码看。同时，千万不要把自己陷到 TC2 里面，把路子走得宽一些，VC、gcc、Dev-C++、Eclipse……，广阔天地，精彩无限。

第二学计算机。除了 C++，再很难找到别的高级语言像 C 这样与计算机如此接近。只有懂得了 C 语言与计算机的内在原理，才可能轻松、快乐地驾驭 C。课堂教学能教会你类型、选择、循环、输入输出等，我喜欢称他们为 C 语言的“毛尖”，也就是连“皮毛”都没接触到。当每写下一行代码，不仅能想象到它的执行效果，更能熟知它将给 CPU、内存和各种计算机设备带来什么时，那种感觉是非常奇妙且令人激动的。这种情况下，你才能感觉到你真正地是在驾驭 C，用 C 来做一切你想做的事情，而不是在 C 的束缚下蹩手蹩脚地应付各种莫名其妙的问题。

为了找到这种感觉，就不要满足于课堂与一本教材，更不要迷信非专业人士捧出的经典。现在我们有如此好的机会可以接触众多的世界级大师、小师的著作，那么就应该抓住机会去直接与它们对话。下面介绍一些适合于入门的顶级教材。[Deitel94]已经出了第四版，不过国内只能买到第二版，它是一本可以把你引上程序设计的正路的书。[Prata04]的作者写了一系列“Primer”书，本本经典，数次升级，内容充实。[Kelley97]也是一本经典教材，它的最大好处是没有中文版，可以强迫你用英语去思考。[Roberts94]非常另类，作者是斯坦福大学的计算机系的资深教授，著名的学院派计算机教育专家。它的书教的是编程，而不是 C 语言（`scanf()`在 539 页第一次出现），所以如果你想学习上乘的编程技术，这本书是非常好的入门读物，但如果想学精深的 C 语言，它就帮不上忙了。[Harbison02]是“大全”型的，可能是唯一一个为了随时翻阅而值得收藏的书，所以它不是一个入门书，但当作初学者的辅助参考书非常的不错。

课程结束以后，首先要撕烂教材（没错，就是作者里有我名字的那本。撕烂了扔掉，总比让我频频在旧书市看到它更好过一些），然后马上去拜读[Kernighan88]。这本由 C 语言的设计者所著的书用极其简练却精确的文字描述着 C，每次捧卷都能让人对 C 语言的理解更提高一层。如果想获得绝对精确的信息，除了看[C99]别无选择，虽然 C99 很难看。进而可以阅读[Kernighan99]和[Linden94]来提高自己的实际编程能力、扩大见识。前者偏重于技巧与数据结构，后者偏重于经验与原理，尤其是后者还能让你发现原来编程是天底下最有趣的事情。当随着编程量的增加，你犯的错误也越来越多的时候，找来[Maguire93]，成为 bug 终结者。如果通过学习 C 语言你对计算机的深层原理产生了浓厚的兴趣，那么就看[Bryant02]吧。此外还有一本专门“贬”C 语言的书——[Koenig89]，也值得一看。这些书的作者兢兢业业地完成本职工作之余，还把很多相关但不在本书论述范围的内容以参考文献的形式推荐给读者。从参考文献出发，你可以发现更多值得品味的。

国人原创的技术书籍虽然不比老外经典，但也有其价值，间或冒个精品。这里推荐一本[林锐 03]，它论厚重自然不比大师的书，但颇有特点，挺无厘头的，一些即兴的感慨也给编程添加了点点人情味。

第三学数学。我学生阶段最大的憾事就是在本科时不仅没认识到数学的重要性，还对它产生了极其错误的看法，以至于今天一见到数学好的人就景仰得不敢抬头，自惭形秽。当初未遇名师指点，落此下场也算生不逢时。今以我的惨痛遭遇告诫大家，数学真的是百科之母，必须用精力好好孝敬。想体会数学的奇妙及现实、数学与 C 语言的完美结合，就到 http://acm.hit.edu.cn 去在线做题吧。

数学在程序设计中被具体化为算法与数据结构，关于它们的书我读的不多，下面的介绍更多的是参考别人

的评论而不是我个人的观点。数据结构比较简单，也应该先学习，建议看[Weiss96]，然后过渡到算法。算法领域里就林林总总彩旗飘飘了，[Cormen01]可能是一个很合适的入口，它帮你总览算法，寻找自己感兴趣的领域再继续找别的书籍深入下去。算法领域里最重要的书是[Knuth98]，非常经典，也非常大部头的三卷本，也非常贵，任何一个能读完哪怕其中一卷的人都足以令人景仰。我现在还只能把它们供奉到书架上。

三、C 以后学什么？

虽然 C 永远学不完，但绝不能抱 C 守缺。事实上，前面所述的很多内容已经不是 C 语言的范畴了，它们应该属于用计算机解决实际问题的技术。那么回到语言本身，C 可能是大多数同学学习的第一种计算机语言，但它不应该是最后一种。

计算机这个人类发明的最伟大的工具是每个人都无法回避的，随身带几种计算机语言，是潇洒走天下的一个有力保证。诚然说，考试过后还能再用上 C 的人是少数，但能不再编程的人在工科院校里还是很珍稀的。C 语言强大，号称无所不能，可它并不是解决所有问题的最佳选择。没有任何一种语言可以包打天下，它们都有自己擅长的领域和不擅长的方向。我们必须要针对自己的领域特色选择一种或几种最适合的语言来自学，同时也不需担心学习 C 语言的历程会白费，因为语言之间的“形”虽然不同，但“神”都是相通的。本文第二部分里建议大家学习的内容里很多就是这“神”的一部分。

能够超脱语言去思考程序设计的问题，才是真正的领会了编程的要旨。能把自己的思想用语言行云流水地表达出来，才是真正的编程高手。

参考文献：

注：括号内的书名是英文原名。斜体字的“中”表示有中文版，“英”表示有英文影印版，“E”表示有英文电子版，“电”表示有中文电子版（不包括超星格式），有删除线的表示该版本已绝版。请不要向我要索要电子版，并且大部分电子版属于盗版，慎重下载。同时因为世界总在变化，我不能保证你读到此文时这些信息依然有效。

[Deitel94] H. M. Deitel，《C 程序设计教程（C How to Program）》第二版，*中*。书中错误不少，不知道是原版的问题还是翻译的问题，但至少翻译版的排版一团糟，有点糟蹋好书了。

[Prata04] Stephen Prata，《C Primer Plus（C Primer Plus）》第五版，*中E*。“Primer”的名头太大了，以至于中文版干脆不翻译书名了。

[Kelley97] Al Kelley、Ira Pohl，《C 语言教程（A Book on C: Programming in C）》，*英*。

[Roberts94] Eric S. Roberts，《C 语言的科学与艺术（The Art and Science of C: A Library Based Introduction to Computer Science）》，*中英*。虽然另类，但有很多创新值得回味。我正在学习他把图形库引入 C 语言教学的方法。

[Harbison02] Harbison、Steele，《C 语言参考手册（C: A Reference Manual）》第五版，*英*。其实，这本书是有中文版的，但为了您的健康，还是忘掉他吧，阿门……

[Kernighan88] Kernighan、Ritchie，《C 程序设计语言（The C Programming Language）》第二版，*中~~英~~E*。这本书被简称为“K&R2”，并被尊称为 C 语言的“圣经”。Ritchie 就是 C 语言的爸爸。

[C99] ISO/IEC 9899:1999，C 标准 1999 年版，*E*。被简称为 C99。可以这样描述它：“1. C99 永远是对的；2. 如果 C99 错了，请参看第一条。”可笑的是，我们的国家计算机等级考试二级考试的很多题目都是违背 C99 的。相信 Ritchie 来考二级也会被郁闷住。

[Kernighan99] Kernighan、Pike，《程序设计实践（The Practice of Programming）》，*中英电*。中英文对照阅读，又学知识又练英语。

[Linden94] Peter van der Linden，《C 专家编程（Expert C Programming）》，*~~中~~E*。这是一本行文非常幽默的书（可惜译文版把很多幽默都搞丢了，这也是没办法的事情），因为书面上印着一条腔棘鱼，所以它在业界被戏

称为“鱼书”。

[Maguire93] Steve Maguire，《编程精粹— Microsoft 编写优质无错 C 程序秘诀（Writing Solid Code: Microsoft's Techniques for Developing Bug-Free C Programs）》，*E 电*。网上很多地方传播的电子版写的名字是《Write Clean Code》，这里替 Maguire 澄清一下。

[Bryant02] Bryant、O’Hallaron，《深入理解计算机系统（Computer Systems: A Programmer’s Perspective）》，*中英 E*。我一定要说，这本书的中译本是为数不多的翻译精品。

[Koenig89] Andrew Koenig，《C 陷阱与缺陷（C Traps and Pitfalls）》，*中 E 电*。因为年代久远，书中有些观点已经不合时宜，正好用来练习边读边批判。

[林锐 03] 林锐、韩永泉，《高质量程序设计指南——C++/C 语言》第二版，*中电*。我觉得这本书错误观点很多很多，正确的也很多很多，欢迎发表你的看法。

[Weiss96] Mark Allen Weiss，《数据结构与算法分析——C 语言描述（Data Structures and Algorithm Analysis in C）》第二版，*中*。

[Cormen01] Cormen、Leiserson、Rivest、Stein，《算法导论（Introduction to Algorithms）》第二版，*英 E*。

[Knuth98] Donald Knuth，《计算机程序设计艺术（The Art of Computer Programming）》I、II、III 卷，*中英*。凡发现书中错误的第一个读者都将得到作者亲笔签发的 2.56 美元的支票。Knuth 退隐后闲赋在家，江湖风传第 IV 卷马上出版，他正在写第 V 卷（计划一共写 VII 卷）。但愿你不要让 Knuth 的写作速度大于你的阅读速度。

Boost 源码笔记：Boost::multi_array

本文作者：谢轩

1. 动机

C++是一门自由的语言，允许你自由的表达自己的意图，对不对？ 所以我们既然可以 new 一个一维数组，也应该可以 new 出多维数组，对不对?先来看一个例子：

```
int* pOneDimArr = new int[10]; //新建一个 10 个元素的一维数组
pOneDimArr[0] = 0; //访问
int** pTwoDimArr = new int[10][20]; //错误!
pTwoDimArr[0][0] = 0; //访问
```

但是，很可惜，三四两行代码的行为并非如你所想象的那样——虽然从语法上它们看起来是那么“自然”。

这里的问题在于，new int[10][20]返回的并非 **int****类型的指针，而是 **int (*)[20]**类型的指针（这种指针被称为行指针，对它“+1”相当于在数值上加上一行的大小（本例为 20），也就是说，让它指向下一行），所以我们的代码应该像这样：

```
int (*pTwoDimArr)[20] = new int[i][20]; //正确
pTwoDimArr[1][2] = 0; //访问
```

注意 pTwoDimArr 的类型——int(*)[20]是个很特殊的类型，它不能转化为 int**，虽然两者索引元素的语法形式一样，都是“p[i][j]”的形式，但是访问内存的次数却不一样，语义也不一样。

最关键的问题还是：以上面这种朴素的方式来创建多维数组，有一个最大的限制，就是：除了第一维，其它维的大小都必须是**编译期确定**的。例如：

```
int (*pNdimArr)[N2][N3][N4] = new int[n1][N2][N3][N4];
```

这里 **N2,N3,N4** 必须都是编译期常量，只有 n1 可以是变量，这个限制与多维数组的索引方式有关——无论多少维的数组都是线性存储在内存中的，所以：

```
pTwoDimArr[i][j] = 0;
```

被编译器生成的代码类似于：

```
*( (int*)pTwoDimArr+i*20+j ) = 0;
```

20 就是二维数组的行宽，问题在于，如果允许二维数组的行宽也是动态的，这里编译器就无法生成代码（***20*** 所在的地方应该放什么呢？）。基于这个原因，C++只允许多维数组的第一维是动态的。

不幸的是，正由于这个限制，C++中的多维数组就在大多数情况下变成了有名无实的无用之物。我们经常可以在论坛上看到关于多维数组的问题，一般这类问题的核心都在于：如何模仿一个“完全动态的”多维数组。这里“完全动态”的意思是，所有维的大小都可以是动态的变量，而不仅是第一维。论坛上给出的答案不一而足，有的已经相当不错，但是要么缺乏可扩展性（即扩展到 N 维的情况），要么在访问元素的形式上远远脱离了内建的多维数组的访问形式，要么消耗了过多额外的空间。归根到底，我们需要的是一个类似这样的多维数组实现：

```
//创建一个 int 型的 DIMS 维数组，dim_sizes 用于指定各维的大小，即 n1*n2*n3
multi_array<int,DIMS> ma ( dim_sizes[n1][n2][n3] );
```

```
ma[i][j][k] = value; //为第 i 页 j 行 k 列的元素赋值
ma[i][j] = value; //编译错!
ma[i] = value; //编译错!
ma[i][j][k][l] = value;//编译错!
```

