



图灵程序设计丛书

C程序员必读经典

原版畅销 11 年

「毒舌程序员」

为你揭开指针的
真实面纱

征服 指针

「日」前桥和弥 著
吴雅明 译



人民邮电出版社
POSTS & TELECOM PRESS

前桥和弥

1969年出生，著有《彻底掌握C语言》、《Java之谜和陷阱》、《自己设计编程语言》等。其一针见血的“毒舌”文风和对编程语言深刻的见地受到广大读者的欢迎。

作者主页：

<http://kmaebashi.com/>

吴雅明

13年编程经验。其中7年专注于研发基于Java EE和.NET的开发框架以及基于UML 2.0模型的代码生成工具。目前主要关注的方向有：Hadoop/NoSQL、HTML5、智能手机应用开发等。

译者序

在平时的工作中,我时常遇到两种人:一种是刚毕业的新人,问他们:“以前学过 C 语言吗?”他们大多目光游离,极端不自信地回答说:“学过,但是……”;另一种是做过几年 C 语言开发并自我感觉良好的人,他们大多可以使用指针熟练地操作字符数组,但面对菜鸟们提出的诸如“为什么数组的下标是从 0 而不是从 1 开始”这类“脑残”问题时,总是不耐烦地回答道:“本来就是这样嘛。这是常识,你记住了就行!”(可本来为什么是这样的呢?)

本书的作者不是大学老师,更不是那些没有写过几行程序的学究,而是一位至今还工作在开发一线的程序员(在国内,工作了 5 年的你如果还在做“码农”,肯定会坐立不安了吧)。他带给大家的不是教科书中死板的说教,而是十多年经验沉淀下来的对无数个“脑残”问题的解答。在这本书初版面世的 11 年后,我在东京一个大型书店的 C 语言类别的书架上,依然还能看见这本书被放在一个非常醒目并且触手可及的位置上。

能从书架上挑出本书的人,我想大多都是对 C 语言指针带有“恐惧感”的程序员吧!其实所谓的“恐惧感”来源于“困惑”,而“困惑”又来自于“对知识点不够透彻的理解”。作者运用幽默风趣并且不失犀利的笔法,从“究竟什么是 C 语言指针”开始,通过实验一步一步地为我们解释了指针和数组、内存、数据结构的关系,以及指针的常用手法。另外,还通过独特的方式教会我们怎样解读 C 语言那些让人“纠结”的声明语法。

带着学习的态度,我对原著的每一个章节阅读三次以上后才开始动笔翻译。每次阅读我都会有新的收获,建议购买本书的读者不要读了一遍就将其束之高阁(甚至一遍读不下来就扔到一边)。隔一段时间再来读一遍,收获会更多。

在翻译的过程中,我身边的许多人给了我莫大的支持和鼓励。我的同事的宋岩、王红升在 C 语言方面都具有 10 年以上的编程经验,他们经常牺牲个人的休息时间帮我试读译稿,提出了诸多宝贵的意见和建议。开始翻译这本书时,我儿昀好刚出生三个月。新的生命改变了一家的生活状态,带给我们更多的是感动和欢乐。妻子葛亚文在我翻译本书的期间默默承受了产后在身体上和精神上的巨大压力,这不是一句感谢能够回报的。借此祝愿一家——四季有昀,岁月静好!

吴雅明

2012/11/26 于北京

前 言

这是一本关于 C 语言的数组和指针的书。

一定有很多人感到纳闷：“都哪朝哪代了，还出版 C 语言的书。”

C 语言确实是非常陈旧的语言，不过也不可能马上放弃对它的使用。至少在书店里，C 语言方面的书籍还是汗牛充栋的，其中专门讲解指针的书也有很多。既然如此，还有必要旧瓶装新酒吗？这才是最应该质疑的吧。

但是，每当我看到那些充斥在书店里的 C 语言入门书籍，总会怀疑这些书的作者以前根本没有使用 C 开发过大规模的系统。当然，并不是所有书的作者都这样。

指针被认为是 C 语言中最大的难点，对它的讲解，很多书都搞得像教科书一样，叙述风格雷同，让人感觉有点装腔作势。就连那些指针的练习题，其中的说明也让人厌倦。

能够炮制出这样的书籍，我想一般都得归功于那些连自己对 C 语言语法都是一知半解的作者。特别是面对那些在封面上堂堂正正地印上“第 2 类信息处理考试”^①字样的书，这种感觉更加强烈。

当我还是个菜鸟的时候，也曾对数组和指针的相关语法感到非常“纠结”。

正是抱着“要是那个时候上天能让我遇见这样一本书，那可真帮了大忙”的想法，我写了这本书。

本书的内容，是基于我很久以前（1998 年 7 月）就开始在网上公开的内容：

“深入学习数组和指针”

<http://kmaebashi.com/programmer/pointer.html>

“当我傻呀？既然可以在网上阅读，我干嘛还买你的书？”我想对有此想法的人说：“我敢打票，绝不会让你吃亏的，请放心地拿着这本书去收款台结账吧！”因为此书在出版过程中追加

① 日本国内关于计算机信息处理方面的考试，主要面向计算机系统开发、维护、运用领域的初级技术人员。

了大量的文字和插图，实际上已经比网上公开的内容丰富了许多。

另外，在阅读本书的过程中，请留心以下几点。

- ❑ 本书的读者群虽然定位于“学习过 C 语言，但是在指针的运用上遇到困难”的读者，但还是能随处可见一些高难度的内容。那是因为我也不能免俗，偶尔也喜欢把自己掌握的知识拿出来显摆一下。

对于初学者，你完全没有必要从头开始阅读。遇到还不太明白的地方，也不要过分纠结。阅读中可以跳跃章节。对于第 0 章和第 1 章，最好还是按顺序阅读。如果认为第 2 章有点难度，你可以先去啃第 3 章。如果第 3 章也不懂，不妨尝试先去阅读第 4 章。这种阅读方式是本书最大的卖点。

- ❑ 在本书中，我会经常指出一些“C 的问题点”和“C 的不足”。可能会有一些读者认为我比较讨厌 C 语言。恰恰相反，我认为 C 是一门伟大的开发语言。倒不是因为“情人眼里出西施”、“能干的坏小子也可爱”这样的理由，毕竟在开发现场那些常年被使用的语言中，C 语言还是有相当实力的。就算是长得不太帅，但论才干，那也是“开发现场的老油条”了。

所以，因阅读本书而开始抱怨“C 语言真是很差劲”的读者，你即使计划了什么“去揍 Dennis Ritchie^①之旅”，我也不会去参加的。如果有“去揍 James Gosling^②之旅”，那还是有点心动的。哈，还是算了吧，得过且过就行啦。

在本书的写作过程中，我得到了很多人的帮助。

繁忙之中阅读大量原稿并指出很多错误的泽田大浦先生、山口修先生、桃井康成先生，指出本书网上公开内容的错误的人们，还有那些受到发布在公司内部的内容的影响而沦为“实验小白鼠”的人们，以及通过 `fj.com.lang.c` 和各种邮件列表进行讨论并且提供各种信息的人们，正是因为你们，本书的内容才能更加可靠。当然，遗留的错误由我来承担所有责任。

发现我的网页，并给予出版机会的技术评论社的熊谷裕美子小姐，还有给予初次写书的我很多指导的编辑高桥阳先生，如果没有他们的大力协助，这本书是不可能诞生的。

在这里，我谨向他们致以深深的谢意。

2000 年 11 月 28 日 03:33 J.S.T.

前桥和弥

① C 语言之父。本书中对他做了介绍。

② Java 语言之父。

目 录

第 0 章 本书的目标与结构——引言	1	1.3.7 声明函数形参的方法	48
0.1 本书的目标	1	第 2 章 做个实验见分晓——C 是怎么	
0.2 目标读者和内容结构	3	使用内存的	51
第 1 章 从基础开始——预备知识和		2.1 虚拟地址	51
复习	7	2.2 C 的内存的使用方法	56
1.1 C 是什么样的语言	7	2.2.1 C 的变量的种类	56
1.1.1 比喻	7	2.2.2 输出地址	58
1.1.2 C 的发展历程	8	2.3 函数和字符串常量	61
1.1.3 不完备和不统一的语法	9	2.3.1 只读内存区域	61
1.1.4 ANSI C	10	2.3.2 指向函数的指针	62
1.1.5 C 的宝典——K&R	11	2.4 静态变量	64
1.1.6 C 的理念	12	2.4.1 什么是静态变量	64
1.1.7 C 的主体	14	2.4.2 分割编译和连接	64
1.1.8 C 是只能使用标量的语言	15	2.5 自动变量(栈)	66
1.2 关于指针	16	2.5.1 内存区域的“重复使用”	66
1.2.1 恶名昭著的指针究竟是什么	16	2.5.2 函数调用究竟发生了什么	66
1.2.2 和指针的第一次亲密接触	17	2.5.3 可变长参数	73
1.2.3 指针和地址之间的微妙关系	23	2.5.4 递归调用	80
1.2.4 指针运算	26	2.6 利用 malloc()来进行动态内存分配	
1.2.5 什么是空指针	27	(堆)	84
1.2.6 实践——swap 函数	31	2.6.1 malloc()的基础	84
1.3 关于数组	34	2.6.2 malloc()是“系统调用”吗	88
1.3.1 运用数组	34	2.6.3 malloc()中发生了什么	89
1.3.2 数组和指针的微妙关系	37	2.6.4 free()之后, 对应的内存区域	
1.3.3 下标运算符[]和数组是没有		会怎样	91
关系的	39	2.6.5 碎片化	93
1.3.4 为什么存在奇怪的指针运算	42	2.6.6 malloc()以外的动态内存分配	
1.3.5 不要滥用指针运算	43	函数	94
1.3.6 试图将数组作为函数的参数		2.7 内存布局对齐	98
进行传递	45	2.8 字节排序	101

2.9 关于开发语言的标准和实现—— 对不起，前面的内容都是忽悠的.....	102	3.5.6 练习——挑战那些复杂的声 明	153
第 3 章 揭秘 C 的语法——它到底是 怎么回事	105	3.6 应该记住：数组和指针是不同的 事物	157
3.1 解读 C 的声明	105	3.6.1 为什么会引起混乱	157
3.1.1 用英语来阅读	105	3.6.2 表达式之中	158
3.1.2 解读 C 的声明	106	3.6.3 声明	160
3.1.3 类型名	109	第 4 章 数组和指针的常用方法	161
3.2 C 的数据类型的模型	111	4.1 基本的使用方法	161
3.2.1 基本类型和派生类型	111	4.1.1 以函数返回值之外的方式来返 回值	161
3.2.2 指针类型派生	112	4.1.2 将数组作为函数的参数传递	162
3.2.3 数组类型派生	113	4.1.3 可变长数组	163
3.2.4 什么是指向数组的指针	114	4.2 组合使用	166
3.2.5 C 语言中不存在多维数组！	116	4.2.1 可变长数组的数组	166
3.2.6 函数类型派生	117	4.2.2 可变长数组的可变长数组	172
3.2.7 计算类型的大小	119	4.2.3 命令行参数	174
3.2.8 基本类型	121	4.2.4 通过参数返回指针	177
3.2.9 结构体和共用体	122	4.2.5 将多维数组作为函数的参数 传递	181
3.2.10 不完全类型	123	4.2.6 数组的可变长数组	182
3.3 表达式	125	4.2.7 纠结于“可变”之前，不妨 考虑使用结构体	183
3.3.1 表达式和数据类型	125	4.3 违反标准的技巧	187
3.3.2 “左值”是什么——变量的 两张面孔	129	4.3.1 可变长结构体	187
3.3.3 将数组解读成指针	130	4.3.2 从 1 开始的数组	189
3.3.4 数组和指针相关的运算符	132	第 5 章 数据结构——真正的指针的 使用方法	193
3.3.5 多维数组	133	5.1 案例学习 1：计算单词的出现频率	193
3.4 解读 C 的声明（续）	137	5.1.1 案例的需求	193
3.4.1 const 修饰符	137	5.1.2 设计	195
3.4.2 如何使用 const？可以使用到 什么程度？	139	5.1.3 数组版	200
3.4.3 typedef	141	5.1.4 链表版	203
3.5 其他	143	5.1.5 追加检索功能	211
3.5.1 函数的形参的声明	143	5.1.6 其他的数据结构	214
3.5.2 关于空的下标运算符[]	146	5.2 案例学习 2：绘图工具的数据结构	218
3.5.3 字符串常量	148	5.2.1 案例的需求	218
3.5.4 关于指向函数的指针引起的 混乱	151	5.2.2 实现各种图形的数据模型	219
3.5.5 强制类型转换	152		

5.2.3	Shape 型	221	6.2	惯用句法	245
5.2.4	讨论——还有别的方法吗	223	6.2.1	结构体声明	245
5.2.5	图形的组合	228	6.2.2	自引用型结构体	246
5.2.6	继承和多态之道	233	6.2.3	结构体的相互引用	247
5.2.7	对指针的恐惧	236	6.2.4	结构体的嵌套	248
5.2.8	说到底，指针究竟是什么	237	6.2.5	共用体	249
第 6 章	其他——拾遗	239	6.2.6	数组的初始化	250
6.1	陷阱	239	6.2.7	char 数组的初始化	250
6.1.1	关于 strncpy()	239	6.2.8	指向 char 的指针的数组的初 始化	251
6.1.2	如果在早期的 C 中使用 float 类型的参数	240	6.2.9	结构体的初始化	252
6.1.3	printf()和 scanf()	242	6.2.10	共用体的初始化	252
6.1.4	原型声明的光和影	243	6.2.11	全局变量的声明	253

