TURING 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

C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。

TURING 图灵程序设计丛书

征服C指针

[日]前桥和弥 著
吴雅明 译

人民邮电出版社
北京

图书在版编目（CIP）数据

征服C指针 /（日）前桥和弥著 ; 吴雅明译. -- 北京 : 人民邮电出版社, 2013.2
（图灵程序设计丛书）
ISBN 978-7-115-30121-5

Ⅰ. ①征… Ⅱ. ①前… ②吴… Ⅲ. ①C语言－程序设计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2012)第280432号

内 容 提 要

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。

图灵程序设计丛书

征服C指针

◆ 著 [日] 前桥和弥
译 吴雅明
责任编辑 傅志红
执行编辑 乐 馨

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街14号
邮编 100061 电子邮件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京 印刷

◆ 开本：800×1000 1/16
印张：16.5
字数：333千字 2013年2月第1版
印数：1-4 000册 2013年2月北京第1次印刷

著作权合同登记号 图字：01-2012-4256号

ISBN 978-7-115-30121-5

定价：49.00元

读者服务热线：(010)51095186转604 印装质量热线：(010)67129223
反盗版热线：(010)67171154

版 权 声 明

（图灵公司感谢臧秀涛、刘建平对本书的审读）

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：“我敢打包票，**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。
——译者注

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。倒不是因为有“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

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”，难到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“这样才像 **C 语言的风格**”。

> 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

* *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。

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 2 章

做个实验见分晓——C 是怎么使用内存的

2.1 虚拟地址

关于“地址”的概念，我们在第 1 章已经做了如下说明：

> 变量总是保存在内存的“某个地方”。“某个地方”这样的说法不容易理解，因此，就像使用“门牌号”确定“住址”一样，在内存中，我们给变量分配“门牌号”。在 C 的内存世界里，“门牌号”被称为“地址”。

经过以上的说明，有人可能会有这样的想法：

> 哦，原来是这样子的！俺的机器是 128MB 的内存，这 128MB 的内存是以字节为单位从 0 开始分配了连续的编号呀。也就是说，如果在俺的机器里用十进制来粗略地计数，从 0 开始有大约 128 000 000 个地址吧*。那么使用 printf()打印指针的值，其结果究竟是什么样的呢？

* 准确地说，应该是 128×1024×1024，结果是 134 217 728。

但是，**如今的计算机可不是那么简单的哦**。

现在的 PC 机和工作站的操作系统大多都提供了多任务环境，可以同时运行多个应用程序（进程）。那么，假设同时运行两个应用程序，然后尝试打印各自的变量地址，会出现一致的结果吗？若遵循刚才的推论，不会。

好吧，我们一起做个实验。首先，请将代码清单 2-1 编译成可执行的文件。

这里可执行文件的名称为 vmtest。

代码清单 2-1　vmtest.c

```
 1:  #include <stdio.h>
 2:
 3:  int main(void)
 4:  {
 5:      int         hoge;
 6:      char        buf[256];
 7:
 8:      printf("&hoge...%p\n", &hoge);
 9:
10:      printf("Input initial value.\n");
11:      fgets(buf, sizeof(buf), stdin);
12:      sscanf(buf, "%d", &hoge);
13:
14:      for (;;) {
15:          printf("hoge..%d\n", hoge);
16:          /*
17:           * getchar()让控制台处于等待输入的状态.
18:           * 每次敲入回车键，增加 hoge 的值
19:           */
20:          getchar();
21:          hoge++;
22:      }
23:
24:      return 0;
25:  }
```

如今的操作系统，大多提供了多窗口环境，请你在自己的操作系统中打开**两个新的窗口**。若习惯用 UNIX，请使用 kterm（或者类似的工具）；若习惯用 Windows，请使用 DOS 窗口。另外，如果你**没有使用完全相同的方式启动**这些窗口，后面的实验可能会进行得不太顺利。Windows 的话，通过开始菜单打开两个新窗口就 OK。

然后，请在两个窗口分别试着运行刚才的程序（如果有必要，可以通过 cd 命令进入程序所在的目录）。

在我的环境里，得到了如图 2-1 所示的结果。

在第 8 行，通过 printf()输出变量 hoge 的地址，然后通过第 11 行的 fgets()，程序进入了等待输入的停止状态。显然，启动后的两个窗口肯定都处于运行状态，但 hoge 的地址却**两边完全相同**。