这样一个 *multi_array*，能够自动管理内存，拥有和内建多维数组一致的界面，并且各维的大小都可以是变量——正符合我们的要求。看起来，实现这个 multi_array 并非难事，但事实总是出乎意料，下面就是对 boost 中已有的一个 **multi_array** 实现的剖析——你几乎肯定会发现一些出乎意料的（甚至是令人惊奇的）地方。

2. Boost 中的多维数组实现——boost::multi_array

在 Boost 库中就有一个用于描述多维数组的功能强大的 MultiArray 库。它实现了一个通用、与标准库的容器一致的接口，并且具有与 C++中内建的多维数组一样的界面和行为。正是基于这种通用性的设计，MultiArray 库与标准库组件甚至用户自定义的泛型组件之间可以具有很好的兼容性，并能够很好的协同工作。

除此之外，MultiArray 还提供了诸如改变大小、重塑（reshaping）以及对多维数组的视图访问等极为有用的特性，从而使 MultiArray 比其它描述多维数组的方式（譬如：std::vector< std::vector<...> > ）更为便捷、高效。

下面我们就逐层深入，去揭开 boost::multi_array 的神秘面纱——对示例程序进行调试、跟踪是分析库源代码最有效的手段之一。我们就从 MultiArray 文档中的示例程序入手：

```
// 略去头文件包含
int main () {
  // 创建一个尺寸为 3×4×2 的三维数组
  #define DIMS 3 //数组是几维的
  typedef boost::multi_array<double,DIMS> array_type; // (1-1)
  array_type A(boost::extents[3][4][2]);  // (1-2)
  // 为数组中元素赋值
  A[1][2][0] = 120;   // (1-3)
  ... ...
  return 0;
}
```

在上述代码中，**boost::multi_array** 的两个模板参数分别代表数组元素的类型和**数组的维度**。**（1-2）**处是三维数组对象的构造语句。**boost::extents[3][4][2]**则用于指明该数组各维的大小，这里的含义是：定义一个 **3*4*2** 的三维数组。你肯定对 boost::extents 心存疑惑——为什么对它可以连用“[]”？如果用多了一个或用少了一个“[]”又会如何？下面我就为你层层剥开 boost::**extents** 的所有奥秘——

3. extents——与内建数组一致的方式

boost::extents 是一个**全局对象**，在 base.hpp 中：

```
typedef detail::multi_array::extent_gen<0> extent_gen;
... ...
multi_array_types::extent_gen extents; //注意它的类型!
```

可见 **extents** 的类型为 **extent_gen**，后者位于 extent_gen.hpp 中：

```
// extent_gen.hpp
template <std::size_t NumRanges>
class extent_gen {
    range_list ranges_; // (2-1)
... ...
extent_gen(const extent_gen<NumRanges-1>& rhs, const range& a_range)
{ // (2-2)
 ... //
}
extent_gen<NumRanges+1> operator[](index idx) //(2-3)
{ return extent_gen<NumRanges+1>(*this,range(0,idx)); }
    //返回一个 extent_gen，不过第二个模板参数增加了 1
};
```

boost::extent_gen 重载了 **operator[]**操作符，但是，既然这里只有一个“[]”，为什么我们可以写“extents[n1][n2][n3][...]呢？继续看——

如果把 **boost::extents[n1][n2][n3]**展开为操作符调用的方式就相当于：

```
boost::extents.operator[](n1).operator[](n2).operator[](n3);
```

boost::extents 对象的类型是 **extent_gen<0>** ——其 **NumRanges** 模板参数为 0。extents.operator[](n1) 调用（**2-3**），返回一个 **extent_gen<NumRanges+1>**，也就是 **extent_gen<1>**，由于这个类型也是 extent_gen，所以仍然可以对它调用 operator[](n2)，返回 **extent_gen<2>**，然后再对该返回值调用 operator[](n3)，最终返回一个 **extent_gen<3>**类型的对象。

这里，每一重 operator[]调用都转到（**2-3**），在那里将参数 idx 以 range 包装一下再转发给构造函数（**2-2**），注意此时调用的是 **extent_gen<NumRange+1>**类型的构造函数。至于 range(0,idx)则表示一个**[0,idx)**的下标区间。

总结一下 extents 的基本工作方式——每对它调用一次 operator[]，都会返回一个 **extent_gen<NumRange+1>**类型的对象，所以，对于 boost::extents[n1][n2][n3]，依次返回的类型是：

extent_gen<1> => extent_gen<2> => extent_gen<3>

最后一个也是最终的返回类型——**extent_gen<3>**。其模板参数 **3** 表示这是用于三维数组的下标指定。其成员 ranges_中，共有[0，n1)、[0，n2)、[0，n3)三组区间。这三组区间指定了我们定义的 multi_array

对象的三个维度的下标区间，值得注意的是这些区间都是前闭后开的（这和 STL 的迭代器表示的区间一样）。当 boost::extents 准备完毕后，就被传入 multi_array 的构造函数，用于指定各维的下标区间：

```
// multi_array.hpp
explicit multi_array(const extent_gen<NumDims>& ranges):
super_type((T*)initial_base_,ranges){
    allocate_space();   // （2-5）
}
```

这里，multi_array 接受了 ranges 参数中的信息，取出其中各维的下标区间，然后保存，最后调用 **allocate_space()**来分配底层内存。

4. 使用 extent_gen 的好处——do things right!

使用 **boost::extents** 作参数的构造过程和内建多维数组的方式一致，简练直观，语义清晰。首先，boost::extents 使用“[]”，能让人很容易想到内建多维数组的声明，也很清晰地表达了每个方括号中数值的含义——表明各维度的下标区间；最关键的还是，**使用 boost::extents，可以防止用户一不小心写出错误的代码**，例如：

```
multi_array<int,3> A(boost::extents[3][4][2][5]);
//错！不能用四维的下标指定来创建一个三维数组！
```

上面的语句是完全错误的，因为 mult_array 是个三维数组，而 boost::extents 后面却跟了四个“[]”，这显然是个错误。这个错误被禁止在编译期，这是由于在语法层面，multi_array<int,3>的构造函数只能接受 extent_gen<3>类型的参数，而根据我们前面对 extents 的分析，**boost::extents[3][4][2][5]** 返回的却是 extent_gen<4>类型的对象，于是就会产生编译错误。这种编译期的强制措施阻止了用户一不小心犯下的错误（如果你正在打瞌睡呢？），也很清晰明了地表达（强制）了语义的需求。

5. 另一种替代方案及其缺点

另外，还有一种声明各维大小的替代方式，就是使用所谓的 Collection Concept，例如：

```
// 声明一个 shape（“形状”），即各个维度的 size
boost::array<int,3> shape = {{ 3, 4, 2 }};
array_type B(shape);    //3*4*2 的三维数组
```

这种构造方式将调用 multi_array 的第二个构造函数：

```
// multi_array.hpp
```

```
template <class ExtentList>
explicit multi_array( ExtentList const& extents ) :
    super_type((T*)initial_base_,extents) {
        boost::function_requires< // (2-4)
detail::multi_array::CollectionConcept<ExtentList> >();
    allocate_space();   // (2-6)
}
```

这个构造函数的形参 extents 只要是符合 collection concept 就可以了——**shape** 的类型为 boost::array，当然符合这个 concept。这个构造函数的行为与接受 extents_gen 的构造函数是一样的——仍然是先取出各维的 range 保存下来，然后分配底层内存。至于**（2-4）**处的代码，则是为了在编译期静态检查模板参数 ExtentList 是否符合 Collection concept，实现细节在此不再赘述。

把这种方式与使用 extent_gen 的方式作一个简单的比较，很容易就看出优劣：采用这种方式，就不能保证编译期能够进行正确性的检查了，例如：

```
boost::array<int,4> shape = {{3,4,2,5}}; //一个四维数组的 shape
multi_array<int,3> A(shape); // 竟然可以通过编译！！
```

这里，用一个四维的 shape 来指定一个三维 multi_array 是不恰当的，但是却通过了编译，这是由于该构造函数对它的参数 extents 没什么特殊要求，只是把它作为一个普通的 collection 来对待，构造函数会根据自己的需要从 extents 中取出它所需要的各维下标区间——A 是三维数组，于是构造函数从 **shape** 中取出前三个数值作为 A 三个维度的下标区间，而不管 shape 究竟包含了几个数值。这样的语句在语义上是不清晰甚至错误的。但是既然这样的构造函数存在，设计者自然有他的道理，文档中就明确的表明，这个构造函数最大的用处就是编写维度无关（dimension-independent）的代码，除此之外 multi_array 库默认为前一种构造函数。

6. multi_array 的架构

无论采用哪一种构造函数，构造的流程都是相似的——将一系列下标区间传入基类的构造函数中去，基类构造完成之后就调用相同的 allocate_space()函数（见（2-5）和（2-6）处），allocate_space，顾名思义，用于为多维数组的元素分配空间。

然而，在这个看似简单的构造过程背后，却隐藏了两三层层次结构，每层的结构和功能都有所不同，担任了不同的角色，有些层次则能够单独拉出来复用于其他地方。

下面我们就来看看 multi_array 的架构到底长什么样 ☺ 首先，multi_array 继承自 **multi_array_ref**：

```
// multi_array_ref.hpp
template <typename T, std::size_t NumDims>
class multi_array_ref :  //multi_array 的基类！！
public const_multi_array_ref<T,NumDims,T*>
```

```
{
    typedef const_multi_array_ref<T,NumDims,T*> super_type;
    ... ...
    explicit multi_array_ref(T* base, //指向数组存储空间的指针
        const extent_gen<NumDims>& ranges): //下标区间
    super_type(base,ranges)    //把初始化的任务转发给基类（3-1）
    { }
    ... ...
};
```

而 **multi_array_ref** 又以 **const_multi_array_ref** 为基类：

```
// multi_array_ref.hpp
class const_multi_array_ref : //multi_array_ref 的基类!管理底层存储!
public multi_array_impl_base<T,NumDims>
{
    ... ...
    explicit const_multi_array_ref(TPtr base,
        const extent_gen<NumDims>& ranges) :
        base_(base), storage_(c_storage_order())    // （3-2）
        { init_from_extent_gen(ranges); }
    ... ...
    storage_order_type storage_;//支持多种存储策略！（3-3）

};
```

multi_array 的构造之路途经（**3-1**）处（multi_array_ref 的构造函数），延伸至（**3-2**）处（const_multi_array_ref 的构造函数）——这里看似一个终结，因为再没有参数传递给 const_multi_array_ref 的基类 multi_array_impl_base 了。但是心中还是疑惑：为什么会有如此多层的继承结构？这样的类层次结构设计究竟有什么玄机呢？

7. 多层继承的奥秘——访问界面&&复用性

转到基类 const_multi_array_ref 的声明，似乎可以看出一些端倪：

```
template< ... >
class const_multi_array_ref {
    ... ...
    //和所有的 STL 容器一致的迭代器界面！！
    const_iterator   begin() const;
    const_iterator   end() const;
    ... ...
```

```
    //和 std::vector 一致的元素访问界面！！
    const_reference  operator[](index i) const;
    ... ...
};
```

看到上面这些声明，是不是有些面熟？**STL**！对，这些成员函数的声明是与 STL 中 container concept 完全一致的。也就是说，这里提供了**与 STL 容器一致的访问界面**，所谓与 STL 的兼容性正是在这里体现出来了。而 **const_multi_array_ref** 更是“类如其名”，**const_multi_array_ref** 中所有访问元素、查询数组信息等成员函数都返回 const 的 reference 或 iterator。而反观 multi_array_ref 的声明，其中只比 const_multi_array_ref 多了访问元素、查询数组信息的对应的 non-const 版本成员函数。

那么 const_multi_array_ref 的基类 multi_array_impl_base 的职责又是什么呢？multi_array_impl_base 是属于实现细节的，它的作用只是根据数组信息（const_multi_array_ref 中的成员变量）计算偏移量、步长等，也就是把多维的下标最终转化为一维偏移量。而 multi_array_impl_base 的基类——value_accessor_**n** 或者 value_accessor_**one**—的功能则是提供一个对原始数据的访问。这种访问方式是大家所熟悉的把多维索引转化为一维偏移量的方式——凡是写过用一维数组模拟多维数组的人都应该清楚。至于为什么要有 value_accessor_**n** 和 value_accessor_**one** 两个版本，后文会详细阐释。