第 0 章

本书的目标与结构——引言

0.1 本书的目标

在 C 语言的学习中，指针的运用被认为是最大的难关。

关于指针的学习，我们经常听到下面这样的建议：

“如果理解了计算机的内存和地址等概念，指针什么的就简单了。”

“因为 C 是低级语言，所以先学习汇编语言比较好。”

果真如此吗？

正如那些 C 语言入门书籍中提到的那样，变量被保存在内存的“某个地方”。为了标记变量在内存中的具体场所，C 语言在内存中给这些场所分配了编号（地址）。因此，大多数运行环境中，所谓的“指针变量”就是指保存变量地址的变量。

到此为止的说明，所有人都应该觉得很简单吧。

理解“指针就是地址”，可能是指针学习的必要条件，但不是充分条件。现在，我们只不过刚刚迈出了“万里长征的第一步”。

如果观察一下菜鸟们实际使用 C 指针的过程，就会发现他们往往会有如下困惑。

□ 声明指针变量 `int *a;`……到这里还挺像样的，可是当将这个变量作为指针使用时，依然悲剧地写成了 `*a`。

- ❑ 给出 `int &a`; 这样的声明（这里不是指 C++ 编程）。
- ❑ 啥是“指向 `int` 的指针”？不是说指针就是地址吗？怎么还有“指向 `int` 的指针”，“指向 `char` 的指针”，难道它们还有什么不同吗？
- ❑ 当学习到“给指针加 1，指针会前进 2 个字节或者 4 个字节”时，你可能会有这种疑问：“不是说指针是地址吗？这种情况下，难道指针不应该是前进 1 个字节吗？”
- ❑ `scanf()` 中，在使用 `%d` 的情况下，变量之前需要加上 `&` 才能进行传递。为什么在使用 `%s` 的时候，就可以不加 `&`？
- ❑ 学习到将数组名赋给指针的时候，将指针和数组完全混为一谈，犯下“将没有分配内存区域的指针当做数组进行访问”或者“将指针赋给数组”这样的错误。

出现以上混乱的情形，并不是因为没有理解“指针就是地址”这样的概念。其实，真正导演这些悲剧的幕后黑手是：

- ❑ C 语言奇怪的语法
- ❑ 数组和指针之间微妙的兼容性

某些有一定经验的 C 程序员会觉得 C 的声明还是比较奇怪的。当然也有一些人可能并没有这种体会，但或多或少都有过下面的疑问。

- ❑ C 的声明中，`[]` 比 `*` 的优先级高。因此，`char *s[10]` 这样的声明意为“指向 `char` 的指针的数组”——搞反了吧？
- ❑ 搞不明白 `double (*p)[3]`; 和 `void (*func)(int a)`; 这样的声明到底应该怎样阅读。
- ❑ `int *a` 中，声明 `a` 为“指向 `int` 的指针”。可是表达式中的指针变量前 `*` 却代表其他意思。明明是同样的符号，意义为什么不同？
- ❑ `int *a` 和 `int a[]` 在什么情况下可以互换？
- ❑ 空的 `[]` 可以在什么地方使用，它又代表什么意思呢？

本书的编写就是为了回答以上这样的问题。

很坦白地说，我也是在使用了 C 语言好几年之后，才对 C 的声明语法大彻大悟的。

世间的人们大多不愿意承认自己比别人愚笨，所以总是习惯性地认为“实际上只有极少的人才能够精通 C 语言指针”，以此安慰一下自己那颗脆弱的心。

例如，你知道下面的事实吗？

- 在引用数组中的元素时，其实 `a[i]` 中的 `[]` 和数组毫无关系。
- C 里面不存在多维数组。

如果你在书店里拿起这本书，翻看几页后心想：“什么呀？简直是奇谈怪论！”然后照原样把书轻轻地放回书架。那么你恰恰需要阅读这本书。

有人说：“因为 C 语言是模仿汇编语言的，要想理解指针，就必须理解内存和地址等概念。”你可能会认为：

“指针”是 C 语言所特有的、底层而邪恶的功能。

其实并不是这样的。确实，“C 指针”有着底层而邪恶的一面，但是，它又是构造链表和树等“数据结构”不可缺少的概念。如果没有指针，我们是做不出像样的应用程序的。所以，凡是真正成熟的开发语言，必定会存在指针，如 Pascal、Delphi、Lisp 和 Smalltalk 等，就连 Visual Basic 也存在指针。早期的 Perl 因为没有指针而饱受批评，从版本 5 开始也引入了指针的概念。当然，Java 也是有指针的。很遗憾，世上好像对此还存有根深蒂固的误解。

在本书中，我们将体验如何将指针真正地用于构造数据结构。

“指针”是成熟的编程语言必须具有的概念。

尽管如此，为什么 C 的指针却让人感觉格外地纠结呢？理由就是，C 语言混乱的语法，以及指针和数组之间奇怪的兼容性。

本书旨在阐明 C 语言混乱的语法，不但讲解了“C 特有的指针用法”，还针对和其他语言共有的“普遍的指针用法”进行了论述。

下面，让我们来看一下本书的具体结构。

0.2 目标读者和内容结构

本书的目标读者为：

- 粗略地读过 C 语言的入门书籍，但还是对指针不太理解的人
- 平时能自如地使用 C 语言，但实际对指针理解还不够深入的人

本书由以下内容构成。

- 第 1 章：从基础开始——预备知识和复习
- 第 2 章：做个实验见分晓——C 是怎样使用内存的

- 第3章：揭秘 C 的语法——它到底是怎么回事
- 第4章：数组和指针的常用用法
- 第5章：数据结构——真正的指针的使用方法
- 第6章：其他——拾遗

第1章和第2章主要面向初学者。从第3章开始的内容，是为那些已经具备一定经验的程序员或者已经读完第1章的初学者准备的。

面向初学者，第1章和第2章从“指针就是地址”这个观点开始讲解。

通过 `printf()` 来“亲眼目睹”地址的实际值，应该说，这不失为理解指针的一个非常简单有效的方式。

对于那些“尝试学习了 C 语言，但是对指针还不太理解”的人来说，通过自己的机器实际地输出指针的值，可以相对简单地领会指针的概念。

首先，在第1章里，针对 C 语言的发展过程（也就是说，C 是怎样“沦为”让人如此畏惧的编程语言的）、指针以及数组进行说明。

对于指针和数组的相互关系，市面上多数的 C 语言入门书籍只是含混其辞地做了敷衍解释（包括 *K&R*^{*}，我认为该书是诸恶之源）。这还不算，他们还将本来已经用数组写好的程序，特地用指针运算的方式重新编写，还说什么“这才像 C 语言的风格”。

* *K&R* 被称为 C 语言的宝典（中文版叫《C 程序设计语言（第2版·新版）》），在后面我们会提及这本书的背景。此书的作者之一就是 C 语言之父 Dennis Ritchie 本人。

像 C 语言的风格？也许是可以这么说，但是以此为由炮制出来的难懂的写法，到底好在哪里？哦？执行效率高？为什么？这是真的吗？

产生这些疑问是正常的，并且，这么想是正确的。

了解 C 语言的发展过程，就能理解 C 为什么会有“指针运算”等这样奇怪的功能。

第1章中接下来的内容也许会让初学者纠结，因为我们将开始接触到数组和指针的那些容易让人混淆的语法。

第2章讲解了 C 语言实际上是怎样使用内存的。

在这里同样采用直观的方式将地址输出。请有 C 运行环境的读者一定亲手输入例程的代码，并且尝试运行。

对于普通的局部变量、函数的参数、`static` 变量、全局变量及字符串常

量（使用" "包围的字符串）等，知晓它们在内存中实际的保存方式，就可以洞察到 C 语言的各种行为。

遗憾的是，几乎所有使用 C 语言开发的程序，运行时检查都不是非常严密。一旦出现诸如数组越界操作，就会马上引起“内存区域破坏”之类的错误。虽然很难将这些 bug 完全消灭，但明白了 C 如何使用内存之后，至少可以在某种程度上预防这些 bug 的出现。

第 3 章讲解了与数组和指针相关的 C 语言语法。

虽然我多次提到“究竟指针为什么这么难”，但是对于“指针就是地址”这个观点，在理解上倒是非常简单。出现这种现象，其实缘于 C 语言的数组和指针的语法比较混乱。

乍看上去，C 语言的语法比较严谨，实际上也存在很多例外。

对于那些平时和我们朝夕相处的语法，究竟应套用哪条规则？还有，哪些语法需要特殊对待？关于这些，第 3 章里会做彻底的讲解。

那些自认为是老鸟的读者，可以单独拿出第 3 章来读一读，看看自己以前是如何上当的。

第 4 章是实践篇，举例说明数组和指针的常用用法。如果读者理解了这部分内容，对付大部分程序应该不在话下。

老实说，对于已经将 C 语言使用得像模像样的读者来说，第 4 章中举出的例子并没有什么新意。但是，其实有些人对这些语法只是一知半解，很多时候只不过是依照以前的代码“照猫画虎”罢了。

阅读完第 3 章后去读第 4 章，对于那些已经能够熟练使用的写法，你也会惊呼一声：“原来是这个意思啊！”

第 5 章中，解说指针真正的用法——数据结构的基本知识。

前四章中的例子，都是围绕 C 语言展开的。第 5 章里则会涉及其他语言的指针。

无论使用哪种语言编程，“数据结构”都是最重要的。使用 C 语言来构造数据结构的时候，结构体和指针功不可没。

“不仅仅对于 C 语言的指针，连结构体也不太明白”的读者，务必不要错过第 5 章。

第6章中，对到此为止还没有覆盖到的知识进行拾遗，并且为大家展示一些可能会遇到的陷阱以及惯用语法。

和类似的书籍相比，本书更加注重语法的细节。

提起语法，就有“日本的英语教育不偏重语法”，以至于给人“就算不明白也没有什么”的印象。确实是这样，我们早在不懂“サ行”怎样变形之前，就已经会说日语了。

但是，C语言可不是像日语那样复杂的自然语言，它是一门编程语言。

单纯地通过语法来解释自然语言是非常困难的。比如，日语“いれたてのおちゃ”，利用假名汉字变换程序只能变换成“淹れたてのお茶”（中文意思是‘沏好的茶’），尽管如此，同样通过假名变换程序，“いれたてのあついおちゃ”竟然也可以变换成“入れた手の厚いお茶”（中文意思是‘沏好的热腾腾的茶’）^①。编程语言最终还是通过人类制订的语法构成的，它要做到让编译器这样的程序能够解释。

“反正大家都是这么写，我也这么写，程序就能跑起来。”

这种想法，让人感觉有点悲哀。

我希望不仅是初学者，那些已经积累了一定经验的程序员也能阅读本书。通过深入理解C的语法，可以让我们真正领会直到今天还像“口头禅”一样使用的那些程序惯用写法。

无论如何，让我们做到“知其然知其所以然”，这样有利心理健康，不是吗？

^① 中文里也有类似的例子，如：他是先知。→他是先知道那件事的人。——译者注

第 1 章

从基础开始——预备知识和复习

1.1 C 是什么样的语言

1.1.1 比喻

在 Donald C. Gause 和 Gerald M. Weinberg 合著的《你的灯亮着吗？》^[1]一书中，有这样一节（根据需要，我做了必要的删减）。

某计算机制造商开发了一种新型打印机。

技术小组在如何保证打印精度的问题上非常苦恼，每次进行新的测试时，工程师都不得不花很长的时间测量打印机的输出结果来追求精确性。

丹 (Dan Daring) 是这个小组中最年轻但或许是最聪明的工程师。他发明了一种工具，即每隔 8 英寸就在铝条上嵌上小针。使用这个工具，可以很快地找到打印机输出位置的误差。

这个发明显著地提高了生产效率，丹的上司非常高兴，提议给丹颁发一个公司的特别奖赏。他从车间里拿了工具，带回办公室，这样他写报告的时候还可以仔细地研究一下。

这个上司显然还用不惯这个工具，当他把这个工具放在桌子上的时候，将针尖朝上了。更不幸的是，当丹的上司的上司友好地坐到桌角上，打算谈谈给丹颁发奖励时，部门内的所有人都听到了他痛苦的尖叫声——他的屁股上被扎了两个相距 8 英寸的孔。

C 语言就恰如这个工具。也就是说，它是一门

* 这本书的副标题为“发现问题的真正所在”，它通过一些趣闻轶事来告诉世人“不要急于寻找问题的答案，而是应该先去考虑当前的问题是什么”。

- 为了解决眼前问题，由开发现场的人发明的，
- 虽然使用方便，
- 但看上去不怎么顺眼，
- 如果不熟悉的人糊里糊涂地使用了它，难免会带来“悲剧”的语言。

1.1.2 C的发展历程

众所周知，C原本是为了开发UNIX操作系统而设计的语言。

如此说来，好像C应该比UNIX更早问世，可惜事实并非如此，最早的UNIX是用汇编来写的。

因为厌倦了总是苦哈哈地使用汇编语言进行编程，UNIX的开发者Ken Thompson开发了一种称为“B”的语言。B语言是1967年剑桥大学的Martin Richard开发的BCPL（Basic CPL）的精简版本。BCPL的前身是1963年剑桥大学和伦敦大学共同研究开发的CPL（Combined Programming Language）语言。