这两个进程的 hoge 看上去地址完全相同，但它们确实是在各自进程里面彼此独立无关的两个变量。根据 Input initial value.的屏幕窗口提示，我们接着输入一个任意值，这个值被赋予变量 hoge（第 12 行），在第 15 行又被 printf()输出。随后，getchar()让程序处于等待输入状态。每次敲击回

车键，hoge 的值都会增长并且被输出到窗口。正如我们看到的那样，这两个 hoge 内存地址明明是相同的，却各自保持着不同的值。

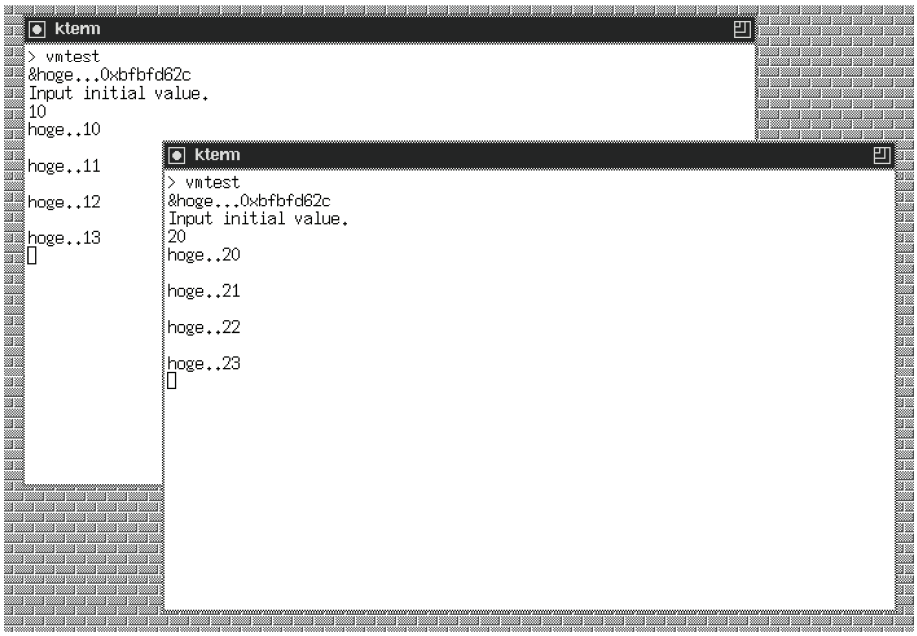

图 2-1 通过两个进程同时表示变量的地址

我在 FreeBSD 3.2-RELEASE 和 Windows 98 上运行了这个实验，都得到了以上的结果（编译器是 gcc）。

通对这样的实验发现，在如今的运行环境中，使用 `printf()`输出指针的时候，打印输出的并不是物理内存地址本身。

当今的操作系统都会给应用程序的**每一个进程分配独立的**“虚拟地址空间”。这和 C 语言本身并没有关系，而是操作系统和 CPU 协同工作的结果。正是因为操作系统和 CPU 努力地为每一个进程分配独立的地址空间，所以就算我毛手毛脚、糊里糊涂地制造了一个 bug，破坏了某个内存区域，顶多也就是让当前的应用程序趴窝，但不会影响其他进程*。

当然了，真正去保存内存数据的还是物理内存。操作系统负责将物理内存分配给虚拟地址空间，同时还会对每一个内存区域设定“只读”或者“可读写”等属性。

* Windows 95/98 时代，经常出现由于应用程序的异常导致操作系统瘫痪的情况……想不到操作系统也会有这样的硬伤，唉。

通常，因为程序的执行代码是只读的，所以有时候会和其他进程共享物理内存*。另外，当我们启动了几个笨重的应用程序而使内存出现不足时，操作系统把物理内存中还没有被引用的部分倒腾出来，保存到硬盘上。当程序再次需要引用这个区域的数据的时候，再从磁盘写回到内存（恐怕会把别的部分从磁盘里面取出来也说不定哦）。这个操作完全是在操作系统的后台进行的，对于应用程序来说，压根儿不知道背后发生的事。这时硬盘“咔嗒咔嗒”响，机器的反应也慢了下来。