至此，multi_array 的大致架构就已经浮现出来了：

multi_array -> **multi_array_ref** -> **const_multi_array_ref** -> multi_array_impl_base -> value_accessor_**n**/value_accessor_**one**

其中每一层都担任各自的角色：

- multi_array ： 为数组元素分配空间，将各种操作转发至基类。
- multi_array_ref ： 提供与 STL 容器一致的数据访问界面。也可以独立出来作为一个 adapter 使用。
- const_multi_array_ref ： 提供 const 的 STL 数据访问界面。也可以作为一个 const adapter 使用。
- multi_array_impl_base 及其基类 ：最底层实现，提供一组对原始数据的基本操作。

这种架构看似复杂，却提供了很高的复用性，其中的**(const_)multi_array_ref** 类都可以独立出来作为一个 adapter 使用——例如：

```
int a[24]; //一维的 10 个元素数组，位于栈上！
//把一维数组 a“看成”一个 3*4*2 的三维数组：
multi_array_ref<int,3> arr_ref(a,boost::extents[3][4][2]);
arr_ref[i][j][k] = value; //和 multi_array 一样的使用界面
```

很简单吧！从此你就不用辛辛苦苦的去模拟多维数组了。即使是位于栈上的一维数组，也可以把它“看成”是一个多维数组。倘若你不想让 multi_array 来自动分配内存的话，你可以自行分配数组（可以位于栈上或堆上）然后用 **multi_array_ref** 把它包装成一个多维的数组。

8. multi_array 的存储策略

接下来，就来看看 multi_array 的存储策略，例如：C 风格的多维数组存储方式是按行存储，而 fortran 恰恰相反，是按列存储，甚至，用户可能有自己的存储策略要求。那么，如何支持多种风格的存储策略呢？秘密就在于代码（**3-3**）处，const_multi_array_ref 的成员 **storage_** ——其类型为 storage_order_type，下面的声明指出了 storage_order_type 的“本来面目”——**general_storage_order<NumDims>**：

```
// multi_array_ref.hpp
    ... ...
    typedef general_storage_order<NumDims> storage_order_type;
    ... ...
    // storage_order.hpp
    template <std::size_t NumDims>
    class general_storage_order {
    general_storage_order(const c_storage_order&){ // (4-1)
        for (size_type i=0; i != NumDims; ++i)
        { ordering_[i] = NumDims - 1 - i; }
        ascending_.assign(true); //缺省为升序排列
    }
    ... ...
    boost::array<size_type,NumDims> ordering_; //各维的优先顺序
    boost::array<bool,NumDims> ascending_;//升序排列还是降序排列
    };
```

在（**4-1**）处的构造函数中，ordering_和 ascending_是两个数组，缺省情况下，当函数（4-1）执行完毕后，ordering_中的元素应当是{NumDims-1, NumDims-2,...1,0}，也就是说，NumDims-1 维在先，然后是 NumDims-2 维，...，如果将这些元素作为各维度存储顺序的标识——具有较小 ordering_值的维度优先——那么就这和 C 语言中的存储方式就完全一致了，而 **ascending_**勿庸置疑就是用来表明各维度是否升序存储。其实 general_storage_order 还有一个模板构造函数，它是为了支持更为一般化的存储策略（例如 fortran 的按列存储或用户自定义的存储策略）。这里不作详述。

除了存储策略，**const_multi_array_ref** 的构造还通过调用 init_from_extent_gen 函数，将 extents 中的内容取出来进行处理，并以此设定其它若干表述多维数组的变量（（3-3）处其它一些变量），具体细节不再赘述。

现在关于一个多维数组的所有信息都已经准备齐备，可谓“万事具备，只欠‘空间’”。multi_array 下面要做的就是调用前面提到的 **allocate_space** 来为数组中的元素分配空间了。

```
// multi_array.hpp
void allocate_space() {
    ... ...
    base_ = allocator_.allocate(this->num_elements(),no_hint);
    ... ...   std::uninitialized_fill_n(base_,allocated_elements_,T());
```

```
}
```

原来，在底层，存储仍然是退化为一维数组的存储： allocate_space 使用 allocator_分配一块连续的空间用以存储元素，其中 **num_elements()**返回的就是数组**各维度的大小的乘积**，即**数组的总元素个数**。分配完之后，就将首指针赋给表述数组基地址的成员 base_，然后 **std::uninitialized_fill_n** 负责把该数组进行缺省初始化。至此 multi_array 的构造工作终于大功告成了。

9. 一致性界面——GP 的灵魂

multi_array 的另一重要特性就是支持与内建多维数组相同的访问方式，也就是说，multi_array 支持以连续的“[]”来访问数组元素。就以文章开头给出的示例代码（**1-3**）处的赋值语句为例，让我们看看 multi_array 是如何支持这种与内建数组兼容的访问方式的。

```
// multi_array_ref.hpp
// 使用 operator[]来访问元素,返回类型 reference 是什么呢？不是 T&!
reference operator[](index idx) {
    return super_type::access(boost::type<reference>(),
              idx,origin(),this->shape(),this->strides(),
              this->index_bases());
}
```

这个调用转入了 value_accessor_n::access(...)之中：

```
// base.hpp
// in class value_accessor_n
template <typename Reference, typename TPtr>
Reference access(boost::type<Reference>,
              index idx,TPtr base,const size_type* extents,
              const index* strides,const index* index_base)
{
     TPtr newbase = base + idx * strides[0];
     return Reference(newbase,extents+1,
                 strides+1,index_base+1);
}
```

这个连续调用 operator[]的过程和 extend_gen 是很类似的——每调用一层就返回一个“proxy(替身)”，这个替身并非实际元素，因为这时候后面还跟有“[]”，所以对这个替身也应该能调用 operator[]，从而使这个过程能够连续下去直到后面没有“[]”为止”。

举个例子，如果以 A[x1][x2][x3]方式访问 A 中的元素，就相当于

```
A.operator[x1].operator[x2].operator[x3] //连续调用“[]”
```

这三次operator[]调用返回的类型依次为：

```
sub_array<T,2> -> sub_array<T,1> -> T&
```

注意 **sub_array** 的第二个模板参数（维度）递减的过程，当递减到 0 时，表明已经递归到了最后一个维度，真正的访问开始了，所以最后一次调用“[]”返回的恰好是对元素的引用（这就刚好印证了前面所说的——当且仅当“[]”的个数和数组的维数相同的时候，才能够取出元素，否则你得到的返回值要么还是 **sub_array<...>**（而不是元素），要么会由于试图在 T&上继续调用“[]”而编译失败）那么，这一切究竟是如何做到的呢？

从上面的递归过程，我们可以轻易看出：真正对元素进行访问并返回 **T&**的任务交给了 **sub_array<T,1>**。那么这个 sub_array 到底是个什么东东？

10.sub_array 的秘密

sub_array 的定义在 sub_array.hpp 中：

```
// sub_array.hpp
template <typename T, std::size_t NumDims>
class sub_array : public const_sub_array<T,NumDims,T*>;

template <typename T, std::size_t NumDims, typename TPtr>
class const_sub_array :
        public multi_array_impl_base<T,NumDims>;
//base.hpp
template <typename T, std::size_t NumDims>
class multi_array_impl_base:public
value_accessor_generator<T,mpl::size_t<NumDims> >::type ;
```

唔，原来 sub_array 最终继承自 **multi_array_impl_base**，后者的基类型由 **value_accessor_generator** 来生成——它会根据 **NumDims** 的不同而生成不同的基类型：

```
// base.hpp
template <typename T, typename NumDims>
struct value_accessor_generator {
    ... ...
    typedef typename  //如果 NumDims 为 1，则类型为 value_accessor_one
    mpl::apply_if_c<(dimensionality == 1),
        choose_value_accessor_one<T>,
        choose_value_accessor_n<T,dimensionality>
    >::type type; //把这个类型作为 multi_array_impl_base 的基类!
};
```

很显然，如果 **dimensionality == 1**，那么“**::type**”就是 value_accessor_**one<T>**，而只有对 value_accessor_**one** 使用“[]”才能返回 **T&**，否则，“**::type**”被推导为 value_accessor_**n**，这

只是个“proxy”而已，对它运用“[]”只会返回 sub_array<T,**NumDims-1**>，从而可以对它继续这个连续调用“[]”的过程，直到 **dimensionality** 降为 **1**，sub_array 的基类才变成了 value_accessor_**one**，这时候再使用“[]”就会返回 **T&!**

11.取出元素

根据上面的分析，取元素的任务最终交给 value_accessor_**one**，其成员函数 **access** 如下：

```
template <typename Reference, typename TPtr>
Reference access(boost::type<Reference>,index idx,TPtr base,
          const size_type*,const index* strides,
          const index*) const {
   return *(base + idx * strides[0]); //终于取出了数据!
}
```

这里，access 的返回类型 **Reference** 就是 **T&**，即数组中元素类型的引用，从而可以将指定元素的引用返回，达到访问数组元素的目的。看到这里，multi_array 以内建数组访问方式访问数组元素的过程基本已经弄清楚了，至于其中一些细节，尤其是计算地址的细节，譬如：偏移量的计算、步长的使用等，皆已略去了。

现在也许你会有这样的疑惑：以内建多维数组的“[][][]...”访问方式来访问 multi_array 的元素的能力真的如此重要吗？费这么大力气、写这么多代码还不如以多参数的方式重载 operator[]呢（“[i,j,...]”形式）！这么大代价真的值得吗？值得！这是勿庸置疑的。以内建数组访问方式访问 multi_array 元素的能力最重要的表现就是：可以把 multi_array 作为一个内建多维数组来对待，**与内建类型的一致性是你的类能够和泛型算法合作的关键**。举个例子，用户编写了一个函数模板：

```
template <typename ReturnType, typename _3DArray>
ReturnType someAlgo(_3Array& mda){//可以作用于内建多维数组
... ...
   mda[x][y][z] = mda[z][x][y]; //对内建多维数组的访问形式!
  ... ...
}
```

因为有了以内建数组访问方式访问 multi_array 元素的能力，这个 someAlgo 算法就**可以同时应用在内建数组和 multi_array 上**（否则用户就得为 multi_array 提供一个单独的重载版本），如此一来，代码的可重用性、可扩展性都大大提高了。

12.效率

效率是 C++永恒的主题，MultiArray 库也不例外。执行时间效率上，纵观 MultiArray 库对数组元素的访问代码，虽然函数调用嵌套层数甚多，但多数调用都是简单的转发或者编译期的递归，在一个成熟的现代 C++编译器下，这些转发的函数调用代码应该可以很轻易地被优化掉，所以在效率上几乎没有什么损失。在空

间方面，由于大量运用模板技术，基本能够在编译期决定的内容都已决定，没有为运行期带来不必要的空间上的负担。总的看来，Boost.MultiArray 库的确是难得的高效又通用的多维数组的实现。

13.结语

本文只是将 multi_array 最基本的功能代码做了一个扼要的分析，正如文章开始所说，multi_array 还有许多很有用的特性，如果读者想充分了解 multi_array 的运作机制与实现技巧，就请深入完整地分析 multi_array 的代码吧，相信一定会大有收获的！

词法分析自动生成器 CScanGenner 的设计与实现

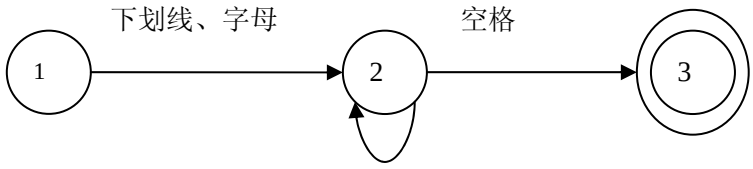
哈尔滨工业大学
谢煜波（iamxiaohan@126.com）

现代编译器设计的一个趋势是尽量使用各种自动生成工具来实现编译器的各个部份，其中词法分析器通常是编译器的第一个部份，并且词法分析也广泛应用与各种程序中。现在常用的词法分析器程序有两种：ScanGen 以及 Lex。

ScanGen 是一种表驱动形式的词法分析器，它最后产生的结果不是一个实实在大的词法分析器程序，而是一张转换表，通过这种转换表去驱动真正的词法分析器的运行。举个例子，我们先来想一下我们自己是怎样识别一个单词串的，比如说识别一个 C 语言的变量名吧：