B语言不直接生成机器码，而是由编译器生成栈式机（Stack Machine）用的中间代码，中间代码通过解释器（interpreter）执行（类似Java和早期的Pascal）。因此，B语言的执行效率非常低，结果，在后来的UNIX开发过程中人们放弃了使用B语言。

在这之后的1971年，Ken Thompson的同事Dennis Ritchie对B语言做了改良，追加了char数据类型，并且让B语言可以直接生成PDP-11*的机器代码。曾经在很短的时间内，大家将这门语言称为NB（New B）。

之后，NB改称为C语言——这就是C语言的诞生。

后来，主要是为了满足使用UNIX的程序员的需要，C语言一边接受来自各方面的建议，一边摸着石头过河般地进行着周而复始的功能扩展。

1978年出版了被称为C语言宝典的*The C Programming Language*一书。

此书取了两位作者（Brian Kernighan和Dennis Ritchie）的姓氏首字母，简称为K&R。在后面提到的ANSI标准制定之前，此书一直作为C语言语法的参考书被人们广泛使用。

听说这本书在最初发行的时候，Prentice-Hall出版社制订了对于当时存在的130个UNIX站点平均每个能卖9本的销售计划（相比*Lift With UNIX*^[2]）。

* 现在已经不存在的DEC（美国数字设备公司）生产的微型电脑。

当然了，哪怕是初版 *K&R* 的销售量，也以 3 位数的数量级超过了 Prentice-Hall 出版社最初的销售计划*。原本只是像“丹的工具”一样为了满足自用的 C 语言，历经坎坷，最终成为全世界广泛使用的开发语言。

* 我手头这本 *K&R* 是在 1997 年 5 月 1 日出版的第二版（翻译修订版），已经是第 211 次印刷了。这个行业的图书能有这样的业绩，确实惊人。



补充

B 是什么样的语言？

在 C 语言的入门书籍中，经常提到 C 是 B 语言的进化版本。但几乎所有的书对 B 语言的介绍都只有这么多，没有具体说明 B 语言究竟是一门什么样的语言。

正如前面描述的那样，B 是在虚拟机上运行的、解释型开发语言。B 语言没有像 Java 那样想要去实现“到处运行”的宏伟目标，它只是因为当时运行 UNIX 的 PDP-7 硬件环境的限制，而只能采用解释器这样的实现方式。

B 是“没有类型”的语言。虽然 C 里面有 `char`、`short`、`int`、`float` 和 `double` 等很多的数据类型，但是 B 使用的类型只有 `word`（你可以认为和 `int` 差不多）。作为本书的主题——指针，在 B 里面和整数一样使用。因为无论怎么说，指针无非就是内存中的地址，对于机器来说，它是可以和整型同样对待的（关于这点，在本章中会详细说明）。

NB 是具有类型概念的语言。为了把指针和整数纠葛不清的 B 移植到 NB，Dennis Ritchie 在指针运用的设计上下了很大的工夫。C 的指针让人感觉很难理解，可能也有这方面的原因。

关于 B 语言，如果你想知道更详细的内容，可以浏览 Dennis Ritchie 的网页：<http://cm.bell-labs.com/cm/cs/who/dmr/index.html> 中的“Users' Reference to B”^[3]等内容。

1.1.3 不完备和不统一的语法

C 语言是开发现场的人们根据自身的需要开发出来的语言，所以具备极高的实用性。但反过来从人类工程学的角度来看，它就不是那么完美了。

比如：

```
if (a == 5) { ← 本来应该写成==的地方却写成了=
```

相信大家都犯过这样的错误吧。

在日语键盘上，“-”和“=”在同一按键上，因此经常会发生下面的问题：

```
for (i = 0; i < 100; i++) ← 忘了和[shift]键一起按下
```

就连这样的错误，编译器也往往无法察觉。现在的编译器倒是可以给出警告，可是早期的编译器对这样的错误是完全忽略的。

使用 `switch case` 的时候，也经常发生忘了写 `break` 的错误。

幸运的是，如今的编译器，对于容易犯的语法错误，在很多地方可以给我们警告提示。因此，不但不能无视这些编译器的警告，相反应该提高编译器的警告级别，让编译器替我们指出尽可能多的错误*。

* 尽管如此……假设无视了有返回值的函数返回的值，`lint` 会给出一个警告。为了消除这个讨厌的警告，你特地使用 `(void)printf(...)` 打印返回的值，这么干是不是就有点过了？C 原本就是“本性恶劣”的语言，警告级别过高，会否定一些既存的程序写法，反而带来不好的后果。此外，如果最大限度地提高警告级别，有些应用程序自身包含的头文件也会引起警告。总之，这些问题还是比较麻烦的。

要 点

提高编译器的警告级别。

不可无视或者制止编译器的警告。

C 语言是在使用中成长起来的语言。因此，由于很多历史原因遗留了一些“奇怪的”问题。具有代表性的有位运算符“&”和“|”的优先顺序问题。

通常，如“==”的比较运算符的优先级要低于那些做计算的运算符。因此，

```
if (a < b + 3)
```

这样的条件表达式中，虽然可以不使用括号来写，但是当使用了位运算符的时候，就行不通了。

想要进行“将 `a` 和 `MASK` 进行按位与运算后的结果，再和 `b` 做比较运算”，

```
if (a & MASK == b)
```

按照上面的写法，因为 `&` 运算符的优先级低于 `==` 运算符，所以被解释成了下面这样：

```
if (a & (MASK == b))
```

这是因为在没有“`&&`”和“`||`”运算符的时代，使用“`&`”和“`|`”来代替而留下的后遗症。

1.1.4 ANSI C

即使在 *K&R* 出版之后，C 仍在不断地扩展。

比如关于结构体的一次性赋值,在 *K&R* 的初版里面并没有记述,其实这个功能在 *K&R* 出版之前,就已经在 Dennis Ritchie 的 C 编译器里实现了。从某种意义上来说, *K&R* 的第一版刚出版就已经过时了。这在计算机图书界是常有的事。

另外, *K&R* 的记述也不一定就是严密的,由于运行环境的不同,程序运行也存在差异。

鉴于这些原因,经过一番纷争,终于在 1989 年,ANSI (American National Standard Institute, 美国国家标准学会) 通过了 C 语言的标准规范,这就是通常被称为 ANSI C 的 C 语言。目前使用的 C 程序,大部分都是基于 ANSI C 编写的。

顾名思义,ANSI 是美国的标准。难道不存在 C 语言的国际标准吗? 那是不可能的。ANSI C 后来被 ISO 采用,目前 C 的真正标准应该是 ISO 的 C。另外,原始的 ANSI 标准说明书的章节编号和 ISO 不同。

ISO-C 标准的名称为“ISO-IEC 9899-1990”,当然这是用英语命名的。JIS (日本工业标准) 标准 (JIS X3010) 原样采用了 ISO-C 标准,所以英语不太好的人 (比如我),也可以获取日语版的标准文档。^①

从日本工业标准协会可以得到 JIS X3010 标准,以及 JIS X3010 的手册。另外,《信息处理: 编程语言篇》里面也包含了这个标准*,此书可以从书店订购。如果是那些大型的书店,也许在书架上就能发现这本书。

在本书后面的内容中,提到的“标准”都是指 JIS X3010。

* 2001 年, JIS 手册全面修订,删除了《信息处理: 编程语言篇》中已经存在的内容。如果在所有的标准中都记录相同的信息会引起不必要的麻烦。

1.1.5 C 的宝典——*K&R*

之前已经介绍过, Brian Kernighan 和 Dennis Ritchie (C 语言之父) 合著的 *The C Programming Language* 被称为 *K&R*。在制订 ANSI C 之前, *K&R* 是 C 语言语法的使用标准。

人们把 ANSI 之前的 C 称为“K&R C”,这可能会引起一些误解。但无论怎样,制订了 ANSI C 标准之后,追随 ANSI C 的 *K&R* 就紧跟着出版了第 2 版^[4]。

在本书中,提起 ANSI C 之前的 C,我们还是尊重事实,称之为“ANSI C 之前的 C”。

① 国内读者可参考中国国家标准 GB/T 15272-94,它即为 ISO-IEC 9899-1990 的中文翻译版。
——译者注

* 最初的 K&R 第 2 版，因为翻译质量的问题，恶评如潮。重新翻译后再次出版了修订版（原书相同）。第 2 版旧翻译版的封面为绿色，新翻译版的封面为白色。^①

此外，本书中提及 K&R 时，是指日语版的《编程语言 C》的第 2 版修订版*。

顺便提一下，通过

<http://www.cs.bell-labs.com/cm/cs/cbook/index.html>

可以看到 K&R 的网页，各语种的 K&R 封面排列在一起，颇为壮观。



补充

新的 C

C 语言的功能扩展并没有随着 ANSI C 的发布而停下脚步。ISO 通过 ISO C9X 这个代号名称，计划制定具备更多扩展功能的 C 语言规范。

从 ISO C9X 这个代号名称可以看出来，大家都很期待在 20 世纪 90 年代完成新标准的制订，标准文档封面上的日期是 1999 年 12 月 1 日——好险呀!!

作为 ISO C99 的扩展功能，除了提供对复数类型的支持之外，还包括“可以用变量定义本地数组变量的元素个数”、“将数组作为结构体的成员进行声明时元素个数可以不定义（只需写[]）”等功能。这些功能看上去都和本书的主题有重合的部分。

尽管如此，这个“新的 C”在今后究竟能使用多久，我们现在不得而知。所以，本书不参照 ISO C99。

1.1.6 C 的理念

ANSI C 标准，附有 Rationale（理论依据）。

可以通过下面的地址在线获取 Rationale。

<ftp://ftp.uu.net/doc/standards/ansi/x3.159-1989/>

Rationale 中有“keep the spirit of C”（保持 C 的精神）一节，关于“C 的精神”是这样介绍的：

- ① 请信任程序员（Trust the programmer）
- ② 不要阻止程序员去做需要做的工作（Don't prevent the programmer

^① 中文版请参考《C 程序设计语言（第 2 版·新版）》。——译者注

第 3 章

揭秘 C 的语法——它到底是怎么回事

3.1 解读 C 的声明

3.1.1 用英语来阅读

在 1.2.2 节的补充内容中，我认为像

```
int *hoge_p;
```

还有

```
int hoge[10];
```

这样的声明方式很奇怪。

对于这种程序的声明方式，可能也有很多人感觉不到有什么别扭的地方。那就再看下面的这个例子（经常被使用）：

```
char *color_name[] = {  
    "red",  
    "green",  
    "blue",  
};
```

这里声明了一个“指向 char 的指针的数组”。

正如 2.3.2 节中介绍的那样，可以像下面这样声明一个“指向将 double 作为参数并且返回 int 的函数的指针”：

```
int (*func_p)(double);
```

关于这样的声明，在 *K&R* 中有下面这样一段说明：

```
int *f();      /* f: 返回指向 int 指针的函数*/
```

和

```
int (*pf)();  /* pf: 指向返回 int 的函数的指针*/
```

这两个声明最能说明问题。在这里，因为*是前置运算符，它的优先级低于()，为了让连接正确地进行，有必要加上括号。

首先，这段文字中有谎言。

声明中*、()和[]并不是运算符。在语法规则中，运算符的优先顺序是在别的地方定义的。

先将这个问题放在一边。如果你老老实实地去读这段文字，该会嘀咕“是不是搞反了”。如果说

```
int (*pf)();
```

是指向函数的指针，使用括弧先将星号（指针）括起来岂不是很奇怪？

这个问题的答案，等你明白过来就会觉得非常简单。C语言本来是美国人开发的，最好还是用英语来读*。

以上的声明，如果从 pf 开始以英语的顺序来读，应该是下面这样：

```
pf is pointer to function returning int
```

翻译成中文，则为

```
pf 为指向返回 int 的函数的指针。
```

要 点

用英语来读 C 的声明。

* 在 *K&R* 中，登载了 dc1 这个解析 C 的声明的程序，同时也记载了程序的输出结果，但是日语版并没有对这一段进行翻译，而是一成不变地转载了英文原文。

3.1.2 解读C的声明

在这里，向读者介绍阅读 C 语言声明的方法：机械地向前读。

为了把问题变得更简单，我们在这里不考虑 `const` 和 `volatile`。（3.4 节考虑了 `const`）接下来遵循以下步骤来解释 C 的声明。

- ① 首先着眼于标识符（变量名或者函数名）。
- ② 从距离标识符最近的地方开始，依照优先顺序解释派生类型（指针、数组和函数）。优先顺序说明如下，

- ①用于整理声明内容的括弧
- ②用于表示数组的[], 用于表示函数的()
- ③用于表示指针的*

- ❸ 解释完成派生类型, 使用“of”、“to”、“returning”将它们连接起来。
- ❹ 最后, 追加数据类型修饰符(在左边, int、double 等)。
- ❺ 英语不好的人, 可以倒序用日语(或者中文)^①解释。