* 现在，将程序的一部分以**共享库**的形式来共享使用，已经是很普遍的设计手法了。在写入之前，连内存数据也可以共享。

之所以能够这样，多亏了虚拟地址。正是因为避免了让应用程序直接面对物理内存的地址，操作系统才能够顺利地对内存区域进行重新配置（参照图 2-2）。

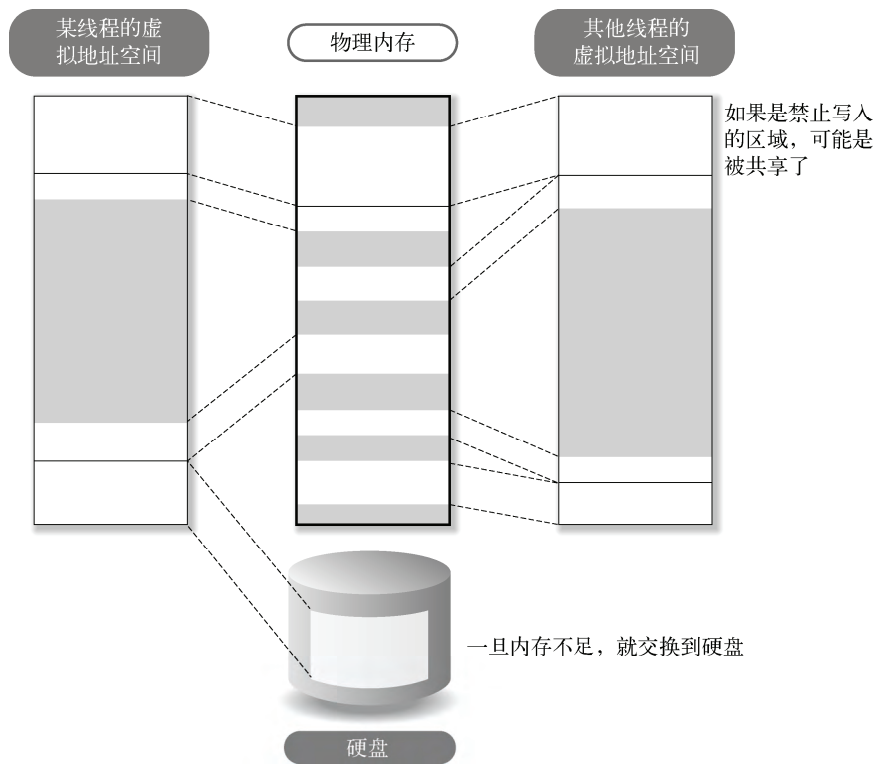

图 2-2　虚拟内存的概念图

要　点

在如今的运行环境中，应用程序面对的是虚拟地址空间。

补充 关于 scanf()

代码清单 2-1 中，使用下面的两条语句让用户输入整数值：

```
fgets(buf, sizeof(buf), stdin);
sscanf(buf, "%d", &hoge);
```

在一般的 C 入门书籍中，却经常看到以下写法：

```
scanf("%d", &hoge);
```

在这一次的例子中如果使用这种写法，程序是不会如愿运行起来的。因为这种写法在一开始漏掉了 getchar(),没有使程序执行进入输入等待状态。

这个问题是由 scanf()的自身实现造成的。

scanf()不是以行单位对输入内容进行解释，而是对连续字符流进行解释（换行字符也视为一个字符）。

scanf()连续地从流读入字符，并且对和格式说明符（%d）相匹配的部分进行变换处理。

例如，当格式说明符为%d 的时候，输入

```
123↵
```

从流中取得 123 部分的内容，并对它进行处理。换行符依旧会残留在流中。因此，后续的 getchar()会吞食这个留下的换行符。

此外，当 scanf()变换失败的时候（比如，尽管你指定了%d，但是输入的却是英文字符），scanf()会将导致失败的部分遗留在流中。

在读入过程中有几个对象被成功地变换，则 scanf()的返回值就为几。如果做一下错误检查，可能有人会写出下面的代码：

```
while (scanf("%d", &hoge) != 1) {
    printf("输入错误，请再次输入！");
}
```