1. 我们首先看第一个符号是不是下划线或者字母，如果是的话我们就把这个字符加入输出缓冲区，并进入下一步。
2. 继续查看紧跟着的这个符号是不是下划线、字母或数字，如果是我们就把这个字符加入输出缓冲区，并重复这一步，如果是空格我们就进入第三步。
3. 表明变量名定义完成，把输出缓冲区中的内容返回，这就是一个被识别出来的变量名。

上面这个步骤可以用如下的一幅图来表示：



上面这张图，我们就称之为状态转换图，用语言可以如下描述：

- 在状态是 1 的情况下如果当前输入字符是下划线或字母就转到下一状态 2。
- 在状态是 2 的情况下如果当前输入字符是下划线、字母、数字就转到下一状态 2。
- 在状态是 2 的情况下如果当前输入字符是空格则转到下一状态 3。
- 在状态是 3 的情况下，把输出缓冲区的内容返回，表明一个单词串被识别出来。

从上面的文字描述中我们可以看出，其实决定词法分析器怎么运作的有两个关键因素，一个因素是当前状态，另一个因素是当前的输入字符。现在，我们把当前状态做为行索引，当前字符做为列索引，就得到一张如下的表：

输入 / 状态	下划线	字母	数字	空格
1	进入状态 2	进入状态 2		
2	进入状态 2	进入状态 2	进入状态 2	进入状态 3
3				

上面这张表就被视为词法分析器的驱动表，这样，我们的词法分析器就有如入一样的很简单的结构：

```
DRIVER_TABLE M[][];
string buffer;
char ch;
STATE s = 1; // 初始状态为 1
while(input.get(ch))
```

```
{
    s = M[s][ch]; // 进入下一状态
    if(s == 2)
{
    buffer += buffer;
}
else if(s == 3)
{
    return buffer;
}
else
{
    printf( “ERROR!”);
}
}
```

从上面可见，由于有那张词法分析器的驱动表的存在，使得我们的词法分析程序变得非常简单。由于 ScanGen 生成的结果就是那张驱动表，因此，这还带来另外一个好处，也就是词法分析器的主体程序可以用任何语言书写，而不像 Lex，Lex 最后生成的结果是实实在在的词法分析器的 C/C++程序，因而你不用其它语言来使用 Lex。当然 ScanGen 也有一个弱点，就是转换表所含的信息相对较少，因此，这限制了词法分析器的随意度，它不像 Lex 那样灵活。

ScanGen 最早是由微斯康星大学的一个教授写的，版权属于微斯康星大学，用的语言是 Pascal，它可以把用户书写的词法说明文件（这在后面会详细论述）转换成一张驱动表。CScanGenner 可以算是 ScanGen 的一个 C++版本，它用 C++语言书写的，并且取消了原 ScanGen 对词法说明文件的一些限制（比如说最大规模，单词串最大长度等），另外，在词法说明文件的描述上支持 C 风格而不是原来的 Pascal 风格（最大的一点变化就是大小写相关），另外，CScanGenner 是开源软件，你可以在 MIT 许可协议[1]下随意使用。

本文不是 CScanGenner 的说明文档，只是描述一下他在实现上的原理及一些实现中的细节问题，关于 CScanGenner 的详细说明，你可以访问纯 C 论坛：http://purec.binghua.com 或者 http://iamxiaohan.nease.net 取得。

单词串其实都是通过一种称之为正则表达式的东东定义的，或说可以用正则表达式来描述，比如上面那个例子就可以用如下的一个正则表达式来对其进行描述：

ID := (下划线|字母).(下划线|字母|数字)*;

上面这个就是一个正则表达式，“|”表示“或”关系，“.”表示“连接”关系，“*”表示出现零次或多次，上面的意思就是：下划线或者字母后面，出现零次或多次下划线或者字母或者数字，这个符号串被定义成一个 ID（也就是我们常说的变量名）。当然，正则表达式还有“+”：表示出现一次或多次；“？”表示出现零次或一次……等等运算符，有兴趣的可以在网上搜索一下，这里就不详细描述了。

还记得那前面那幅状态转换图么？它同上面的正则表达式其实都是描述的同样一个东东，这就使我们不得不怀疑它们之间是否存在一种转换关系？实际上，形式语言的理论告诉我们：正则表达式是可以转换成为一张转换图的（有限状态机）。我们现在就来看看一这种转换。

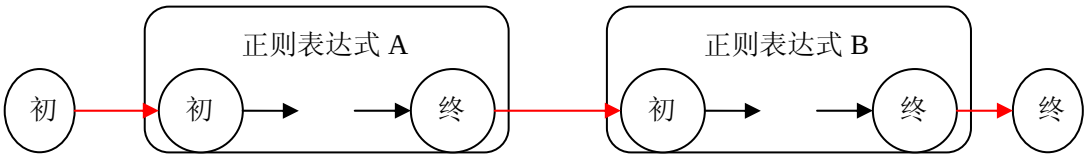
正则表达式其实有三种最基本的关系：

- 一是连接关系：

C = A.B

它可以用如下的方式转换：

[1] MIT 许可协议由 BSD 协议发展而来，与 BSD 协议几乎完全相同，并且取消了广告条款的限制。简单来说就是只要声明你的项目源于谁，你基本上就可以完全自由的处理所获得的资源，包括商业使用。详细内容可以查看：
http://www.opensource.org/licenses/mit-license.php

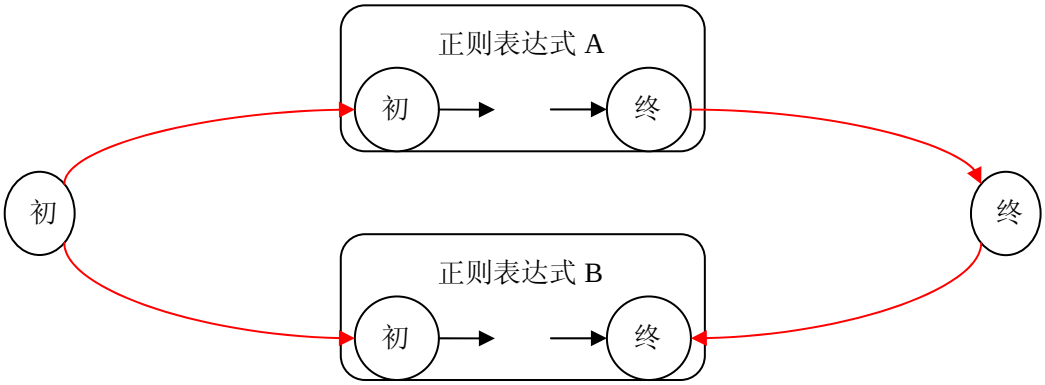


这就是正则表达式 C 所对应的状态图，红色线表示输入可以是 epsilon，即可以为空，什么都不做就能非常自由的从一个状态由红线进入到下一个状态。

- 二是“或”关系：

C = A|B

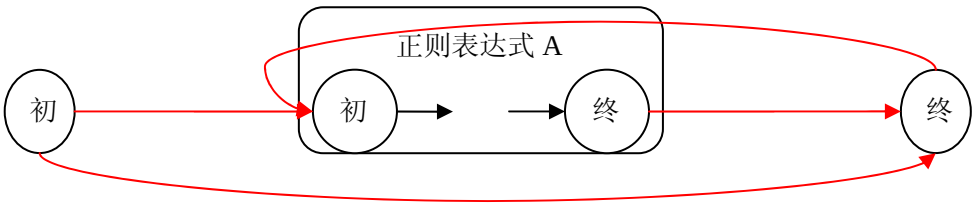
它可以用如下的方式转换：



三是“*”关系：

C = A*

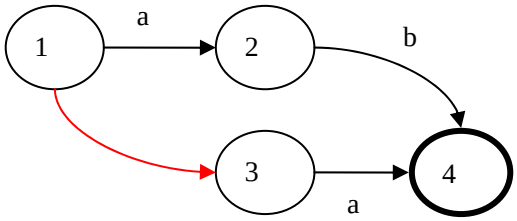
它可以用如下的方式转换：



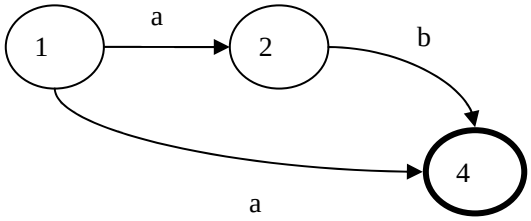
另外的关系都可以由这三种关系合成，比如“+”关系：

A+ = A.A*

上面种种表明，我们能找到一种方法，把一个描述词法的正则表达式转换成一张状态表（有限自动机），而状态表可以很简单的转换为我们的二维表（如前所示），这样我们的驱动表就可以产生出来了。不过上面的状态表有这样一个问题：假如存在如下的一张状态表：



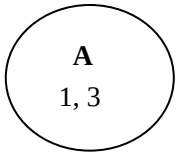
上面这个自动机其实等价于：ab|a，但现在问题来了，在状态 1 下，输入 a 本来应当是到状态 2，但是由于状态 1 有一根红线到状态 3，即在状态 1 下，可以不输入任何字符自由的到状态 3，故而，可以是状态 1 先到状态 3 然后在根据输入的 a 到状态 4，也就是说在状态 1 下，面对输入 a，可以到状态 2 也可以到状态 4，如下图一般：



如果是这样的话，那么在状态表中，就会出现一个表格中填入了两个动作，一个动作是到状态 2，另一个动作是到状态 4，这对于词法扫描程序来说是一个无法处理的事情，因为它不知道是应当按照动作一，还是应按照动作二进行。

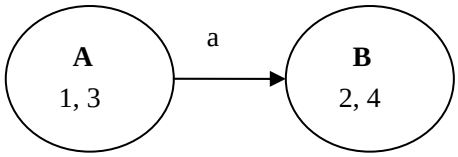
上面的状态图反应的一种状态机称之为不确定有限状态机（NDFA），而词法扫描程序处理的必须是确定有限状态机（DFA），因此，必须采用一种算法把一个 NDFA 转换成一个 DFA，有一种算法非常常用，这就是所谓的子集化算法。

首先，我们从状态 1 开始，对状态 1 求它的 epsilon 闭集，即求得所有可以由状态 1 通过红线（即无需输入字符，或说输入字符为空（epsilon））所能达到的状态的集合。在上例中就是：状态 1，状态 3。这是需要注意的是，在求的时候应当是一个递归的算法，即状态 1 能通过 epsilon 到状态 3，那么应当把状态 3 加入到这个集合中来，随后还要求状态 3 可以通过 epsilon 所达到的状态 n，这个 n 也应当加入集合中，因为状态 1 可以不需要输入字符就能达到状态 3，而状态 3 不需要输入字符就能达到状态 n，相当于状态 1 可以不需要输入字符就能达到状态 n，故而，状态 n 也就加入集合之中。现在我们可以开始构造一个新的状态图：



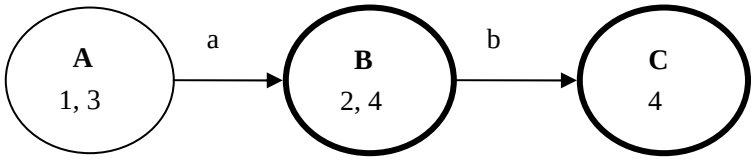
生成了一个新状态 A，它含有原状态 1，3。

下面我们考查这个新状态 A 中的原状态，状态 1 输入是 a 时，会转到状态 2，原状态 3 输入是 a 时会转到状态 4，于是我们可以构造一个新的状态 B，让它含有状态 2 及状态 4，如下所示：



这时我们就得到了一个确定的东东，即在状态 A 下输入 a 时，只能到状态 B，这就是一个确定的 DFA。在实际的求解过程中，我们还应对 B 的每个元素就它的 epsilon 闭集，即，如果 2 或者 4 能通过不输入任何字符到达状态 m，那么 m 也应被加入到状态 B 中。

继续我们的算法，当 B 中的状态 2 遇到输入 b 后，会转到状态 4，因此，我们还需构造一个新的状态 C，让它含有状态 4：



由于原来的 NDFA 中，状态 4 是终态，也就是说：在 NDFA 中，如果状态 4 出现，即表明当前的输入串（被放在输出缓冲区中的东东）应当被接收（将输出缓冲区中的东东返回）。现在在 DFA 中，对于状态 B 来说，它含有 NDFA 中的终态 4，但是也含有非终态 2，那么这个输出缓冲区中的东东应不应当返回呢？这就需要由后面的输入决定了，如果后面的输入是 b，说明它不应当返回，它还应转到状态 C 去，因为这其实是到达了原 NDFA 中的，现在 DFA 中的状态 B 中的状态 2，如果后面的输入是其它字符，则说明它应当返回了，这其实是到达了原 NDFA 中的，现在 DFA 中的状态 B 中的状态 4，真正的终态。这里还可以发现一个很有兴的现象：正则表达式其实是一种最大正向匹配算法，它总是尽力的尝试去匹配更多的输入。