数组元素个数和函数的参数属于类型的一部分。应该将它们作为附属于类型的属性进行解释。

比如,

```
int (*func_p)(double);
```

- ❶ 首先着眼于标识符。

```
int (*func_p)(double);
```

英语的表达为:

func_p is

- ❷ 因为存在括号, 这里着眼于*。

```
int (*func_p)(double);
```

英语的表达为:

func_p is pointer to

- ❸ 解释用于函数的(), 参数是 double。

```
int (*func_p)(double);
```

英语的表达为:

func_p is pointer to function(double) returning

- ❹ 最后, 解释数据类型修饰符 int。

```
int (*func_p)(double);
```

英语的表达为:

func_p is pointer to function(double) returning int

① 这里同样可以倒序用中文解释。——译者注

⑤ 翻译成中文：

func_p 是指向返回 int 的函数的指针。

使用和上面相同的方式，我们对各种各样的声明进行解读，如下表（表 3-1）。

表3-1 解读各种各样的C语言声明

C语言	英语表达	中文表达
int hoge;	hoge is int	hoge是int
int hoge[10];	hoge is array(元素个数10) of int	hoge是int的数组(元素个数10)
int hoge[10][3];	hoge is array(元素个数10) of array(元素个数3) of int	hoge是int数组(元素个数3)的数组(元素个数10)
int *hoge[10];	hoge is array(元素个数10) of pointer to int	hoge是指向int的指针的数组(元素个数10)
int (*hoge)[3];	hoge is pointer to array(元素个数3) of double	hoge是指向int的数组(元素个数3)的指针
int func(int a);	func is function(参数为int a) returning int	func是返回int的函数(参数是int a)
int (*func)(int a);	func is pointer to function(参数为int a) returning int	func_p是指向返回int的函数(参数为int a)的指针

正如大家看到的这样，C 语言的声明不能从左往右按顺序解读（无论是英语、中文，还是日语），而是左右来回地解读。

K&R 中指出：在 C 语言中，变量的声明仿效表达式的语法。可是，勉强地去模拟本质上完全不同的事物，结果就是“四不像”。

* 其实，Java 的大部分声明语法还是能做到这点的。

“使声明的形式和使用的形式相似”是 C（还有从 C 派生的 C++、Java* 等语言）特有的奇怪的语法。

K&R 中同时也写道：

C 的声明语法，特别是指向函数指针的语法，受到了严厉的批评。

在 Pascal 中，C 的 int hoge[10]可以这样声明，

```
var
  hoge : array[0..9] of integer;
```

这种声明，从左向右用英语按顺序解读是完全没有问题的。

顺便说一下,C的作者 Dennis Ritchie 开发了一种叫 Limbo^[7]的语言。Limbo 中各种标记的使用方法,一眼就可以看出来和 C 非常相似*,但是声明语法完全设计成 Pascal 风格。其实作者自身也在反省 C 语言的声明语法。

* 比如,不使用 `begin~end` 或者 `if~endif` 而是使用中括号。

3.1.3 类型名

在 C 中,除标识符以外,有时候还必须定义“类型”。

具体来说,遇到以下情况需定义“类型”:

- 在强制转型运算符中
- 类型作为 `sizeof` 运算符的操作数

比如,将强制转型运算符写成下面这样:

```
(int*)
```

这里指定“`int*`”为类型名。


从标识符的声明中,将标识符取出后,剩下的部分自然就是类型名。

表3-2 类型名的写法

声 明	声明的解释	类 型 名	类型名的解释
<code>int hoge;</code>	<code>hoge</code> 是 <code>int</code>	<code>int</code>	<code>int</code> 类型
<code>int *hoge;</code>	<code>hoge</code> 是指向 <code>int</code> 的指针	<code>int *</code>	指向 <code>int</code> 的指针类型
<code>double (*p)[3];</code>	<code>p</code> 是指向 <code>double</code> 的数组(元素个数3)的指针	<code>double(*)[3]</code>	指向 <code>double</code> 的数组(元素个数3)的指针类型
<code>void (*func)();</code>	<code>func</code> 是指向返回 <code>void</code> 函数的指针	<code>void (*)()</code>	指向返回 <code>void</code> 函数的指针类型

在表 3-2 最后两个例子中,括起星号的括弧(*)好像有点多余,但是一旦去掉括弧,意思就完全不一样了。

(`double *[3]`)是将 `double *hoge[3]`的标识符去掉后形成的,所以这个类型名被解释成“指向 `double` 的指针的数组”。



补充

如果将指针后置……

C 的声明语法虽然奇怪,但有人说 Pascal 风格写起来长得像裹脚布,同样让人感到厌恶。

```
var
    hoge : array[0..9] of integer;
```

关于这样的声明，`array` 后面紧接着`[]`，用来表示这是个数组，但是这样让人感觉 `array` 这个单词太长了。顺手在后面追加的 `of` 也是多余的。尝试将这些多余的部分去掉，结果就像下面这样：

```
var
    hoge : [0..9] integer;
```

如果改成C的写法：

```
hoge[10] int;
```

如果仅仅就 `int` 前置还是后置的问题来说，感觉和C的声明方式也没多大差别。

可是，一旦涉及指针，情况就不一样了。C的指针运算符`*`是前置的。

在Pascal中，运算符`^`相当于C中`*`的，而且它是后置的。

如果同样地将C的指针运算符`*`也放在标识符后面，即使兼顾“变量的声明仿效表达式的语法”，声明也会变成下面这样：

```
int func_p^(double);
```

如果这个声明表示“指向返回 `int` 的函数（参数为 `double`）的指针”，差不多也符合英语的阅读顺序。不过 `int` 放在前面终究是个问题。

此外，一旦使用后置的`^*`，通过指针引用结构体的成员的时候，就可以不要`->`运算符了。

原本

```
hoge->piyo
```

只是

```
(*hoge).piyo
```

的语法糖，所以又可以写成

```
hoge^.piyo
```

因此`->`完全可以不要的。

此外，将解引用后置，可以使包含结构体成员和数组引用的复杂表达式变得简洁*。

关于这一点，“The Development of the C Language^[5]”中也有说明：

Sethi [Sethi 81] observed that many of the nested declarations and

* C 中，`^`被作为异或运算符使用。这里，我们且不必去关心这一点。

* 另外，如果将指针的强制转型也进行后置，同样也能起到简化表达式的作用。

expressions would become simpler if the indirection operator had been taken as a postfix operator instead of prefix, but by then it was too late to change.

请允许我用我这二把刀的英语水平给大家翻译一下：

Sethi 认为，如果将解引用由前置变成后置，嵌套的声明和表达式就会变得更简单。但是，如今想要修正，已经为时已晚。

3.2 C 的数据类型的模型

3.2.1 基本类型和派生类型

假设有下面这样的声明：

```
int (*func_table[10])(int a);
```

根据上节中介绍的方法，可以解释成：

指向返回 `int` 的函数（参数为 `int a`）的指针的数组（元素个数 10）

如果画成图，可以用这样的链结构^①来表示：

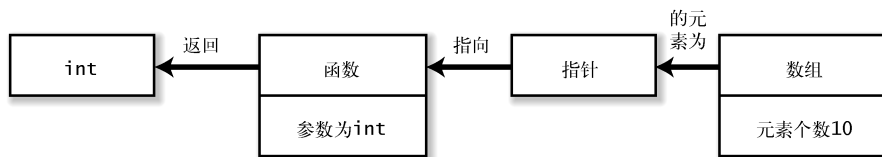


图 3-1 用图表现“类型”

这种表示，在本书中称为“类型链的表示”。

姑且先忽视结构体、共用体、**typedef** 等类型，概要地进行说明，链的最后面的元素*是基本类型，这里可能是 `int` 或者 `double`。

此外，从倒数第 2 个元素开始的元素都是派生类型。所谓“派生类型”，就是指从某些类型派生出来的类型。

* 如果按照日语的语序，应该是最前面的元素。^②

① 由于日语和中文语序的差异，图中日文原书的箭头和本书相反。——译者注

② 译本采用中文的语序。——译者注

除了结构体、共用体之外，还有以下3种派生类型：

- 指针
- 数组（“元素个数”作为它的属性）
- 函数（参数信息作为它的属性）

关于派生类型，*K&R*中有这样一段描述（p.239）：

除基本的算术类型以外，利用以下的方法可以生成概念上无限种类的派生类型。

- 给出类型的对象的数组
- 返回给出类型的对象的函数
- 指向给出类型的对象的指针
- 包含各种一系列对象的结构体
- 能够包含各种类型的数个对象中的任意一个共用体

一般来说，这些对象的生成方法可以递归使用。

可能大家完全不明白这段描述在说什么。其实归纳一下，可以表述成下面这句话*，

* 实际上，派生还有其他几个限制，关于这些我们在后面介绍。

从基本类型开始，递归地（重复地）粘附上派生类型，就可以生成无限类型。

通过如图 3-1 的方式将链不断地接长，就可以不断生成新的“类型”。

另外，在链中，最后的类型（数组(元素 10)）整合了全体类型的意义，所以我们将最后的类型称为类型分类。

比如，无论是“指向 `int` 的指针”，还是“指向 `double` 的指针”，结果都是“指针”；无论是“`int` 的数组”，还是“指向 `char` 的指针的数组”，结果都是“数组”。

3.2.2 指针类型派生

在 1.2.1 节中，引用了标准中的一节，在此，请允许我再次引用。

指针类型（`pointer type`）可由函数类型、对象类型或不完整的类型派生，派生指针类型的类型称为引用类型。指针类型描述一个对象，该类对象的值提供对该引用类型实体的引用。由引用类型 `T` 派

生的指针类型有时称为“(指向)T的指针”。从引用类型构造指针类型的过程称为“指针类型的派生”。这些构造派生类型的方法可以递归地应用。

“由引用类型 T 派生的指针类型有时称为‘(指向)T 的指针’”这句话,可以在图 3-2 中用链来表现。

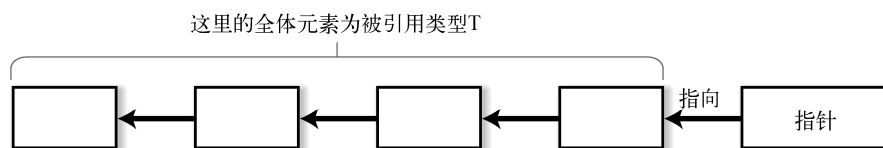


图 3-2 指针类型派生

对于指针类型来说,因为它指向的类型各不相同,所以都是从既存的类型派生出“指向 T 的指针”这样的类型。

大多数处理环境中的指针,在实现上都只是单纯的地址,所以无论从什么类型派生出的指针,在运行时的状态都是大体相同的*。但是,加上*运算符求值的时候,以及对指针进行加法运算的时候,由不同类型派生出来的指针之间就存在差异。

* 前面也曾经提到,如果解释得详细一些,对于指向 `char` 的指针和指向 `int` 的指针,偶尔存在位数不相同的处理环境。

前面已经提到,如果对指针进行加法运算,指针只前进指针所指向类型的大小的距离。这一点对于后面的说明有着非常重要的意义。

可以使用图 3-3 来解释指针类型。

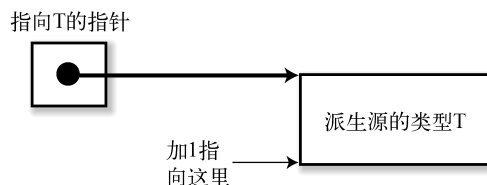


图 3-3 指针类型的图解

3.2.3 数组类型派生

和指针类型相同,数组类型也是从其元素的类型派生出来的。“元素个数”作为类型的属性添加在类型后面(参照图 3-4)。

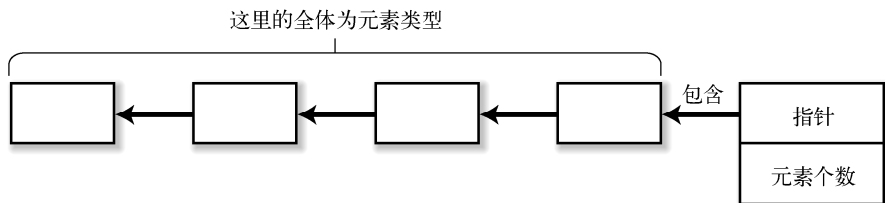


图 3-4 数组类型派生

数组类型本质就是将一定个数的派生源的类型进行排列而得到的类型。如图 3-5 所示。

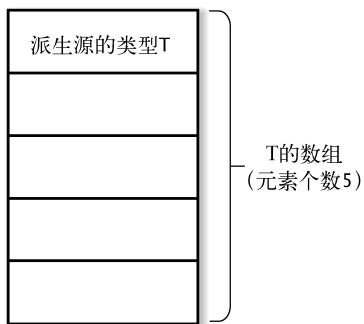


图 3-5 数组行的图解

3.2.4 什么是指向数组的指针

“数组”和“指针”都是派生类型。它们都是由基本类型开始重复派生生成的。

也就是说，派生出“数组”之后，再派生出“指针”，就可以生成“指向数组的指针”。

一听到“指向数组的指针”，有人也许要说：

这不是很简单嘛，数组名后不加[]，不就是“指向数组的指针”吗？

抱有这个想法的人，请将 1.3 节的内容重新阅读一下！的确，在表达式中，数组可以被解读成指针。但是，这不是“指向数组的指针”，而是“指向数组初始元素的指针”。

实际地声明一个“指向数组的指针”，

```
int (*array_p)[3];
array_p 是指向 int 数组（元素个数 3）的指针。
```

根据 ANSI C 的定义，在数组前加上&，可以取得“指向数组的指针”^{*}。因此，

```
int array[3];
int (*array_p)[3];
```

```
array_p = &array;    ← 数组添加&，取得“指向数组的指针”
```