分析一下上面的代码，我们就会知道，**一旦用户错误输入过一次，这段程序就会进入无限循环**。原因就是：错误输入的那部分字符串，将会被下一个 scanf()读到。

像代码清单 2-1 那样，如果将 fgets()和 sscanf()组合使用，就可以避免这个问题。

当然，一旦对 fgets()函数的第 2 个参数赋予超过指定长度的字符串，也是会出问题的。不过，如果像代码清单 2-1 这样指定了 256 个字符，对于自己使用的程序来说应该是足够了。

顺便说一下，在 scanf()中通过指定复杂的格式说明符，同样可以避免问题的发生。但我还是感觉不如使用 fgets()这种方式来得便利。

此外，为了解决这个问题，有人会使用 fflush(stdin);，其实这是个**错误的处理方法**。

fflush()是对输出流使用的，它不能用于输入流。标准中并没有定义用于输入流的 fflush()的行为。

2.2　C 的内存的使用方法

2.2.1　C的变量的种类

C 语言的变量具有区间性的作用域*。

* 在标准中，“作用域”（scope）和“连接”（linkage）是分别定义的，用语句块包围的是作用域，static 和 extern 分别控制静态连接和外部连接。
对于全局变量，作用域指文件作用域，链接指外部链接。
对于程序员来说，这些方式都是控制命名空间的，它们没有什么不同。在本书中，我们统一使用“作用域”这种叫法。

在开发一些小程序的时候，也许我们并不在意作用域的必要性。可是，当你书写几万行，甚至几十万行的代码的时候，没有作用域肯定是不能忍受的。

C 语言有如下三种作用域。

❶ 全局变量

在函数之外声明的变量，默认地会成为全局变量。全局变量在任何地方都是可见的。当程序被分割为多个源代码文件进行编译时，声明为全局变量的变量也是可以从其他源代码文件中引用的。

❷ 文件内部的静态变量

就算对于像全局变量那样被定义在函数外面的变量，一旦添加了 static，作用域就只限定在当前所在的源代码文件中。通过 static 指定的变量（包括函数），对于其他源代码文件是不可见的。在英语中，static 是“静态的”的意思，我实在想不明白为什么这个功能莫名其妙地被冠以“static”，这一点可以算是 C 语言的一个未解之谜。

❸ 局部变量

局部变量是指在函数中声明的变量。局部变量只能在包含它的声明的语句块（使用{}括起来的范围）中被引用。

局部变量通常在函数的开头部分进行声明，但也可以在函数内部某语句块的开头进行声明。例如，在“交换两个变量的内容时，需要使用一下临时变量”的情况下，将局部变量声明放在当前语句块开头还是比较方便的。

局部变量通常在它所在的语句块结束的时候被释放。如果你不想释放某个局部变量，可以在局部变量上加上 `static` 进行声明（在后面有详细说明）。

另外，除了作用域不同，C 的变量之间还有**存储期**（storage duration）的差别。

❶ 静态存储期（static storage duration）

全局变量、文件内的 `static` 变量、指定 `static` 的局部变量都持有**静态存储期**。这些变量被统称为**静态变量**。

持有静态存储期的变量的寿命从程序运行时开始，到程序关闭时结束。换句话说，静态变量一直存在于内存的同一个地址上。

❷ 自动存储期（auto storage duration）

没有指定 `static` 的局部变量，持有**自动存储期**。这样的变量被称为**自动变量**。

持有自动存储期的变量，在程序运行进入它所在的语句块时被分配以内存区域，该语句块执行结束后这片内存区域被释放*。

这个特征通常使用“栈”的机制来实现。2.5 节中会对此做详细说明。

接下来就不是“变量”了。C 中可以使用 `malloc()`函数动态分配内存。通过 `malloc()`动态分配的内存，寿命一直延续到使用 `free()`释放它为止。

在程序中，如果需要保持一些数据，必须在内存中的某个场所取得相应大小的内存区域。总结一下，在 C 中有三种内存区域的寿命。

❶ 静态变量

寿命从程序运行时开始，到程序关闭时结束。

❷ 自动变量

寿命到声明该变量的语句块被执行结束为止。

❸ 通过 `malloc()`分配的领域

寿命到调用 `free()`为止。

* 如果说明得细致一些，在几乎所有的处理环境中，并不是“程序执行进入语句块时”给自动变量分配内存区域，而是在“程序执行进入函数时”统一地进行内存区域分配的。

要　点

C 中有三种内存领域的寿命。

- ❑ 静态变量的寿命从程序运行时开始，到程序关闭时结束。
- ❑ 自动变量的寿命到声明该变量的语句块执行结束为止。
- ❑ 通过 malloc()分配的领域的寿命到调用 free()为止。