上面描述的其实是形式语言的一些原理性的东东，在任一本形式语言的书或者某些编译原理的书中都会有较为详细的论述，这里只是简单的介绍了一下，用以说明 CScanGenner 所采用的算法原理，如果对这部份内容想详细了解的朋友，可以参阅相关资料。

上面介绍的就是 CScanGenner 实现的算法基理，总结一下，说简单一点，就是由用户根据需要用正则表达式编写词法说明文件，然后，CScanGenner 根据这些说明文件，把说明文件中的正则表达式先转换成一个一个的 NDFA，再把这一个一个的 NDFA 合在一起当成一个由“或”关系组成的巨大的 NDFA，随后对这个巨大的 NDFA，应用前述的子集化算法转化成一个巨大的 DFA，最后再把这个 DFA 转换成一张驱动表输出，至此，CScanGenner 的工作完成，用户可以使用这张驱动表驱动自己的词法驱动程序。下面我们就来看看这其间的一些实现上的细节问题，当然，由于篇幅有限（最主要的还是认为没有必要），笔者不会完全描述所有的细节，只是对笔者觉得有意义的地方进行一些阐述。其中若有不当之处，欢迎您来信指教，笔者的电子邮箱是：xieyubo@126.com。

CScanGenner 所采用的分析方法是最简单的递归下降法，它对每一个语法项的处理都有一个函数进行，关于 CScanGenner 的语法定义及递归处理函数请参见 CScanGenner 的源代码。如果你对递归下降的语法处理方法还不太熟悉，请参见本文参考文献一的前两章，它用一个很小而很简单的例子说明了什么是递归下降，怎么编写递归下降的分析程序。这里就不详细描述 CScanGenner 是怎么使用递归下降的，主要来看看它是怎么进行从正则表达式到 NDFA 再到 DFA 的转换的。

首先我们来看看一个只含有两个词法描述，即两个正则表达式的例子：

```
Slash          = '\\';
LineTernimator = 10, 13;
Letter         = 'a'..'z', 'A'..'Z';
UnderLine      = '_';
```

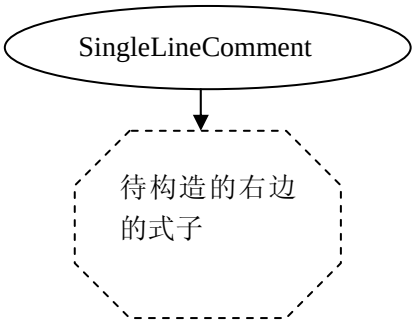
Digit = '0'..'9';

SingleLineComment = Slash . Slash . NOT(LineTerminator)*;

Identifier = (Letter,UnderLine).(Letter,UnderLine,Digit)*

上面蓝色部份就是正则表达式，而深红色部份是对字符所属类别的定义。正则表达式定义了两个在 C 语言中常用的词法符号，一个就是单行注释，另一个就是前面已经描述过的标识符。现在我们来主要看看单行注释这个正则表达式是怎么被 CScanGenner 所翻译的。

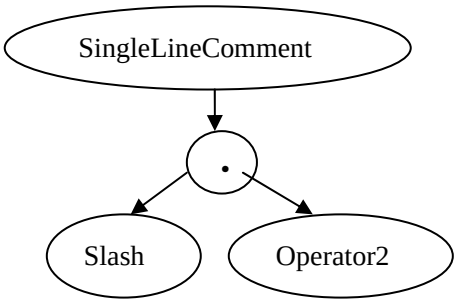
CScanGenner 在用递归下降法扫描这个正则表达式的时候会边扫描边构造出它的语法树。比如，先扫描了 SingleLineComment，于是把这个构造为一个树根：



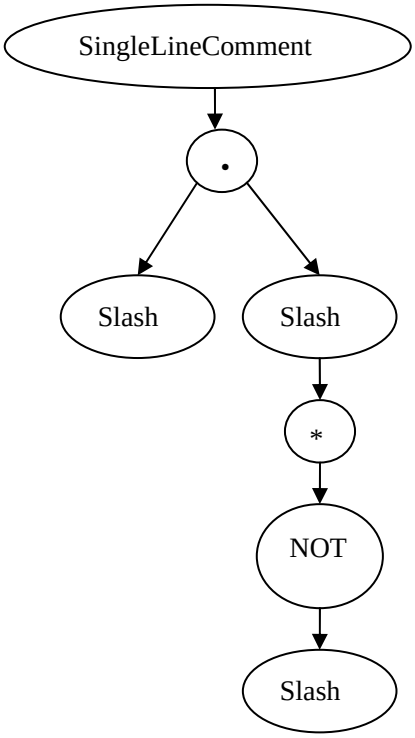
随后，它用一个函数对整个右边式子进行处理，而这个函数将返回一个指针，由根结点保留住这个指针，类似下面的代码：

```
RegularExp.pointer = HandleRightExp();
```

而在 HandleRightExp()中，它又会用递归的方式处理它的每个小部份，比如：整个右部会被视为：Slash . Operator2，如下所示：



其中的 Operator2 就是“Slash . NOT(LineTerminator)*”，随后，系统又会调用函数对 Operator2 进行处理，如此一步一步的递归下去，最后就得到了如下一棵语法树：

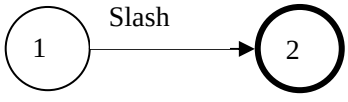


这样，一根语法树就构造出来了，并且，在这棵语法树上，我们保存了这个正则表达式的所有信息，CScanGenner 就是这样，把用户输入的正则表达式首先全部转换成了这样的语法树保存在了内存中。

随后，CScanGenner 开始将正则表达式转换成 NDFA，也即它每棵语法树都转换成一个 NDFA。

对于“.”这个连接操作而言，要想生成它的 NDFA（还记得前面描述的将正则表达式转换成向应的状态图的方法吗？）必须先要生成它左右两个操作数的 NDFA，然后再把这两个 NDFA 连接起来，因此在这里 CScanGenner 采用的是“后根遍历”算法来遍历整个语法树，并在遍历的同时生成对应的 NDFA，由于一个正则表达式就对应一个语法树，因此，当整个语法树都转换成一个 NDFA 的时候，这个正则表达式也就转换成了其相应的 NDFA。

在前面我们已前描述了怎样把由“.”（连接）、“*”（闭集）、“|”（或）关系组合的两个 NDFA 合成一个 NDFA，现在我们来看一看对于叶结点（如上图的“Slash”结点）怎么转换成一个 NDFA。其实这非常简单，让我们用下面一幅图来说明：



上图中还有一个 NOT（非）关系，它可以是如下的格式：

NOT(A, B, C, …)

对于上面这种“非”关系，在 CScanGenner 中是把它转换成一个大的“|”（或）关系来处理的，首先，它把所有的类别都加入一个集合中，然后，再把 NOT 括号中涉及到的类别从集合中删除，这样就得到了一个许可集合，然后，按照“|”（或）关系，把这个集合中所有的元素（当然，每个元素会以产生“Slash”的 NDFA 一样的方式先产生一个 NDFA）产生的 NDFA 组合成一个 NDFA。

现在我们来看看 CScanGenner 中是用怎样一种结构来表示一个 NDFA 的。从状态转换图上我们可以直观的看到一个 NDFA 由若干个状态（即图中的小圈）组成，每一个状态还有一定数量的转换边，而每一个转换边上还绑定着一个输入字符，并且，每个转换边还指出了通过它可以达到的状态。

在 CScanGenner 中，首先定义了一个名为：struct_transEdge 的结构来表示转换表：

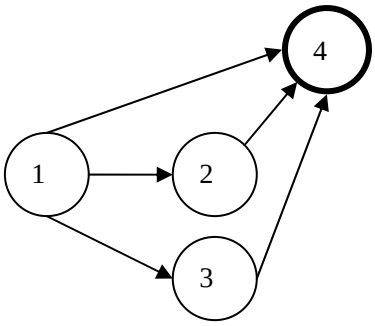
```
struct struct_transEdge
{
    int classNum;
    int nextState;
    enum_tossSaveFlag tossSaveFlag;
};
```

其中 classNum 指明了绑定在这条转换边上的输入字符，nextState 指明了这条转换边指向的状态，tossSaveFlag 这个标志在后面会有详细描述，这里暂时不用理会它。

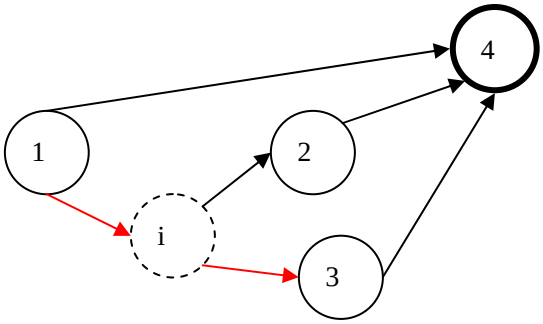
接着，CScanGenner 定义了一个名为：struct_triple 的结构来表示 NDFA 中的一个状态：

```
struct struct_triple
{
    int transEdgeNum;
    struct_transEdge transEdge1;
    struct_transEdge transEdge2;
};
```

从这个结构上我们可以看见，在 CScanGenner 中的每个 NDFA 的状态最多含有两个转换边，这主要是为了方便处理，那两条转换边是不是就足够了呢？如果遇见下面的情况怎么办：

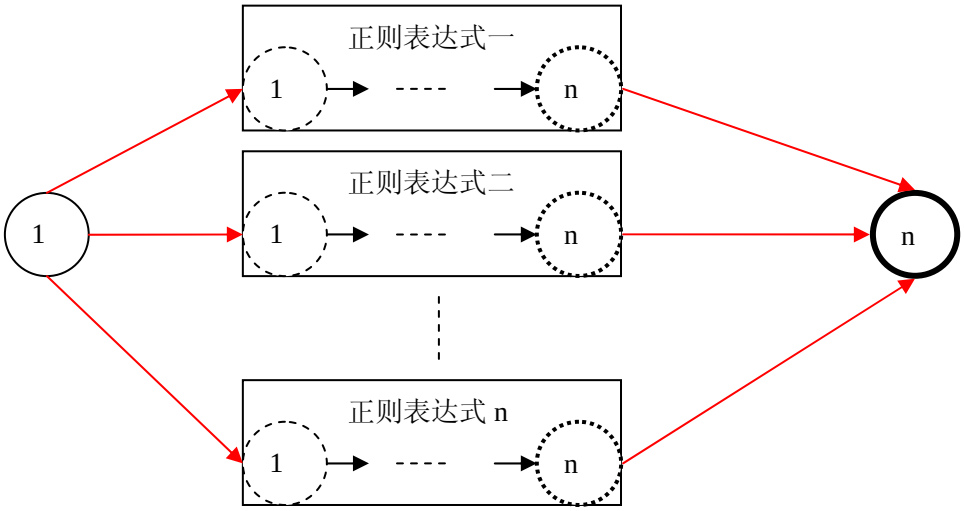


这种情况其实可以加入 epsilon 转换边（即前面的状态图中所使用的红色边，它表示可以不需要任何输入即能在两个状态之前自由转换）及辅助状态来保证每个状态最多有两个转换边，如上面的状态图可转换为：



一个正则表达示所对应的 NDFA，其实就是一个 struct_triple 元素组成的数组，这样我们就把用户输入的正则表达式转换成了一个一个的 NDFA，也即一个一个的由 struct_triple 元素组成的数组。

接下来，我们就应当把所有的这些 NDFA 转换成一个 DFA 了。系统把所有的这些 NDFA（每一个正则表达式对会产生一个 NDFA）看做是由或关系组成的一个巨大的 NDFA，如下图所示：