这样的赋值是没有问题的，因为类型相同。

可是，如果进行

```
array_p = array;
```

这样的赋值，编译器就会报出警告。

“指向 int 的指针”和“指向 int 的数组（元素个数 3）的指针”是完全不同的数据类型。

但是，从地址的角度来看，array 和&array 也许就是指向同一地址。但要说起它们的不同之处，那就是它们在做指针运算时结果不同。

在我的机器上，因为 int 类型的长度是 4 个字节，所以给“指向 int 的指针”加 1，指针前进 4 个字节。但对于“指向 int 的数组（元素个数 3）的指针”，这个指针指向的类型为“int 的数组（元素个数 3）”，当前数组的尺寸为 12 个字节（如果 int 的长度为 4 个字节），因此给这个指针加 1，指针就前进 12 个字节（参照图 3-6）。

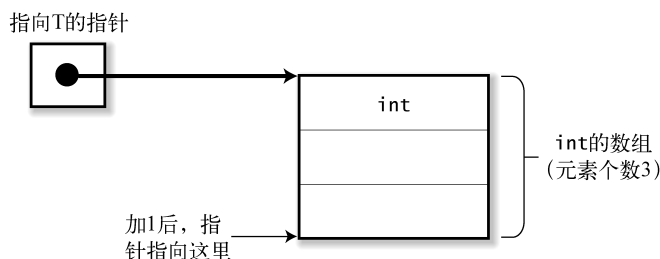


图 3-6 对“指向数组的指针”进行加法运算

道理我明白了，但是一般没有人这么用吧？

可能有人存在以上的想法。但真的有很多人就是这么用的，只不过是自己没有意识到。为什么这么说呢？在后面的章节中将会说明。

* 这里是“数组可以解读成指向它初始元素的指针”这个规则的一个例外（参照 3.3.3 节）。

3.2.5 C语言中不存在多维数组！

在C中，可以通过下面的方式声明一个多维数组：

```
int hoge[3][2];
```

我想企图这么干的人应该很多。请大家回忆一下C的声明的解读方法，上面的声明应该怎样解读呢？

是“int类型的多维数组”吗？

这是不对的。应该是“int的数组（元素个数2）的数组（元素个数3）”。

也就是说，即使C中存在“数组的数组”，也不存在多维数组*。

“数组”就是将一定个数的类型进行排列而得到的类型。“数组的数组”也只不过是派生源的类型恰好为数组。图3-7是“int的数组（元素个数2）的数组（元素个数3）”。

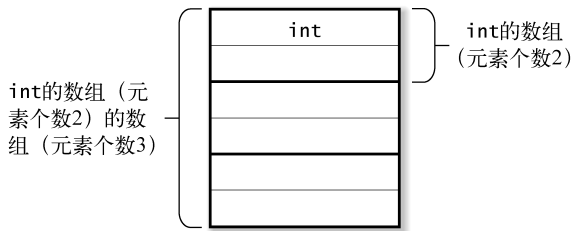


图 3-7 数组的数组

要 点

C语言中不存在多维数组。

看上去像多维数组，其实是“数组的数组”。

对于下面的这个声明：

```
int hoge[3][2];
```

可以通过 `hoge[i][j]` 的方式去访问，此时，`hoge[i]` 是指“int的数组（元素个数2）的数组（元素个数3）”中的第 *i* 个元素，其类型为“int数组（元素个数2）”。当然，因为是在表达式中，所以在此时此刻，`hoge[i]` 也可以被解读成“指向int的指针”。

关于这一点，3.3.5节中会有更详细的说明。

* 在C标准中，“多维数组”这个词最初出现在脚注中，之后这个词也会不时地出现在各个角落。尽管“不存在多维数组”这个观点会让人感觉有些极端，但如果你不接受这个观点，对于C的类型模型的理解，可能就会比较困难。

那么，如果将这个“伪多维数组”作为函数的参数进行传递，会发生什么呢？

试图将“int 的数组”作为参数传递给函数，其实可以直接传递“指向 int 的指针”。这是因为在表达式中，数组可以解释成指针。

因此，在将“int 的数组”作为参数传递的时候，对应的函数的原型如下：

```
void func(int *hoge);
```

在“int 的数组（元素个数 2）的数组（元素个数 3）”的情况下，假设使用同样的方式来考虑，

int 的数组（元素个数 2）的数组（元素个数 3）

其中下划线部分，在表达式中可以解释成指针，所以可以向函数传递

指向 int 的数组（元素个数 2）的指针

这样的参数，说白了它就是“指向数组的指针”。

也就是说，接收这个参数的函数的原型为：

```
void func(int (*hoge)[2]);
```

直到现在，有很多人将这个函数原型写成下面这样：

```
void func(int hoge[3][2]);
```

或者这样：

```
void func(int hoge[][2]);
```

其实，

```
void func(int (*hoge)[2]);
```

就是以上两种写法的语法糖，它和上面两种写法完全相同。

关于将数组作为参数进行传递这种情况下的语法糖，在 3.5.1 节中会再一次进行说明。

3.2.6 函数类型派生

函数类型也是一种派生类型，“参数（类型）”是它的属性。

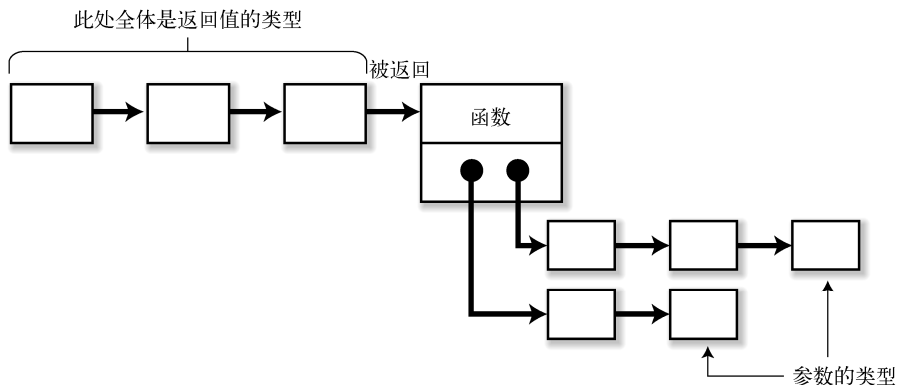


图 3-8 函数类型派生

可是，函数类型和其他派生类型有不太相同的一面。

无论是 `int` 还是 `double`，亦或数组、指针、结构体，只要是函数以外的类型，大体都可以作为变量被定义。而且，这些变量在内存占用一定的空间。因此，通过 `sizeof` 运算符可以取得它们的大小。

像这样，有特定长度的类型，在标准中称为**对象类型**。

可是，函数类型不是对象类型。因为函数没有特定长度。所以 C 中不存在“函数类型的变量”（其实也没有必要存在）。

数组类型就是将几个派生类型排列而成的类型。因此，数组类型的全体长度为：

派生源的类型的大小 × 数组的元素个数

可是，函数类型是无法得到特定长度的，所以从函数类型派生出数组类型是不可能的。也就是说，不可能出现“函数的数组”这样的类型。

可以有“指向函数的指针”类型，但不幸的是，对指向函数类型的指针不能做指针运算，因为我们无法得到当前指针类型的大小。

此外，函数类型也不能成为结构体和共用体的成员。

总而言之：

从函数类型是不能派生出除了指针类型之外的其他任何类型的。

不过“指向函数的指针类型”，可以组合成指针或者作为结构体、共用体的

成员。毕竟“指向函数的指针类型”也是指针类型，而指针类型又是对象类型。

另外，函数类型也不可以从数组类型派生。

可以通过“返回 ~ 的函数”的方式派生出函数类型，不过在 C 中，数组是不能作为函数返回值返回的（参照 1.1.8 节）。

要 点

从函数类型是不能派生出除了指针类型之外的其他任何类型的。

从数组类型是不能派生出函数类型的。

3.2.7 计算类型的大小

除了函数类型和不完全类型（参照 3.2.10 节），其他类型都有大小。

通过

```
sizeof(类型名)
```

编译器可以为我们计算当前类型的大小，无论是多么复杂的类型。

```
printf("size..%d\n", sizeof(int(*)(5))(double));
```

以上的语句表示输出

指向返回 int 的函数（参数为 double）的指针的数组(元素个数 5)的大小。

在这里顺便对以前的内容也做一下复习：模仿编译器的处理方式，尝试计算各种类型的大小。

另外，我们考虑使用如下构成的机器来作为处理环境。

```
int      4 个字节
double   8 个字节
指针     4 个字节
```

【注意！】

在这里我们为了说明方便，特别地对处理环境做了假定。但是，C 语言的标准并没有对 int、double 和指针的大小进行任何规定，数据类型的大小完全取决于各处理环境的具体实现。

因此，我们通常不需要去留意数据类型的物理大小，更不应该依赖数据类型大小进行编程。

可以像下面这样，以日语^①词组的顺序计算类型的大小：

❶ 基本类型

基本类型必定依赖处理环境进行计算。

❷ 指针

指针的大小是依赖处理环境决定的。大部分情况下，指针的大小和派生源的大小没有关系，它的大小是固定的。

❸ 数组

数组的大小可以通过派生源类型的大小乘以元素个数得到。

❹ 函数

函数的大小无法计算。

现在，可以尝试计算刚才的示例。

指向返回 `int` 的函数（参数为 `double`）的指针的数组（元素个数 5）的大小。

❶ 指向返回 `int` 的函数（参数为 `double`）的指针的数组（元素个数 5）。

因为是 `int` 类型，所以在当前假定的处理环境中，计算结果为 4 个字节。

❷ 指向返回 `int` 的函数（参数为 `double`）的指针的数组（元素个数 5）。

因为是函数，所以无法计算大小。

❸ 指向返回 `int` 的函数（参数为 `double`）的指针的数组（元素个数 5）。

因为是指针，所以在当前假定的处理环境中，计算结果为 4 个字节。

❹ 指向返回 `int` 的函数（参数为 `double`）的指针的数组（元素个数 5）。

因为是派生源的大小为 4 的“元素个数为 5 的数组”，所以计算结果为 $4 \times 5 = 20$ 个字节。

同样地，表 3-3 中整理了对各种类型的大小进行计算的结果。

① 这里同样可以用中文词组的顺序来计算。——译者注

表3-3 计算各种类型的大小

声 明	中文的表现	大 小
<code>int hoge;</code>	hoge是int	4个字节
<code>int hoge[10]</code>	hoge是int的数组	$4 \times 10=40$ 个字节
<code>int *hoge[10];</code>	hoge是int的指针的数组（元素个数10）	$4 \times 10=40$ 个字节
<code>double *hoge[10]</code>	hoge是double的指针的数组（元素个数10）	$4 \times 10=40$ 个字节
<code>int hoge[2][3];</code>	hoge是int的数组（元素个数3）的数组（元素个数2）	$4 \times 3 \times 2=24$ 个字节

3.2.8 基本类型

派生类型的底层是基本类型。

基本类型指，`char` 和 `int` 这样的整型以及 `float` 和 `double` 这样的浮点型。这些类型加上枚举类型，统称为算术型*。

此外，在 C 中，通过 `short int` 声明一个变量，和单纯地通过 `short` 声明一个变量，意义是完全一样的。

对于整型和浮点型，怎样的写法是允许的，哪种写法和哪种写法的意义是相同的，这些内容非常琐碎，特整理如下（表 3-4）。

* 在标准中，枚举类型没有包括在基本类型（basic type）中（6.1.2.5）。在 *K&R* 中，它们被混在一起了。

表3-4 整数型、浮点型的种类

推 荐	同义的表现
<code>char</code>	
<code>signed char</code>	
<code>unsigned char</code>	
<code>short</code>	<code>signed short</code> , <code>short int</code> , <code>signed short int</code>
<code>unsigned short</code>	<code>unsigned short int</code>
<code>int</code>	<code>signed</code> , <code>signed int</code> , 无指定类型
<code>unsigned int</code>	<code>unsigned</code>
<code>long</code>	<code>signed long</code> , <code>long int</code> , <code>signed long int</code>
<code>unsigned long</code>	<code>unsigned long int</code>
<code>float</code>	
<code>double</code>	
<code>long double</code>	

`char` 和 `signed char` 或者 `unsigned char` 同义。至于默认情况下，`char` 究竟是有符号的还是无符号的，C 标准并没有定义，而是取决于处理环境。

根据处理环境不同，`long long` 等写法有可能也是允许的，这些都不在标准的约束范围之内。ANSI C 以前的 C，存在 `long float` 这样的写法，它和 `double` 同义，但是在 ANSI C 之后这种写法就被废弃了。

另外，对于这些类型的大小，`sizeof(char)`（包含 `signed`、`unsigned`）必定返回 1。其他的类型全部依赖处理环境的定义。即使是 `char`，在 `sizeof` 肯定会返回 1 的情况下，也没有规定肯定是 8 位，现实中也存在 `char` 为 9 位的处理环境。

偶尔有一些不靠谱的 C 语言入门书籍会跟大家乱嚼舌头：“`int` 的大小是依赖于硬件的，所以尽量不要使用 `int`。”这种观点完全是错误的。不只是 `int`，无论是 `short` 还是 `long`，它们的大小都依赖于处理环境。具有讽刺意味的是，几乎所有持有这种观点的入门书的例程也在使用 `int`。言行不一？说话不算数？哎，我都不知道该怎么说了……

不过标准还是规定了每种类型可以表示的值的范围：

- 有符号 `char`， ± 127
- 无符号 `char`， $0 \sim 255$
- 有符号 `int`，有符号 `short`， ± 32767
- 无符号 `int`，`unsigned short`， $0 \sim 65535$
- 有符号 `long`， ± 2147483647
- 无符号 `long`， $0 \sim 4294967295$

请注意在这里有符号 `int` 的最小值不是 -32768 ，这是因为考虑到有的机器对于负数使用 1 的补码^①。

3.2.9 结构体和共用体

在语法上，结构体和共用体是作为派生类型使用的。

可是直到现在，我们还没有专门去说明结构体和共用体。其理由如下：

- 虽然结构体和共用体在语法上属于派生类型，但是在声明中它和数据类型修饰符（也就是 `int`、`double` 等）处于相同的位置。
- 只有派生指针、数组和函数的时候，类型才可以通过一维链表表示。结构体、共用体派生类型只能用树结构进行表现。

① 在二进制原码的情况下，有符号 `int` 的最小值为 -32767 。但是补码系统中有符号 `int` 的最小值为 -32768 ，因为负数需要把符号位以后的部分取反加 1。——译者注

结构体类型可以集合几个其他不同的类型，而数组只能线性地包含同一个类型。

共用体的语法和结构体相似，但是，结构体的成员是“排列地”分配在内存中，而共用体的成员则是“重叠地”分配在内存中。在第 5 章将会介绍共用体的用途。

让我们通过图 3-9，尝试使用“类型链的方式”来表现结构体和共用体的派生。

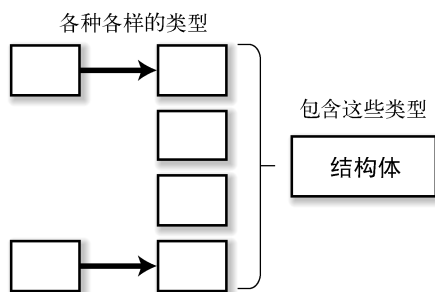


图 3-9 结构体类型的派生

3.2.10 不完全类型

不完全类型指“函数之外、类型的大小不能被确定的类型”。

总结一下，C 的类型分为：

- ❑ 对象类型（char、int、数组、指针、结构体等）
- ❑ 函数类型
- ❑ 不完全类型

结构体标记的声明就是一个不完全类型的典型例子。

对于男性（Man），他可能有妻子（wife）。如果是未婚男性，wife 就是 NULL。所以，Man 这样的类型，可以声明成下面这样：

```
struct Man_tag {
    :
    struct Woman_tag *wife; /*妻*/
    :
};
```

作为妻子，可以这样声明：

```
struct Woman_tag {
    :
    struct Man_tag *husband; /*夫*/
    :
};
```

这种情况下，`struct Man_tag` 和 `struct Woman_tag` 是相互引用的，所以无论先声明哪一边都很麻烦。

可以像下面这样通过先声明结构体标记来回避以上问题：

```
struct Woman_tag;    ← 将 tag 提前声明

struct Man_tag {
    :
    struct Woman_tag *wife; /* 妻 */
    :
};
struct Woman_tag {
    :
    struct Man_tag *husband; /* 夫 */
    :
};
```

在我的环境中，结构体必须使用 `typedef`，所以，

```
typedef struct Woman_tag Woman;    ← 提前对 tag 进行类型定义

typedef struct {
    :
    Woman *wife; /* 妻 */
    :
} Man;

struct Woman_tag {
    :
    Man *husband; /* 夫 */
    :
};
```


对这种情况，在 `Woman` 类型的标记被声明的时候，还不知道其内容，所以无法确定它的大小。这样的类型就称为**不完全类型**。

因为不能确定大小，所以不能将不完全类型变成数组，也不能将其作为结构体的成员，或者声明为变量。但如果仅仅是用于取得指针，是可以使用不完全类型的。上面的结构体 `Man`，就是将 `Woman` 类型的指针作为它的成员。

之后，在定义 `struct Woman_tag` 的内容的时候，`Woman` 就不是不完全类型了。

在 C 标准中，`void` 类型也被归类为不完全类型。

3.3 表达式

3.3.1 表达式和数据类型

直到现在，我们没有进行明确地定义就使用了**表达式**（`expression`）这个词。

首先介绍**基本表达式**（`primary expression`），基本表达式是指：

- ❑ 标识符（变量名、函数名）
- ❑ 常量（包括整数常量和浮点数常量）
- ❑ 字符串常量（使用“”括起来的字符串）
- ❑ 使用 `()` 括起来的表示式

此外，对表达式使用运算符，或通过运算符将表达式和表达式相互连接，这些表示方法也称为表达式。

也就是说，“5”、“hoge”都是表达式（如果已经声明了以 `hoge` 作为名称的变量）。此外，“`5 + hoge`”也是表达式。

对于下面的表达式，

```
a + b * 3 / (4 + c)
```

它可以表现成如图 3-10 这样的树结构，这个树结构的所有部分的树*都是表达式。

* 这里指某个特定节点以下的树。

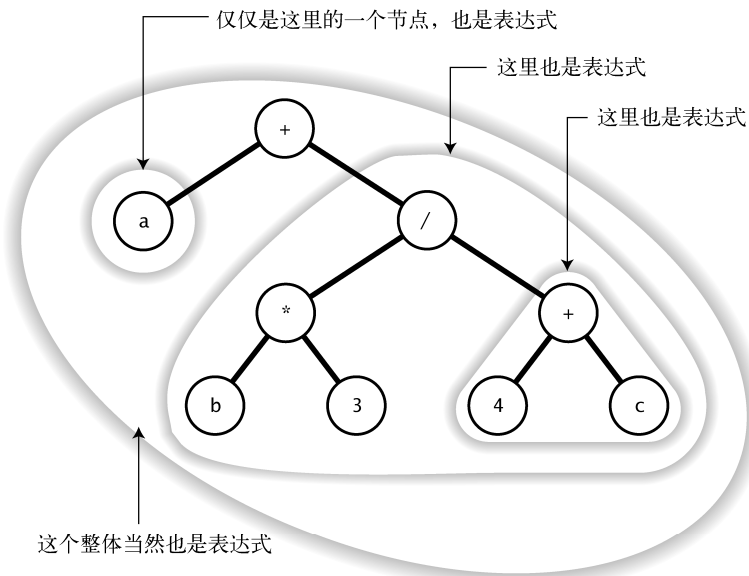


图 3-10 表达式的树结构

此外，所有的表达式都持有类型。

3.2 节中介绍了可以通过链的结构表现类型。如果所有的表达式都持有类型，那么对于表现表达式的树结构的节点，都可以被挂接上表现类型的链（参照图 3-11）。

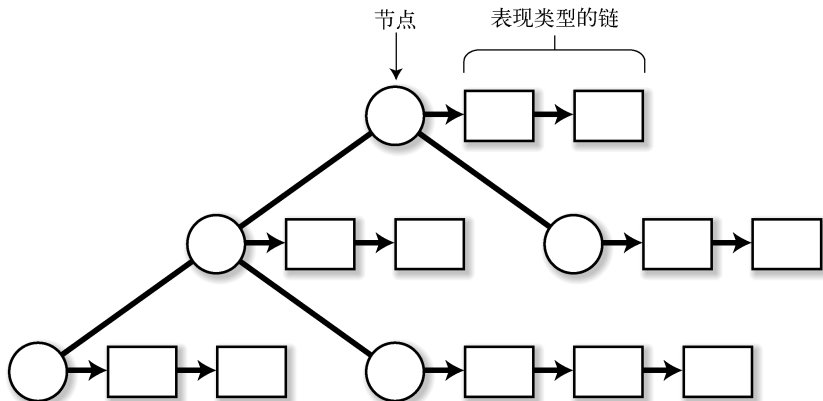


图 3-11 所有的表达式都持有类型

在对表达式使用运算符，或者将表达式作为参数传递给函数的时候，表达式中持有的类型具有特别重要的意义。

比如，对于下面这样的数组：

```
char str[256];
```

在输出这个字符数组的内容的时候，使用

```
printf(str);
```

初学者看到这段程序，难免会想：“printf()能这么写吗？”

确实，就像这个世上最有名的程序中写的这样：

```
printf("hello, world\n");
```

第 1 个参数总是传递字符串常量。

可是，在 stdio.h 的原型声明中，printf() 的第 1 参数被定义为“指向 char 指针”。

字符串常量的类型为“char 的数组”，因为是在表达式中，所以它也可以当成“指向 char 的指针”。因此，字符串常量可以传递给 printf()。同样地，str 是“char 的数组”，因为是在表达式中，所以也可以当成“指向 char 的指针”，能够传递给 printf() 也是很自然的事。

如果只是单纯地输出字符串，对于字符串中包含 % 感到麻烦，与其使用

```
printf("%s", str);
```

不如使用 puts() 会更好。这个就扯远了。

此外，下面的写法也许会让某些人感到惊奇：

```
"01234567890ABCDEF"[index]
```

但如果写成这样：

```
str[index]
```

谁都不会觉得奇怪了吧。在表达式中，str 和字符串常量都属于“指向 char 的指针”，它们都可以作为 [] 运算符的操作数*。



补充

针对“表达式”使用 sizeof

sizeof 运算符有两种使用方法。

一种是：

* 运算符 (operator) 的作用对象被称为操作数。比如， $1 + 2$ 这个表达式，1 和 2 是运算符 + 的操作数。

```
sizeof(类型名)
```

另外一种是：

```
sizeof 表达式
```

后者能够返回对象表达式的类型的大小。

在实际开发中，“sizeof 表达式”这种使用方式的唯一用途，就是从编译器获取数组的长度。

对于下面的声明，

```
int hoge[10];
```

在 `sizeof(int)` 为 4 的处理环境中，

```
sizeof(hoge)
```

返回 40。因此将这个结果除以 `sizeof(int)` 就可以得到数组元素的个数。

如果像这个例子这样显式地指定了数组的大小，即使不使用 `sizeof`，使用 `#define` 给大小定义一个合适的名称也是可以满足需求的。不过在下面的情况下，使用 `sizeof` 也许更加方便。

```
char *color_name[] = {  
    "black",  
    "blue",  
    :  
    :  
};
```

```
#define COLOR_NUM (sizeof(color_name) / sizeof(char*))
```

在这种情况下，由于使用了数组初始化表达式，这里可以省略定义数组元素的个数*，因此就不需要使用 `#define` 定义一个固定的常量了。另外，在某些情况下需要在 `color_name` 中追加更多的元素，如果使用 `sizeof`，只需修改程序的一个地方。

无论怎样，`sizeof` 运算符只是向编译器询问大小的信息，所以，它只能在编译器明确知道对象大小的情况下使用。

```
extern int hoge[];
```

以上的情况是不能使用 `sizeof` 的。此外，

```
void func(int hoge[])  
{  
    printf("%d\n", sizeof(hoge));  
}
```

* 参照 3.5.2 节。

这样的程序，也只是输出指针的长度（参照 3.5.1 节）。

也许很多人并不知道，对于“`sizeof` 表达式”，其实不需要括号。当然，加上括号也可以，此时的括号只是单纯地起到括起操作数的作用。

有人尽管知道这一点，但为了阅读的方便，依然使用了括号。

3.3.2 “左值”是什么——变量的两张面孔

假设有下面这样一个声明：

```
int hoge;
```

因为此时 `hoge` 是 `int` 类型，所以，要是可以写 `int` 类型的值的地方，`hoge` 就可以像常量一样使用。

比如，将 5 赋予 `hoge` 之后，下面的语句

```
piyo = hoge * 10;
```

理所当然地可以写成

```
piyo = 5 * 10;
```

但是，在

```
hoge = 10;
```

的情况下，即使此时 `hoge` 的值为 5，

```
5 = 10;
```

这样的置换也是非法的。

也就是说，作为变量，它有作为“自身的值”使用和作为“自身的内存区域”使用两种情况。

此外在 C 中，即使不是变量名，表达式也可以代表“某个变量的内存区域”。比如这种情况：

```
hoge_p = &hoge;
```

```
*hoge_p = 10;    ← *hoge_p 是指 hoge 的内存区域
```

像这样，表达式代表某处的内存区域的时候，我们称当前的表示式为左值（lvalue）；相对的是，表达式只是代表值的时候，我们称当前的表达式为右值。

表达式中有时存在左值，有时不存在左值。比如，根据上下文，表达式可以作为左值或者右值使用，但是5这样的常量，或者`1 + hoge`这样的表达式却只能解释成右值。



补充

“左值”这个词汇的由来

在C以前的语言中，因为表达式在赋值的左边，所以表达式被解释成左值。“左”在英语中是left，left value就被简写成lvalue。

但在C中，`++hoge`这样写法也是合法的，此时hoge是指某处的内存区域，但是怎么看也看不出“左边”的意思。因此，左值这个词真有点让人摸不着头脑。

在标准委员会的定义中，lvalue的l不是left的意思，而表示locator(指示位置的事物)。Rationale中有下面一段描述，

The Committee has adopted the definition of lvalue as an object locator.