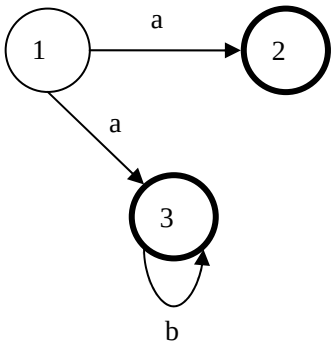


然后，用本文开头所描述的子集化方法，把所有的 NDFA 转换成了 DFA，最后，把这 DFA 输出为一张状态表，CScanGenner 的任务也就完成了。

上面描述的东东就是整个 CScanGenner 的核心算法及实现方式，至于更详细的内容请参看 CScanGenner 的源代码，其中有很详细的注释。

下面我们来谈谈这其中可能会遇到的一些问题，及 CScanGenner 对之的处理方式。对于这些问题，如果您有更好的处理方式的话，请来信告诉我。

第一个问题，我们来看看如果用户有两个正则表达式可以匹配一个输入，比如对于正则表达式：a|ab*，它的状态图如下所示：



在这种情况下，如果我们仍用前面的子集化方法，那么得到的 DFA 的状态转换图就如下所示：

现在问题出来了，看上图的第二个状态，它含有原 NDFA 中的状态 2、3。在原来的 NDFA 中，状态 2、3 都是终态，因此上图的第二个状态所对应的终态到底应当是原 NDFA 中状态 2 所对应的终态还是状态 3 所对应的终态呢？这就起了冲突，也即，如果用户的词法描述文件中有类式的情形：

TOKEN A = a;

TOKEN B = ab*;

那么，CScanGenner 在最后产生出的 DFA 中就会发现有一个状态包含了两个 NDFA 中的终态，这就说明一个输入可以被两个正则表达式进行匹配，对于这种情况，CScanGenner 会直接输出错误信息。

还有一个问题是由{TOSS}指示符引起的。CScanGenner 继承了原 ScanGen 的一个特点，就是可以使用一个称之为{TOSS}的指示符，这个指示符会指示当前的输入字符不被加入到输出缓冲区中，这在某些时候是非常有用的，比如，下面一个正则表达式：

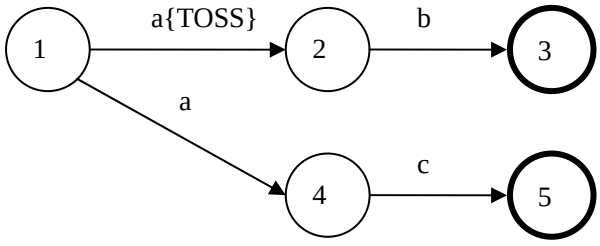
```
Slash = '\';
Quote = '"';

TOKEN A = Slash{TOSS} . Quote;
```

上面这个正则表达式可以匹配一个转义符与引号连用的情况(\")，在这种情况下，由于 Slash 后面有{TOSS}因此，Slash 符号(\)就不被加入输出缓冲区中，这样最后返回给用户的 TOKEN 就是单纯的 Quote(")了。再比如在对字符串就是匹配时，我们可以在正则表达式中对开头与结尾的两个引号后加上{TOSS}标志，这样只有字符串的内容可以被返回给用户串，而字符串开头与结尾的引号就不会返回给用户了。

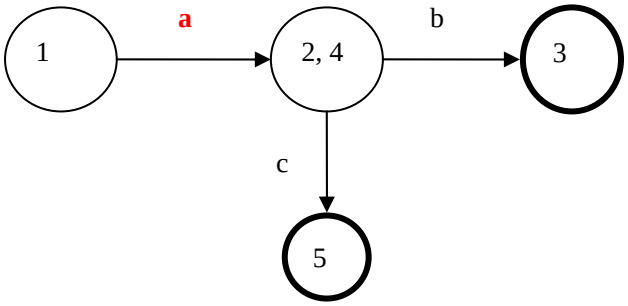
但是{TOSS}标记符的滥用也会导致一些问题，比如看下面的一个正则表达式：

a{TOSS}b|ac

它的意思是如果输入是 ab 的话就不返回前面的 a，如果是 ac 的话就原样返回 a（不进行 TOSS 操作），这个正则表达式的状态图如下所示：



现在我们用子集化的方式把它转换成 DFA：



问题就出来了，对于那个 a，我是应当加入{TOSS}还是不应当加入{TOSS}呢？这就起了冲突。对于出现这种情况，目前的 CScanGenner 的解决之道就是简单的将其做为一种错误给报告出来。

这里还需要把前面的一个遗留问题提一下，在支持{TOSS}指示符后，每个转换边又多了一个属性来保存这个转换边上是否使用了{TOSS}指示符，还记得前面描述转换边的结构体：`struct_transEdge` 中有一个 `tossSaveFlag` 标志吗？

CScanGenner 是一个开源的软件，目前它已经几个项目中实际运用了，但是他还不是一个很完善的东东，比如说它最后生成的 DFA 还没有优化，不是最简 DFA，另外前面所谈到的问题的解决方式还比较初级。非常欢迎您能对 CScanGenner 进行开发与完善，也欢迎您在您的项目中自由的使用这个工具（当然，需要您遵守 MIT 许可协议:），你可能访问 http://purec.binghua.com 或者 http://iamxiaohan.nease.net 获得关于它的全部源代码及其余资源。

基于 Flex 的 c/c++代码加亮工具

本文作者: Tinyfool (tinyfool@gmail.com)

摘要：本文介绍作者 Tinyfool 用 Flex 构建的一个 C/C++代码加亮工具，关键字参考了 Editplus 的语法加亮格式文件，这个加亮工具并不是完全 Ansi C/C++支持的，但是因为仅仅是一个加亮工具，所以这点也不是很大的问题，而且由于提供了代码，使用者可以随时随地的自己添加关键字。

我是一个非科班出身的软件开发者，所以，我是没有正式学过编译原理的。在我的职业生涯里面多次想学，但是总有人告诉我那东西没有什么用处 （也是啊！难道我要写个编译器和 VC 竞争么？），所以我一直没有学。然而后来很多事情改变了我的看法，一方面是正则表达式的广泛用途，另外也是我的虚拟机 项目，我想在里面用自己做的编译器。还有就是我的前同事 LEE 做的虚拟机的广泛使用等等，让我感受到不管什么技术，总有用武之地。今天凌晨 0 点多的时候，我又在看《编译原理技术与工具（龙书）》和《编译原理和实践》，但是光看书不实践等于零，所以，我想我是不是可以做一个代码加亮工具呢？说做就做，到了清晨 5 点多的时候，我写完了。下面是代码：

1.1 htmlpre.lex 文件，进行预处理把一些在 HTML 需要转码的代码先转码：

```
%{
#include <stdio.h>
int yywrap();
%}

%%
"<"             {printf("<");}
">"             {printf(">");}
\t              {printf("        ");}
" "             {printf(" ");}
               {printf(" ");}
%%

int main()
{
        yylex();
        return 0;
}
int yywrap()
{
        return 1;
}
```

1.2 cpp.lex 文件，进行代码加亮：

```
%{
/*
#
*/
#include <stdio.h>
#define FALSE        0
#define TRUE         1
int yywrap();
%}
DIGIT               [0-9]
NUMBER              {DIGIT}+
LETTER              [a-zA-Z]
WORD                {LETTER}+
WHITESPACE          [ \t]+
DELIMITER           [,(){}[]-+*%/="'~!&|<>?:;.#]
NL                  \r?\n
QUOTATION           \"[^"\n]*\"
KEYWORD             "auto"|"bool"|"break"|"case"|"catch"|"char"|"cerr"|"cin"|"class"|
"const"|"continue"|"cout"|"default"|"delete"|"do"|"double"|"else"|"enum"|"explicit"|
"extern"|"float"|"for"|"friend"|"goto"|"if"|"inline"|"int"|"long"|"namespace"|"new"|
"operator"|"private"|"protected"|"public"|"register"|"return"|"short"|"signed"|"sizeof"|
"static"|"struct"|"switch"|"template"|"this"|"throw"|"try"|"typedef"|"union"|"unsigned"|
"virtual"|"void"|"volatile"|"while"|"__asm"|"__fastcall"|"__based"|"__cdecl"|"__pascal"|
"__inline"|"__multiple_inheritance"|"__single_inheritance"|"__virtual_inheritance"
PREWORD             "define"|"error"|"include"|"elif"|"if"|"line"|"else"|"ifdef"|"pragma"|
"endif"|"ifndef"|"undef"|"if"|"else"|"endif"
PREDEF              "#"{PREWORD}
LINECOMMENT         "\/\/".*\n
%%
"/*"            {
                  char c;
                  int done=FALSE;
                  printf("<span class=\"comment\">\n");
                  ECHO;
                  do
                  {
                        while((c=input())!='*')
                        {
                              if(c=='\n')
```

```
                                printf("<br/>\n");
                        else
                                putchar(c);
                }
                putchar(c);
                while((c=input())=='*')
                        putchar(c);
                if(c=='\n') printf("<br/>\n");
                putchar(c);
                if(c=='/') done=TRUE;
            } while(!done);
            printf("</span>\n");
        }
{LINECOMMENT}   {printf("<span class=\"comment\">%s</span><br/>\n",yytext);}
{QUOTATION}     {printf("<span class=\"quotation\">%s</span>",yytext);}
{PREDEF}        {printf("<span style=\"predef\">%s</span>",yytext);}
{KEYWORD}       {printf("<span class=\"keyword\">%s</span>",yytext);}
{NL}            {printf("<br/>\n");}
{WORD}          {ECHO;}
{NUMBER}        {ECHO;}
{WHITESPACE}    {ECHO;}
%%

int main()
{
        printf("<html>\n");
        printf("<head>\n");
        printf("<link href=\"mycpp.css\" rel=\"stylesheet\" type=\"text/css\">\n");
        printf("</head>");
        printf("<body>\n");
        yylex();
        printf("</body>");
        printf("</html>");
        return 0;
}
int yywrap()
{
        return 1;
}
```

1.3 test.cpp 文件，用来测试的加亮效果的小程序：

```
/*
 * just test my code lighting tools
 * by tinyfool
 * 2005-04-15
 *
 * */
#include <stdio.h>
int main()
{
   char x[]="xxx";
   //printf(x);
   return 0;
}
```

上面的代码就是利用这个工具自己做出来的 Html 文件截图的，源代码包里面包含了这些生成的 Html 文件。具体使用方法参看包内的批处理文件，需要系统安装 VC（也可用其他支持 Ansi C 的编译器，但是需要修改批处理），包内有一个我自己编译的 Win 版本的 Flex。

时间仓促，写得远非完美（字符串中的转码还不支持），但是也让我有了非常大的成就感，我会继续改进这个东西的。代码中的谬误和不良也请读者不吝赐教。

作者简介：

网名：Tinyfool

擅长的技术领域：无，什么都沾。

目前的工作动态：嵌入开发。

个人主页：http://www.tinydust.net/

个人 Blog：http://blog.tinydust.net/diary/ (Tinyfool 的开发日记)

E-mail：tinyfool@gmail.com

解读经典样本 EICAR

本文作者：SwordLea(@bbs.hit.edu.cn)

```
X5O!P%@AP[4\PZX54(P^)7CC)7}$EICAR-STANDARD-ANTIVIRUS-TEST-FILE!$H+H*
```

这段代码就是 EICAR，它是欧洲计算机防病毒协会和反病毒软件厂家一同开发的测试文件，其特征码已经广泛包含在各种杀毒软件的病毒代码库里，所以可以用来测试病毒扫描引擎是否在正常工作。

EICAR 是段很经典的代码，虽然使用了 int 21h（机器码 cd 21）和 int 20h（机器码 cd 20）调用，但并没有因为 cd 是不可显示字符而影响这段代码作为文本格式的广泛传播。下面用 debug 解读一下 EICAR，可以看到作者的一番良苦用心。

注意：PUSH 和 POP 指令的成对出现相当于 MOV 指令，但 MOV 指令机器码一般为 Bxh，位于不可显示字符的区间。

```
13A4:0100 58          POP  AX
13A4:0101 354F21       XOR AX,214F
13A4:0104 50          PUSH AX
13A4:0105 254041       AND AX,4140 ; AX = 140h
13A4:0108 50          PUSH AX
13A4:0109 5B          POP  BX     ; BX = 140h
13A4:010A 345C         XOR AL,5C   ; AX = 11Ch
13A4:010C 50          PUSH AX
13A4:010D 5A          POP  DX     ; DX = 11Ch
13A4:010E 58          POP  AX     ; AX = 214Fh
13A4:010F 353428       XOR AX,2834 ; AX = 097Bh, AH = 09(int 21h 的 09 子功能)
13A4:0112 50          PUSH AX
13A4:0113 5E          POP  SI     ; SI = 097Bh
13A4:0114 2937         SUB [BX],SI ; [140] = CD 21 (int 21h)
13A4:0116 43          INC  BX
13A4:0117 43          INC  BX
13A4:0118 2937         SUB [BX],SI ; [142] = CD 20 (int 20h)
13A4:011A 7D24         JGE  0140    ; 跳到已经成为 cd 21 cd 20 的 地方
13A4:011C 45          INC  BP     ; 字符串 EICAR-STANDARD-ANTIVIRUS-TEST-FILE!$
13A4:011D 49          DEC  CX     ; $ 为输出结束标志
13A4:011E 43          INC  BX
13A4:011F 41          INC  CX
13A4:0120 52          PUSH DX
13A4:0121 2D5354       SUB AX,5453
13A4:0124 41          INC  CX
13A4:0125 4E          DEC  SI
13A4:0126 44          INC  SP
13A4:0127 41          INC  CX
```

```
13A4:0128 52          PUSH DX
13A4:0129 44          INC  SP
13A4:012A 2D414E       SUB AX,4E41
13A4:012D 54          PUSH SP
13A4:012E 49          DEC  CX
13A4:012F 56          PUSH SI
13A4:0130 49          DEC  CX
13A4:0131 52          PUSH DX
13A4:0132 55          PUSH BP
13A4:0133 53          PUSH BX
13A4:0134 2D5445       SUB AX,4554
13A4:0137 53          PUSH BX
13A4:0138 54          PUSH SP
13A4:0139 2D4649       SUB AX,4946
13A4:013C 4C          DEC  SP
13A4:013D 45          INC  BP
13A4:013E 2124         AND [SI],SP
13A4:0140 48          DEC    AX  ; 代码执行完毕这里会成为 int 21h 和 int 20h
13A4:0141 2B482A       SUB    CX,[BX+SI+2A] ; 分别是字符串输出(AH=9)和退出
```

AS86 的 MAN

作者：hold(http://blueline.hit.edu.cn/blog/hold)

as86 的资料实在是少之又少。翻译了 man 文档....不过翻译的很烂，就当练英文水平吧:) Linus 为什么要用它来写 boot 程序呢，nnd..翻译真是一项辛苦的工作啊...特别是对翻译的东西还不了解的时候....

as86(1) as86(1)

名称

as86 - as86-8086..80386 处理器的汇编程序

概要格式

as86 [-0123agjuw] [-lm[list]] [-n name] [-o obj] [-b[bin]] [-s sym]
[-t textseg] src
as86_encap prog.s prog.v [prefix_] [as86 options]

描述

as86 是 8086..80386 处理器下的汇编程序，它所采用的语法与 Intel/MS 采取的语法类似，而不同于广泛运用于 UNIX 下的汇编语法（译注,gas 中的语法，AT&T 汇编）

命令行中的 src 参数可为'-',代表对标准输入进行汇编。

as86_encap 是一个脚本，使用了 as86 汇编程序，并且把生成的二进制文件转为一个 C 文件 prog.v，用于被连接或者包含到程序里，例如引导块安装程序。prefix_参数定义一个加到源文件中所有定义的变量的前缀，缺省前缀是源文件名。...

选项

-0 以 16 位代码段运行，当使用了高于 8086 指令集的指令时警告。
-1 以 16 位代码段运行，当使用了高于 80186 指令集的指令时警告。
-2 以 16 位代码段运行，当使用了高于 80286 指令集的指令时警告。
-3 以 32 位代码段运行，不对任何指令发出警告信息（就算使用了 486 或 586 的指令）
-a 使汇编程序部分兼容于 Minix asld.交换了[]与()的用法，并且改变了一些 16 位跳转与调用的语法("jmp @(bx)" 就成了一个合法的指令)
-g 仅仅把 global 符号写入目标或者符号文件中
-j 把所有短跳转指令(译注：8 位跳转称为短跳转)换成相似的 16 位或者 32 位跳转。并且把 16 位条件转移指令换为一个条件短转移命令与一个无条件长跳转组合
-O 汇编程序会做几遍额外的工作，以尝试支持向前引用。最多 30 遍。不推荐使用
-l 产生清单文件(list file),文件名写在选项后
-m 把宏展开后写在清单文件里

-n 把模块名写在选项之后(目标模块，而非源文件)
-o 生成目标文件，文件名写在选项之后
-b 生成纯二进制文件，文件名写在后面。这是一个没有头部的纯二进制文件（译注：类似 Dos 下的 com 和 sys）如果没有-s 选项程序将会在内存地址 0 处开始执行

-s 生成一个 ASCII 码符号文件，文件名写在选项后。很简单就能将其转换，用于与-b 选项生成的二进制文件相关联和封装。如果二进制文件不从地址 0 处开始执行。那么符号文件表中前两项分别代表起始地址与结束地址

-u 假定未定义符号在未指定的段中被导入了

-w- 允许汇编程序输出警告信息

-t n 把所有 text 段的数据放到段 n+3 中.

AS86 资料

特殊字符

* 本行起始地址

;或! 注释起始符,另外，在一行起始处的“unexpected”字符被认为是注释（但是仍然会被显示在终端上）

$ 16 进制数的前缀, C 风格的前缀，比如 0x1234，也可以使用.

% 2 进制数的前缀.

立即数的前缀.

[ ] 间接寻址运算符.

与 MASM 不同，汇编程序没有标识符的类型信息，每个标识符仅仅代表是一个段地址和偏移地址。[]与立即数操作与传统汇编程序一致

例:

```
mov ax,bx
jmp bx
```

寄存器寻址, jmp 指令把 bx 寄存器中的值拷到程序计数器中

```
mov ax,[bx]
jmp [bx]
```

简单的寄存器间接寻址, jmp 指令把 bx 寄存器值指向的内存单元的值拷到程序计数器中

```
mov ax,#1234
```

立即数，把 1234 赋值给 ax 寄存器

```
mov ax,1234
mov ax,_hello
mov ax,[_hello]
```

直接寻址，内存地址 1234 处的存储字赋给 ax 寄存器。注意第三个指令并不十分严格，只是为了与 asld 保持兼容所以保留（译注：若想将_hello 标识符表示的值作为立即数使用，需要加上#前缀 #_hello）

```
mov ax,_table[bx]
mov ax,_table[bx+si]
mov eax,_table[ebx*4]

mov ax,[bx+_table]
mov ax,[bx+si+_table]
mov eax,[ebx*4+_table]
```

变址寻址。两种形式都可以，但是我认为第一种要更正确些，但是我往往用第二种形式:)

条件判断

IF, ELSE, ELSEIF, ENDIF

数字比较

IFC, ELSEIFC

字符串比较 (str1,str2)

FAIL .FAIL
生成用户错误

段相关

.TEXT .ROM .DATA .BSS
设置当前段。可以在前面加上关键字.SECT

LOC 数字表示段 0=TEXT, 3=DATA,ROM,BSS, 14=MAX. 连接器设定的段顺序现在是 0,4,5,6,7,8,9,A,B,C,D,E,1,2,3.段 0 以及所有 3 以上的段都假设为 text 段。注意 64K 限制对 3-14 的段不适用。

标识符类型定义

EXPORT PUBLIC .DEFINE
导出符号

ENTRY 强制连接器在 a.out 文件里包含这个特殊符号

.GLOBL .GLOBAL
将一个标识符定义为外部的，并且强制就算不使用，也必须导入

EXTRN EXTERN IMPORT .EXTERN
导入外部标识符列表

NB: bin 格式的文件不支持外部变量（译注：关于这些格式，推荐参考一下 NASM 的手册。纯 C 论坛上有中文的 NASM 手册）

.ENTER 标识出旧式 bin 格式(obs)的程序入口

数据定义

DB .DATA1 .BYTE FCB
1 字节的对象列表

DW .DATA2 .SHORT FDB .WORD
2 字节的对象列表

DD .DATA4 .LONG
4 字节的对象列表

.ASCII FCC
写到输出的 Ascii 码字符串.

.ASCIZ Ascii 写到输出的 Ascii 码字符串，末尾添加 nul

空间定义

.BLKB RMB .SPACE
以字节为单位计算空间

.BLKW .ZEROW
以字为单位计算空间 (一字 2 字节)

COMM .COMM LCOMM .LCOMM
通用数据域定义

其他实用伪指令

.ALIGN .EVEN
对齐

EQU 定义标识符（译注：可参考 NASM 或者 MASM 的 EQU）

SET 定义可重定义的标识符

ORG .ORG
定义汇编位置（译注：即设置地址计数器的值，建议参考 MASM 的资料）

BLOCK 定义汇编位置并且把原来的汇编位置入栈

ENDB 回到刚才栈里记录的汇编位置

GET INCLUDE
插入新文件 (no quotes on name)
USE16 [cpu]
定义默认操作数大小为 16 位，参数表示程序代码将会运行在什么样的 CPU 的(86,186, 286,386,486,586)指令集上.使用了指定指令集之上的指令会产生警告信息
USE32 [cpu]
定义默认操作数大小为 32 位，参数表示程序代码将会运行在什么样的 CPU 的(86,186, 286,386,486,586)指令集上.使用了指定指令集之上的指令会产生警告信息
END 标识出本文件停止汇编的地方
.WARN 警告信息开关
.LIST 清单 on/off (1,-1)
.MACLIST
宏清单 on/off (1,-1)
宏的使用形式如下

```
MACRO sax
  mov ax,#?1
MEND
sax(1)
```

未实现/未使用的

IDENT Define object identity string.
SETDP Set DP value on 6809
MAP Set binary symbol table map number.

寄存器

BP BX DI SI
EAX EBP EBX ECX EDI EDX ESI ESP
AX CX DX SP
AH AL BH BL CH CL DH DL
CS DS ES FS GS SS
CR0 CR2 CR3 DR0 DR1 DR2 DR3 DR6 DR7
TR3 TR4 TR5 TR6 TR7 ST

操作数类型说明

BYTE DWORD FWORD FAR PTR PWORD QWORD TBYTE WORD NEAR
near 和 far 关键字并没有提供段间寻址编程的能力,所有”far”操作都是
都是通过显式地使用以下指令得到的：指令: jmpi, jmpf, callf, retf,
等等. Near 关键字可以被用来强制使用 80386 的 16 位条件跳转指令.
'Dword'和'word' 能控制远跳转和远调用的操作数的大小

普通指令.

这些指令和其他 8086 汇编程序所提供的指令大体上差不多,(译注:后面的
看不明白了.我的英语功底啊~555) the main exceptions being a few '
Bcc' (BCC, BNE, BGE, etc) instructions which are shorthands f
or a short branch plus a long jump and 'BR' which is the longest
unconditional jump (16 or 32 bit).

长分支

BCC BCS BEQ BGE BGT BHI BHIS BLE BLO BLOS BLT BMI BNE BPC BPL

BPS BVC BVS BR

段间操作

CALLI CALLF JMPI JMPF

段修饰符指令

ESEG FSEG GSEG SSEG

字节操作指令

ADCB ADDB ANDB CMPB DECB DIVB IDIVB IMULB INB INCB MOVB MULB NEGB NOTB ORB OUTB RCLB RCRB ROLB RORB SALB SARB SHLB SHRB SBBB SUBB TESTB XCHGB XORB

标准指令

AAA AAD AAM AAS ADC ADD AND ARPL BOUND BSF BSR BSWAP BT BTC BTR BTS CALL CBW CDQ CLC CLD CLI CLTS CMC CMP CMPS CMPSB CMPSD CMPSW CMPW CMPXCHG CSEG CWD CWDE DAA DAS DEC DIV DSEG ENTER HLT IDIV IMUL IN INC INS INSB INSD INSW INT INTO INVD INVLPG INW IRET IRETD J JA JAE JB JBE JC JCXE JCXZ JE JECXE JECXZ JG JGE JL JLE JMP JNA JNAE JNB JNBE JNC JNE JNG JNGE JNL JNLE JNO JNP JNS JNZ JO JP JPE JPO JS JZ LAHF LAR LDS LEA LEAVE LES LFS LGDT LGS LIDT LLDT LMSW LOCK LODB LODS LODSB LODSD LODSW LODW LOOP LOOPE LOOPNE LOOPNZ LOOPZ LSL LSS LTR MOV MOVS MOVSB MOVSD MOVSW MOVSX MOVW MOVZX MUL NEG NOP NOT OR OUT OUTS OUTSB OUTSD OUTSW OUTW POP POPA POPAD POPF POPFD PUSH PUSHA PUSHAD PUSHF PUSHFD RCL RCR REP REPE REPNE REPNZ REPZ RET RETF RETI ROL ROR SAHF SAL SAR SBB SCAB SCAS SCASB SCASD SCASW SCAW SEG SETA SETAE SETB SETBE SETC SETE SETG SETGE SETL SETLE SETNA SETNAE SETNB SETNBE SETNC SETNE SETNG SETNGE SETNL SETNLE SETNO SETNP SETNS SETNZ SETO SETP SETPE SETPO SETS SETZ SGDT SHL SHLD SHR SHRD SIDT SLDT SMSW STC STD STI STOB STOS STOSB STOSD STOSW STOW STR SUB TEST VERR VERW WAIT WBINVD XADD XCHG XLAT XLATB XOR