尽管如此，JIS X3010 还是将 lvalue 解释成了“左值”。^①

3.3.3 将数组解读成指针

正如在前面翻来覆去提到的那样，在表达式中，数组可以解读成指针。

```
int hoge[10];
```

以上的声明中，hoge 等同于 `&hoge[0]`。

hoge 原本的类型为“int 的数组（元素个数 10）”，但并不妨碍将其类型分类“数组”变换为“指针”。

图 3-12 表现了其变换的过程。

^① 中国国家标准 GB/T 15272-94（189 页）中，也是将 lvalue 解释成左值。——译者注

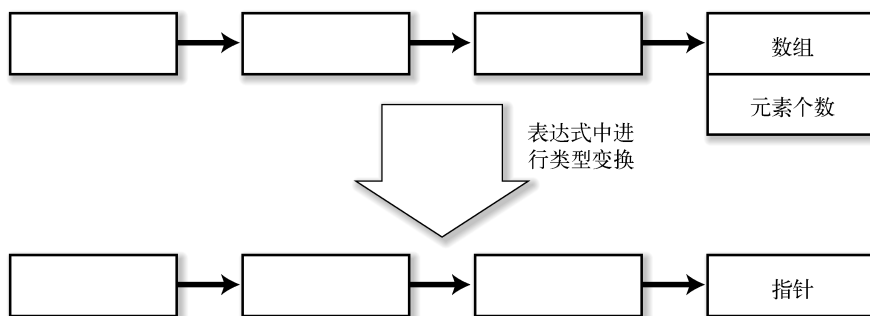


图 3-12 将数组解读成指针

此外，数组被解读成指针的时候，该指针不能作为左值。

这个规则有以下的例外情况。

❶ 数组为 sizeof 运算符的操作数

在通过“sizeof 表达式”的方式使用 sizeof 运算符的情况下，如果操作数是“表达式”，此时即使对数组使用 sizeof，数组也会被当成指针，得到的结果也只是指针自身的长度。照理来分析，应该是这样的吧？可是，当数组成为 sizeof 的操作数时，“数组解读为指针”这个规则会被抑制，此时返回的是数组整体的大小。请参照 3.3.1 节的补充内容。

❷ 数组为&运算符的操作数

通过对数组使用&，可以返回指向整体数组的指针。在 3.2.4 节中已经介绍了“指向数组的指针”。

这个规则已经被追加到 ANSIC 规则之中。此前的编译器，在对数组使用&的时候，大多会报错。因此，当时的程序在这一点上不会出现问题。那么这个规则的制定有什么好处呢？我想应该是为了保持统一吧。

❸ 初始化数组时的字符串常量

我们都知道字符串常量是“char 的数组”，在表达式中它通常被解读成“指向 char 的指针”。其实，初始化 char 的数组时的字符串常量，作为在花括号中将字符用逗号分开的初始化表达式的省略形式，会被编译器特别解释（参照 3.5.3 节）。

在初始化 char 的指针的时候，字符串常量的特别之处，需要引起注意。

3.3.4 数组和指针相关的运算符

以下介绍数组和指针相关的运算符。

▲解引用

单目运算符*被称为解引用。

运算符*将指针作为操作数，返回指针所指向的对象或者函数。只要不是返回函数，运算符*的结果都是左值。

从运算符*的操作数的类型中仅仅去掉一个指针后的类型，就是运算符*返回的表达式类型（参照图 3-13）。

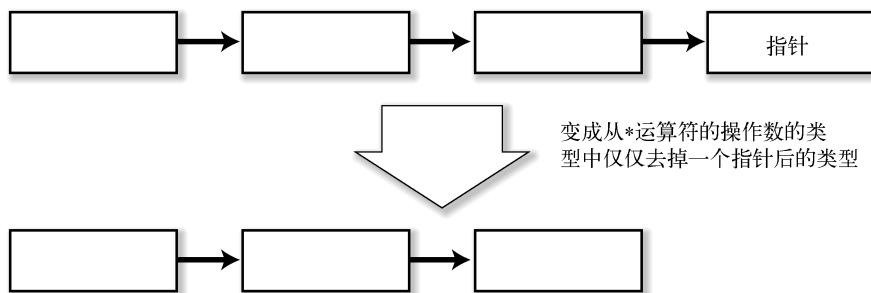


图 3-13 使用解引用而发生的类型的变化

▲地址运算符

单目运算符&被称为地址运算符。

&将一个左值作为操作数，返回指向该左值的指针。对左值的类型加上一个指针，就是&运算符的返回类型（参照图 3-14）。

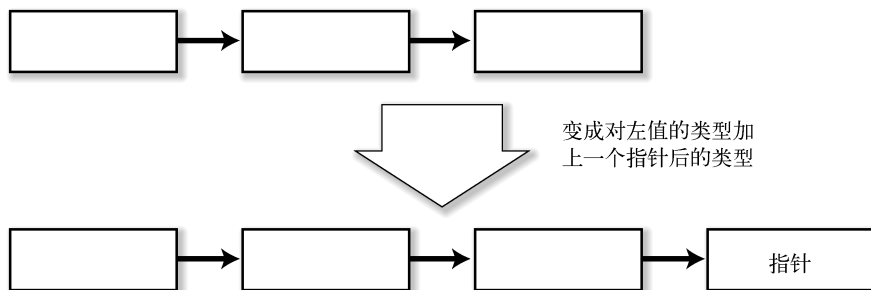


图 3-14 使用地址运算符而发生的类型的变化

地址运算符不能将非左值的表达式作为操作数。

▲下标运算符

后置运算符[]被称为下标运算符。

[]将指针和整数作为操作数。

`p[i]`

是

`*(p + i)`

的语法糖，除此以外没有任何其他意义。

对于声明为 `int a[10]` 的数组，使用 `a[i]` 的方式进行访问的时候，由于 `a` 在表达式中，因此它可以被解读成指针。所以，你可以通过下标运算符访问数组（将指针和整数作为操作数）。

归根结底，`p[i]` 这个表达式就是 `*(p + i)`，所以下标运算符返回的类型是，从 `p` 的类型去掉一个指针的类型。

▲->运算符

在标准中，似乎并没有定义->运算符的名称，现实中有时它被称为“箭头运算符”。

通过指针访问结构体的成员的时候，会使用->运算符。

`p->hoge;`

是

`(*p).hoge;`

的语法糖。

利用 `*p` 的 `*`，从指针 `p` 获得结构体的实体，然后引用成员 `hoge`。

3.3.5 多维数组

在 3.2.5 节中，我们提到了 C 语言中不存在多维数组。

那些看起来像多维数组的其实是“数组的数组”。

第 5 章

数据结构——真正的指针的使用方法

5.1 案例学习 1：计算单词的出现频率

5.1.1 案例的需求

直到第 4 章，本书主要对“声明的解读方法”和“数组和指针的微妙关系”进行了说明。

这些内容都属于 C 语言特有的话题，正因为如此，人们才普遍认为“C 的指针比较难”。

但是，指针同时也是构造链表或者树这样的“数据结构”必须的概念。因此，在比较成熟的、正统的编程语言中，必定存在指针*。本章给大家介绍在构造一般数据结构的过程中指针的使用方法。

下面是一个经常被使用的案例，非常对不起，这的确是一个没有新意、老掉牙的案例。

设计开发一个将文本文件作为输入，计算出其中各单词出现频率的程序。

这个程序的名称是 `word_count`*。

使用下面的命令行，

`word_count 文件名`

将英文文本文件的文件名作为参数传递给 `word_count`，程序执行的结果是：

* 对于链表这种程度的数据结构，如果有了集合类库，操作数据时就可以不用顾及指针的存在了。

* 这里容易和 UNIX 的 `wc` 命令产生混淆，可以先不管它。

将英文单词按照字母顺序排序后输出，并且在各单词的后面显示当前单词的出现次数。

如果省略参数，就将标准输入的内容作为输入进行处理。



补充

各种语言中对指针的叫法

正如我反复强调的那样，如果没有指针，就无法构造正统的数据结构，因此，比较成熟的、正统的编程语言，必定会存在指针*。

噢？我怎么听说 Java 就没有指针呢……

我可以负责任地告诉你，这是个谣言。

第4章的补充内容中也曾经提到，Java 只能通过指针来操作数组和对象，因此，Java 比 C 更离不开指针。

在早期的 Java 白皮书中，就有“Java 中没有指针”这样的说法*。Java 中被称为“引用”的概念，在 C 和 Pascal 的程序员看来，怎么看都相当于指针。我认为在“Java 中没有指针”这个观点的背后，弥漫着下面这样“狡猾的”市场营销的气味，

因为对于 C 语言，大家都认为“指针比较难”，如果强调“没有指针”，编程新手也许更容易接受。

但是 Java 的引用又和 C 的指针有着很大的不同。Java 没有指针运算，因此不存在指针运算和数组之间的那种微妙关系，此外你也不能取得指向变量的指针。如果你认为这些差别能成为“Java 中没有指针”的理由，那么 Pascal 是不是也没有指针呢？

除 Java 之外，Lisp、Smalltalk 和 Perl (Ver.5 以后) 中相当于指针的对象也被称为“引用”，但是也有人会使用“指针”这样的叫法。也就是说，这些语言并没有严格地将“引用”和“指针”分开。因为它们的本质相同，所以 Java 故意强调“没有指针”，反而让人觉得奇怪*。

Ruby 中连字符串这样的基本类型也不是不可变的，像这样的语言“没有指针”，是不是很危险？

Pascal、Modula2/3 和 C 一样，都称之为指针。

Ada 中的名称为“Access 类型”。这种叫法有点人让人摸不着头脑。

悲哀的是，C++ 在语法上将“指针”和“引用”区别成两个不同的概念。

C++ 的“指针”和 C、Pascal 的“指针”，以及 Java 的“引用”同义。其次，C++ 中的“引用”是指本来应该被称为“别名”(alias)的对象，

* 以前，FORTRAN、COBOL 和 BASIC 中都没有指针，但是在 fortran90、Visual Basic 等升级版中，正式引入了指针功能。

* 请参照 <http://java.sun.com/docs/overviews/java/java-overview-1.html>。

* 诞生在日本的面向对象的脚本语言 Ruby，作者在自己的著作中就断言“Ruby 中没有指针这样的概念”，其实 Ruby 中也有叫做“引用”的指针。

正因为是别名，所以一旦确定“别名是什么”，就再也不能修改了。

实际上，C++的术语“引用”也是通过指针实现的，所以它其实是一个重复的功能。很多熟练的 C++ 程序员往往不使用“引用”，而总是使用指针。但是，在某些运算符重载，以及复制构造函数的场景下，可能会不得不使用“引用”。对于 C++，有人说它太深奥，有人说使用它开发项目成本太高，甚至有人质疑“是否存在理解 C++ 全貌的人”……总之，C++ 也是一门让人纠结的开发语言。

5.1.2 设计

在开发大型应用程序的时候，非常有必要将程序按照单元功能（模块）分开。尽管这次的程序很小，但为了达到练习的目的，我们准备将这次的程序模块化。

将 `word_count` 程序分割成如图 5-1 的样子。

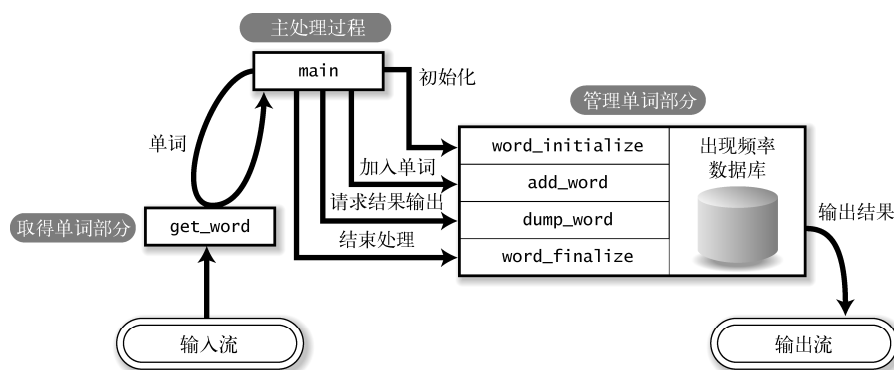


图 5-1 `word_count` 的模块结构

❶ 取得单词部分

从输入流（文件等）一个个地取得单词。

❷ 管理单词部分

管理单词。最终的结果输出功能也包含在这部分。

❸ 主处理过程

统一管理以上两个模块。

* 好像本书的章节安排都是有预谋的哦，哈哈！

* 此时，可以考虑使用4.2.4节中介绍的方法。

对于“取得单词部分”，我们完全可以直接使用1.3.6节中的实现*。

`get_word()`中，对调用方输入的单词字符数做了限制，并且存在一些无论如何都不允许的情况*。对于这一点，其实不必过于纠结，解决方案是提供一个足够大的缓冲——1024个字符的临时缓冲区。

对于如何定义“英语单词”，如果严密地去考虑，那就没完没了了，不妨结合现成的 `get_word()` 中的实现——将使用C的宏 `(ctype.h) isalnum()` 返回真的连续的字符视作单词。

作为公开给其他部分的接口，“取得单词部分”提供了 `get_word.h`（参照代码清单5-1），在需要的时候，可以通过 `#include` 引用这个头文件。

代码清单 5-1 `get_word.h`

```
1: #ifndef GET_WORD_H_INCLUDED
2: #define GET_WORD_H_INCLUDED
3: #include <stdio.h>
4:
5: int get_word(char *buf, int size, FILE *stream);
6:
7: #endif /* GET_WORD_H_INCLUDED */
```

这次的主题是“管理单词部分”。

“管理单词部分”提供了4个函数作为对外接口。

❶ 初始化

```
void word_initialize(void);
```

初始化“管理单词部分”。使用“管理单词部分”的一方，必须在一开始就要调用 `word_initialize()`。

❷ 单词的追加

```
void add_word(char *word);
```

向“管理单词部分”加入单词。

`add_word()`为传入的字符串动态地分配内存区域，并且将字符串保存在其中。

❸ 输出单词的出现频率

```
void dump_word(FILE *fp);
```

将利用 `add_word()` 加入的单词按照字母顺序进行排序, 并且向各单词追加它们各自出现的次数 (调用 `add_word()` 的次数), 最后输出 `fp` 指向的流。