浮点

F2XM1 FABS FADD FADDP FBLD FBSTP FCHS FCLEX FCOM FCOMP FCOMPP FCOS FDECSTP FDISI FDIV FDIVP FDIVR FDIVRP FENI FFREE FIADD FICOM FICOMP FIDIV FIDIVR FILD FIMUL FINCSTP FINIT FIST FISTP FISUB FISUBR FLD FLD1 FLDL2E FLDL2T FLDCW FLDENV FLDLG2 FLDLN2 FLDPI FLDZ FMUL FMULP FNCLEX FNDISI FNENI FNINIT FNOP FNSAVE FNSTCW FNSTENV FNSTSW FPATAN FPREM FPREM1 FPTAN FRNDINT FRSTOR FSAVE FSCALE FSETPM FSIN FSINCOS FSQRT FST FSTCW FSTENV FSTP FSTSW FSUB FSUBP FSUBR FSUBRP FTST FUCOM FUCOMP FUCOMPP FWAIT FXAM FXCH FXTRACT FYL2X FYL2XP1

一段代码的分析

谢煜波 (iamxiaohan@126.com)

请看一段代码：

```
 1:  M[0] := 0
 2:  read(M[1])
 3:  if M[1] >= 0 then goto 5
 4:  goto 7
 5:  M[3] := M[0] - M[1]
 6:  if M[3] >= 0 then goto 16
 7:  writeln(M[1])
 8:  read(M[2])
 9:  M[3] := M[2] - M[1]
10:  if M[3] >= 0 then goto 12
11:  goto 14
12:  M[3] := M[1] - M[2]
13:  if M[3] >= 0 then goto 8
14:  M[1] := M[2] + M[0]
15:  goto 3
16:  halt
```

其中 read(a), 是指从输入读入一个数并放在 a 中, writeln(a),是指把 a 中的数输出到屏幕；halt 表示停止。

上面这段代码是从一本书上看来的，格式与得很好，但还是比较难读，这再一次印证了选择一个易于理解的算法远比良好的格式在可读性方面的贡献大的多。下面我们就来分析分析这段代码。

分析代码也有一些比较基本的方法，当然最常见的就是一条语句一条语句的跟踪执行了，这时需把自己想成一台电脑。但是这种方法是很初级的，当程序中 goto 语句乱飞的情况下，这种分析法会非常吃力的，对于程序我觉得我们还是应当先有一种整体上的把握比较好。

对于上面这段代码，我们先全部浏览一下，可以发现，M[0]除了在第一次赋值之后，后面再也没有为 M[0]赋过值了，所以，上面这段代码中 M[0]的值永远是 0，因此，我们把 M[0]的值完全去掉，代码也就简化成了下面这样：

```
 1:
 2:  read(M[1])
 3:  if M[1] >= 0 then goto 5
 4:  goto 7
 5:  M[3] := -M[1]
 6:  if M[3] >= 0 then goto 16
 7:  writeln(M[1])
 8:  read(M[2])
 9:  M[3] := M[2] - M[1]
10:  if M[3] >= 0 then goto 12
11:  goto 14
12:  M[3] := M[1] - M[2]
13:  if M[3] >= 0 then goto 8
14:  M[1] := M[2]
15:  goto 3
16:  halt
```

再观察，我们可以发现，M[3]这人值从来只在赋值号的左边出现，而没在赋值号的右边出现过，因此，这也是一个在程序中的独立变量，即它不会对其它变量的值产生影响，因引，这种情况，我们也能把它给去掉。程序进一步简化为：

```
 1:
 2:  read(M[1])
 3:  if M[1] >= 0 then goto 5
 4:  goto 7
 5:
 6:  if (-M[1]) >= 0 then goto 16
 7:  writeln(M[1])
 8:  read(M[2])
 9:
10:  if M[2] - M[1] >= 0 then goto 12
11:  goto 14
12:
13:  if M[1] - M[2] >= 0 then goto 8
14:  M[1] := M[2]
15:  goto 3
16:  halt
```

现在剩下的语句也就越来越少了，我们继续分析。

看第 3 句，如果 M[1]大于等于 0 就跳到第 5 句，第 5 句马上又测试-M[1]如果大于等于 0，就跳到 16 句。我们想想如果一个数的及其相反数都大于等于 0，那么这个数是不是只能为零？因此，我们从这里可以知道第 3 句、以及第 5 句都是在测试 M[1]是否为零，因此，我们可以把这两句合成一句，程序再一步被简化为：

```
 1:
 2:  read(M[1])
 3:  if M[1] == 0 then goto 16
 4:
 5:
 6:
 7:  writeln(M[1])
 8:  read(M[2])
 9:
10:  if M[2] - M[1] >= 0 then goto 12
11:  goto 14
12:
13:  if M[1] - M[2] >= 0 then goto 8
14:  M[1] := M[2]
15:  goto 3
16:  halt
```

继续分析，看第 10 句，这是在检测 M[2]是否大于等于 M[1]，如果大于等于就跳到第 12 句，第 12 句开始马上就检测 M[1]是否大于等于 M[2]，如果是就跳到第 8 句。我们想想如果 M[1]大于等于 M[2]，同时 M[2]也大于等于 M[1]，是不是只可能 M[1]与 M[2]这两个数相等？因此，我们从这里可以知道，第 10 句及第 11 句，其实都是在测试 M[1]与 M[2]是否相等。所以，程序还可以简化为：

```
 1:
 2:  read(M[1])
 3:  if M[1] == 0 then goto 16
 4:
 5:
 6:
 7:  writeln(M[1])
 8:  read(M[2])
 9:
10:  if M[2] == M[1] then goto 8
11:
12:
13:
14:  M[1] := M[2]
15:  goto 3
16:  halt
```

现在程序就非常好读了，我们把它转写成我们比较熟悉的，没有 goto 语句的 c 语言风格。

```
read(M[1]);
while(M[1] != 0)
{
   writeln(M[1]);
   read(M[2]);
   while(M[2] == M[1])
   {
      read(M[2]);
   }
   M[1] = M[2];
}
```

现在，我想每个人都能看懂这个程序了，它其实的完成的工作就是读入一串用户输入的数据，而把相邻的重复的数据去掉（不输出），只输出相邻的不重复的数据。

也许会有人认为，这样的分析是没有实际用处的，其实不然，我们如果把最后的那段 C 语言用编译器编译成机器码，然后再反汇编成汇编语言，可能就与我们最开始的那段程序一样。这是在做逆向工程的时候常常用到的。有时候我们自己能把高级语言翻译成机器语言，但有时候我们也需要能把机器语言反译成高级语言。从高级语言到机器语言，有很多的现成的工具软件可以做，但从机器语言到高级语言，这在绝大多数的情况下就只能靠我们自己的脑子了。

虽说只是一段小小的程序，但是从可读性因素，逆向工程等许多方面，我们都能看出不少的问题。

投稿指南

《纯 C 论坛杂志》编辑部成立于公元 2004 年 9 月 28 日。2005 年 1 月开始，杂志更名为《CSDN 社区电子杂志——纯 C 论坛杂志》。新的一年，新的开始，我们有幸和 CSDN 社区电子杂志的编辑一同合作，一同打造属于 CS 人的杂志。本刊定位为相对底层、较为纯粹的计算机科学及技术的研究，着眼于各专业方向基本理论、基本原理的研究，重视基础，兼顾应用技术，希望以此形成本刊独特的技术风格。

本刊目前为双月刊，于每单月 28 号通过网络发行。任何人均可从 **CSDN 电子社区电子杂志**(http://emag.csdn.net)或**纯 C 论坛网站**(http://purec.binghua.com)下载本刊。读者不需要付费。

目前本刊有十大骨干技术版块：

栏目	责任编辑	投稿信箱
计算机组成原理及体系结构	kylix@hitbbs	purec@126.com
编译原理	worldguy@hitbbs	
算法理论与数据结构	xiong@hitbbs	
计算机语言（C/C++）	sun@hitbbs hitool@hitbbs	
汇编语言	ogg@hitbbs	
数据库原理	pineapple@hitbbs	
网络与信息安全	true@hitbbs	
计算机病毒	swordlea@hitbbs	
人工智能及信息处理	car@hitbbs	
操作系统	iamxiaohan@hitbbs	

本刊面向全国、全网络公开征集各类稿件。你的投稿将由本刊各栏目的责任编辑进行审校，对于每一稿件我们都会认真处理，并及时通知您是否选用，或者由各位责任编辑对稿件进行点评。

所有被本刊选用的稿件，或者暂不适合通过电子杂志发表的稿件，将会在**纯 C 论坛网站**上同期发表。所有稿件版权完全属于各作者本人所有。非常欢迎您积极向本刊投搞，让你的工作被更多的人知道，让自己同更多的人交流、探讨、学习、进步！

为了确保本刊质量，保证本刊的技术含量，本刊对稿件有如下一些基本要求：

1．主要以原创（包括翻译外文文献）为主，可以不需要有很强的创新性，但要求有一定的技术含量，注重从原理入手，依据原理解决问题。描述的问题可以很小但细致，可以很泛但全面，最好图文并貌，投稿以 Word 格式发往各栏目的投稿信箱，或直接与各栏目责任编辑联系。本刊各栏目均为活动性栏目，会随时依据稿件情况新开或暂定各栏目，因此，只要符合本刊采稿宗旨的稿件，本刊都非常欢迎，投稿请寄本刊通用联系信箱（purec@126.com）。

2．本刊对稿件的风格或格式没有特殊要求，注重质量而非形式，版权归作者本人所有。不限一稿多投但限制重复性投稿（如果稿件没有在本刊发表，则不算重复性投稿）。稿件的文字、风格除了在排版时会根据需要有必要的改动及改正错别字外，不会对稿件的描述风格、观点、内容、格式进行大的改动，以期最大限度的保留各作者原汁原味的行文风格，因此，各作者在投稿时最好按自己意愿自行排版。也可以到本刊网站（http://purec.binghua.com）或者 CSDN 社区电子杂志(http://emag.csdn.net)下载稿件模板。

3．来稿中如有代码实现，在投稿时最好附带源代码及可执行文件。本刊每期发行时，除了一个 PDF 格式

的杂志外，还会附带一个压缩文件，其中将包含本期所有的源代码及相应资源。是否提供源代码，由作者决定。

4．如果稿件是翻译稿件，请最好附带英文原文，以便校对。另外本刊在刊发翻译稿的同时，如有可能，将随稿刊发英文原文，因此请在投稿前确认好版权问题。

5．来稿请明确注明姓名、电子邮箱、作者单位等信息，以便于编辑及时与各位作者交流，发送改稿意见，选用通知及寄送样刊等，如果你愿意在发表你稿件的同时，提供一小段的作者简介，我们非常欢迎。

6．所有来稿的版权归各作者所有，也由各作者负责，切勿抄袭。如果在文中有直接引用他人观点结论及成果的地方，请一定在参考文献中说明。每篇稿件最好提供一个简介及几个关键词，以方便读者阅读及查询。由于本刊有可能被一些海外朋友阅读，所以非常推荐您提供英文摘要。

7．所有来稿编辑部在处理后，会每稿必复，如果您长时间没有收到编辑部的消息，请您同本编辑部联系。

8．由于本刊是纯公益性质，没有任何外来经济支持，所有编辑均是无偿劳动，因此，本刊暂无法向您支付稿筹，但在适当的时候，本刊会向各位优秀的作者赠送《纯 C 论坛资料(光盘版)》，以感谢各位作者对本刊支持！

9．如果您想转载（仅限于网络）本刊作品，请注明原作者及出处；如果您想出版本刊作品，请您与本刊编辑部联系。

本刊编辑部联系地址：

网址：http://purec.binghua.com
联系信箱：purec@126.com
通信地址：哈尔滨工业大学 计算机科学与技术学院 综合楼 520 室
邮编：150001

再一次诚恳邀请您加盟本刊，为本刊投稿！

《CSDN 社区电子杂志——纯 C 论坛杂志》编辑部
2005.1.28 修订