④ 结束处理

```
void word_finalize(void);
```

结束“单词管理部分”。一旦结束使用“单词管理部分”, 需要调用 `word_finalize()`。

调用 `word_finalize()` 之后, 再调用 `word_initialize()`, 管理单词部分就回到最初的状态 (清空了过去加入的单词)。

将这些整理到头文件中, 代码清单 5-2 所示。

代码清单 5-2 word_manage.h

```
1:  #ifndef WORD_MANAGE_H_INCLUDED
2:  #define WORD_MANAGE_H_INCLUDED
3:  #include <stdio.h>
4:
5:  void word_initialize(void);
6:  void add_word(char *word);
7:  void dump_word(FILE *fp);
8:  void word_finalize(void);
9:
10: #endif /* WORD_MANAGE_H_INCLUDED */
```

在“主处理过程”中, 使用 `get_word()` 努力地从输入流读取单词, 然后通过调用 `add_word()`, 将每次读取到的单词, 不断地加入到“单词管理部分”, 最后再调用 `dump_word()` 将最终结果输出 (参照代码清单 5-3)。

代码清单 5-3 main.c

```
1:  #include <stdio.h>
2:  #include <stdlib.h>
3:  #include "get_word.h"
4:  #include "word_manage.h"
5:
6:  #define WORD_LEN_MAX (1024)
7:
8:  int main(int argc, char **argv)
9:  {
10:     char        buf[WORD_LEN_MAX];
11:     FILE        *fp;
```

```

12:
13:     if (argc == 1) {
14:         fp = stdin;
15:     } else {
16:         fp = fopen(argv[1], "r");
17:         if (fp == NULL) {
18:             fprintf(stderr, "%s:%s can not open.\n",
19:                 argv[0], argv[1]);
20:             exit(1);
21:         }
22:     }
23:     /* 初始化“管理单词部分” */
24:     word_initialize();
25:
26:     /* 边读取文件，边加入单词 */
27:     while (get_word(buf, WORD_LEN_MAX, fp) != EOF) {
28:         add_word(buf);
29:     }
30:     /* 输出单词的出现频率 */
31:     dump_word(stdout);
32:
33:     /* “管理单词部分”的结束处理 */
34:     word_finalize();
35:
36:     return 0;
37: }

```

代码清单 5-3 中，特意将 WORD_LEN_MAX 的值设定得大一些，但是无论如何，这只是提供给临时缓冲的空间的大小。在 add_word() 内部，只是分配需要的内存区域，并且将字符串复制进去，所以不会浪费太多的内存区域。

此外，考虑到例程的简洁，本章的程序省略了对所有的 malloc() 返回值检查。



补充

头文件的写法

写头文件的时候，必须遵守下面两个原则，

- ❶ 所有的头文件中，必须要有防止#include 重复的保护。
- ❷ 所有的头文件只#include 自己直接依赖的头文件。

“防止重复#include 的保护”是指，假设对于 word_manage.h (参照代码清单 5-2) 来说，

```
#ifndef WORD_MANAGE_H_INCLUDED
#define WORD_MANAGE_H_INCLUDED

:

#endif /* WORD_MANAGE_H_INCLUDED */
```

如果像上面这么做,即使当前头文件被重复#include,其中的内容也会被无视,因此,就不会出现重复定义的错误。

下面来谈谈第2个规则。假设对于头文件 a.h,它依赖于头文件 b.h(使用了 b.h 定义的类型或者宏等),因此在 a.h 的开头#include 了 b.h。比如,word_manage.h 使用了结构体 FILE,所以 word_manage.h 中#include 了 stdio.h。

很多人讨厌将#include 嵌套来写*,但在 a.h 依赖 b.h 的时候,如果不采用嵌套,就需要在所有使用 a.h 的地方都写成下面这样,

```
#include "b.h"
#include "a.h"
```

这样做是不是有点笨拙?不光每次写得很辛苦,而且将来某个时刻 a.h 不再依赖 b.h 后,上面的两行可能就永远地留在代码中了。另外,对于这种#include 方式,开发现场的“可爱的”程序员们不可避免地会有下面的举动,

唉~究竟是谁依赖谁,真是完全搞不明白哦!嗯~,先把已经编译通过的.c 文件中开头的#include 部分完全复制过来吧,嘿嘿.....

这样恶搞,肯定会在很多代码文件中留下大量无效的头文件引用。

啊?你问 Makefile 的依赖关系是怎么做的?这必然是通过工具自动生成的——这可是常识哦。

对于头文件,还有一个要点需要跟大家提前说明,那就是“应该把公有的和私有的分开”这个原则,具体的内容会在后面给大家介绍。

要 点

写头文件时必须遵守的原则:

- ❶ 所有的头文件中,必须要有防止重复#include 的保护。
- ❷ 所有的头文件只#include 自己直接依赖的头文件。

* 作为 C 的编程风格的指导书,著名的《Indian Hill 编程风格手册》(参照 <http://denno-tms.u-tokyo.ac.jp/arch/comptech/cstyle/cstyle-ja.htm>)中,也表达了“不要用#include 嵌套”这样的观点。可是,在隶属于 C 处理环境的头文件中,以及广泛使用的免费软件中,随处可以看到嵌套的头文件。(第8次印刷注:上面的网页好像已经不存在了,但在 <http://www.archive.org> 中输入上面的 URL,应该可以看到以前的影子)

5.1.3 数组版

对于 word_count 的“管理单词部分”的数据结构，先让我们考虑使用数组来实现一下。

在使用数组管理单词的时候，可以考虑采取下面的方式：

- ❶ 将单词和其出现的次数整理到结构体中。
- ❷ 将这些结构体组织成数组，并且管理各单词的出现频率。
- ❸ 为了将单词的追加和结果输出变得更加简单，使用将数组的元素按照单词的字母顺序排序的方式进行管理。

据此写的头文件 word_manage_p.h 内容如下（参照代码清单 5-4）：

代码清单 5-4 word_manage_p.h（数组版）

```

1:  #ifndef WORD_MANAGE_P_H_INCLUDED
2:  #define WORD_MANAGE_P_H_INCLUDED
3:  #include "word_manage.h"
4:
5:  typedef struct {
6:      char      *name;
7:      int       count;
8:  } Word;
9:
10: #define WORD_NUM_MAX    (10000)
11:
12: extern Word    word_array[];
13: extern int     num_of_word;
14:
15: #endif /* WORD_MANAGE_P_H_INCLUDED */

```

每当加入一个新的单词时，可以考虑对数组进行下面这样的操作：

- ❶ 从数组的初始元素开始遍历。如果发现同样的单词，将此单词的出现次数加 1。
- ❷ 如果没有发现相同的单词，就在行进到比当前单词“大的单词”（根据字母顺序，位置在后面的单词）的时刻，将当前单词插入到“大的单词”前面。

向数组中插入单词的方法如下（参照图 5-2）：

- ❶ 将插入点后方的元素顺次向后移动。
- ❷ 将新的元素保存在空出来的位置上。

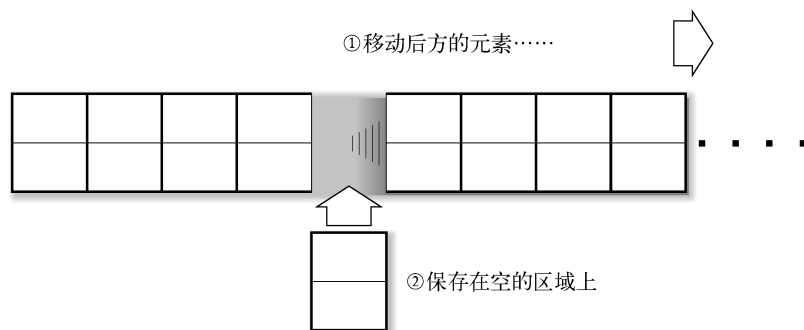


图 5-2 向数组插入元素

此时出现的问题是：每次向数组插入元素的时候，必须要移动后方的元素。

一般地，在初始化数组时必须确定元素个数。当然，你可以使用 4.1.3 节中说明的方法，动态地为数组分配内存空间，此后使用 `realloc()` “嗖嗖地”将数组伸长……其实，前面曾经给大家建议过，应该避免频繁地使用 `realloc()` 扩展内存区域（参照 2.6.5 节）。

如果采用下节中介绍的“链表”，就可以规避以上的问题*。

数组版的管理单词的源代码为代码清单 5-5、代码清单 5-6、代码清单 5-7 和代码清单 5-8*。

代码清单 5-5 initialize.c（数组版）

```

1: #include "word_manage_p.h"
2:
3: Word    word_array[WORD_NUM_MAX];
4: int     num_of_word;
5:
6: /*****
7:  * 初始化管理单词部分
8:  *****/
9: void word_initialize(void)
10: {
11:     num_of_word = 0;
12: }
```

* 其实数组也不是一无是处。排序后数组的检索速度非常快，这就是数组的一个优点，关于这一点可以参照 5.1.5 节。

* 本例中没有对数组越界情况做检查。

代码清单 5-6 add_word.c (数组版)

```

1: #include <stdio.h>
2: #include <stdlib.h>
3: #include <string.h>
4: #include "word_manage_p.h"
5:
6: /*
7:  * 将index后面的元素（包括index）依次向后方移动
8:  */
9: static void shift_array(int index)
10: {
11:     int src;    /* 被复制元素的索引 */
12:
13:     for (src = num_of_word - 1; src >= index; src--) {
14:         word_array[src+1] = word_array[src];
15:     }
16:     num_of_word++;
17: }
18:
19: /*
20:  * 复制字符串。
21:  * 尽管大多数处理环境都有 strdup()这样的函数，
22:  * 但是ANSI C规范中却没有定义这个函数，姑且先自己写一个。
23:  */
24: static char *my_strdup(char *src)
25: {
26:     char    *dest;
27:
28:     dest = malloc(sizeof(char) * (strlen(src) + 1));
29:     strcpy(dest, src);
30:
31:     return dest;
32: }
33:
34: /*****
35:  * 追加单词
36:  *****/
37: void add_word(char *word)
38: {
39:     int i;
40:     int result;
41:
42:     for (i = 0; i < num_of_word; i++) {
43:         result = strcmp(word_array[i].name, word);
44:         if (result >= 0)
45:             break;
46:     }
47:     if (num_of_word != 0 && result == 0) {
48:         /* 发现相同的单词 */
49:         word_array[i].count++;
50:     } else {
51:         shift_array(i);
52:         word_array[i].name = my_strdup(word);

```

```

53:         word_array[i].count = 1;
54:     }
55: }

```

代码清单 5-7 dump_word.c (数组版)

```

1: #include <stdio.h>
2: #include "word_manage_p.h"
3:
4: void dump_word(FILE *fp)
5: {
6:     int i;
7:
8:     for (i = 0; i < num_of_word; i++) {
9:         fprintf(fp, "%-20s%5d\n",
10:             word_array[i].name, word_array[i].count);
11:     }
12: }

```

代码清单 5-8 finalize.c (数组版)

```

1: #include <stdlib.h>
2: #include "word_manage_p.h"
3:
4: /*****
5:  * 管理单词部分的结束处理
6:  *****/
7: void word_finalize(void)
8: {
9:     int i;
10:
11:     /* 释放单词部分的内存区域 */
12:     for (i = 0; i < num_of_word; i++) {
13:         free(word_array[i].name);
14:     }
15:
16:     num_of_word = 0;
17: }

```

5.1.4 链表版

在前面一节中，我们指出了数组版中存在下面的问题：

- ❑ 中途向数组插入元素，后面的元素就必须依次向后方移动，导致效率低下*。
- ❑ 数组初始化时就需要决定元素个数。尽管可以使用 `realloc()` 不断地进行空间扩展，但在数组占用了较大的内存区域的情况下，还是要尽量避免使用这种手法。

* 当然，在删除元素时，如果需要填充空出来的位置，也有必要去移动后面的元素。此时，可以使用“删除标记”这种手法，但这不能算是正当的做法，如果能不用还是不要用吧。



图灵日语好书推荐



只需30天，从零开始编写一个五脏俱全的图形界面的操作系统



帮你掌握编程所需的“数学思维”



日本C语言入门第一书
原版畅销20万册

征服C指针

本书被称为日本最有营养的C参考书。作者是日本著名的“毒舌程序员”，其言辞犀利，观点鲜明，往往能让读者迅速领悟要领。

书中结合了作者多年的编程经验和感悟，从C语言指针的概念讲起，通过实验一步一步地为我们解释了指针和数组、内存、数据结构的关系，展现了指针的常见用法，揭示了各种使用技巧。另外，还通过独特的方式教会我们怎样解读C语言那些让人“纠结”的声明语法，如何绕过C指针的陷阱。

本书适合C语言中级学习者阅读，也可作为计算机专业学生学习C语言的参考。



图灵社区：www.ituring.com.cn

图灵微博：[@图灵教育](#) [@图灵社区](#)

反馈/投稿/推荐邮箱：contact@turingbook.com

热线：(010) 51095186 转 604

分类建议 计算机/编程语言/C语言

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-30121-5



9 787115 301215 >

ISBN 978-7-115-30121-5

定价：49.00元