存储类型修饰符

在 C 的语法中，以下关键字被定义为“存储类型修饰符”。

```
typedef  extern  static  auto  register
```

可是，在这些关键字中，真正是“指定存储区间”的关键字，只有 static。*

extern 使得在其他地方定义的外部变量可以在本地可见；auto 是默认的，所以没有显式指定的必要；register 可以给出编译器优化提示（如今的编译已经很先进了，所以一般也不会使用这个关键字）；至于 typedef，它只是因为可以给编码带来便利才被归纳到存储类型修饰符中来的。

希望大家不要被这众多的“存储类型修饰符”搞得手忙脚乱。

* 可是，当你在函数的外面使用 static 的时候，就是使用作用域来控制了，而不是使用存储期。

2.2.2　输出地址

正如之前所说，C 的变量中有几个阶段的作用域，而且变量之间还有“存储期”的区别。此外，通过 malloc()可以动态分配内存。

在内存中，这些变量究竟是怎样配置的呢？不如让我们来写个测试程序验证一下（参照代码清单 2-2）。

代码清单 2-2　print_address.c

```
 1:  #include <stdio.h>
 2:  #include <stdlib.h>
 3:
 4:  int             global_variable;
 5:  static int      file_static_variable;
 6:
 7:  void func1(void)
 8:  {
 9:      int func1_variable;
10:      static int func1_static_variable;
11:
```

```
12:      printf("&func1_variable..%p\n", &func1_variable);
13:      printf("&func1_static_variable..%p\n", &func1_static_
         variable);
14:  }
15:
16:  void func2(void)
17:  {
18:      int func2_variable;
19:
20:      printf("&func2_variable..%p\n", &func2_variable);
21:  }
22:
23:  int main(void)
24:  {
25:      int *p;
26:
27:      /*输出指向函数的指针*/
28:      printf("&func1..%p\n", func1);
29:      printf("&func2..%p\n", func2);
30:
31:      /*输出字符串常量的地址*/
32:      printf("string literal..%p\n", "abc");
33:
34:      /*输出全局变量*/
35:      printf("&global_variable..%p\n", &global_variable);
36:
37:      /*输出文件内的 static 变量的地址*/
38:      printf("&file_static_variable..%p\n", &file_static_
         variable);
39:
40:      /*输出局部变量*/
41:      func1();
42:      func2();
43:
44:      /*通过 malloc 申请的内存区域的地址*/
45:      p = malloc(sizeof(int));
46:      printf("malloc address..%p\n", p);
47:
48:      return 0;
49:  }
```

在我的环境中运行结果如下：

```
&func1..0x8048414
&func2..0x8048440
string literal..0x8048551
&global_variable..0x804965c
&file_static_variable..0x8049654
&func1_variable..0xbfbfd9d8
&func1_static_variable..0x8049650
&func2_variable..0xbfbfd9d8
malloc address..0x805b030
```

一开始我们说要将变量的地址输出，代码清单 2-2 的第 28 ~ 29 行却输出了**指向函数的指针**。

尽管到目前为止没有提到过函数指针的问题，但是这次的代码中出现了函数指针。函数通过编译器解释成机器码，并且被配置在内存的某个地方的地址上。

在 C 中，正如数组在表达式中可以被解读成指针一样，“函数”也同时意味着“指向函数的指针”。通常，这个指针指向函数的初始地址。

第 32 行输出使用""包围的字符串（字符串常量）的地址。

在 C 中，“字符串”是作为“char 的数组”来表现的。字符串常量类型也是“char 的数组”，因为**表达式中的数组可以解读成“指向初始元素的指针”**，所以表达式中的“abc”，同样也意味着保存这个字符串内存区域的初始地址。

字符串常量在 C 中也被做了特别对待，它总让人感觉“不知道它被保存在内存的哪一片区域”，所以在这里我们也尝试输出它的地址。

第 35 行和第 38 行，分别输出了全局变量的地址和文件内 static 变量的地址。

第 41 行和第 42 行，调用了函数 func1()和 func2()。第 12 行和第 20 行输出自动变量的地址，第 13 行输出 static 局部变量的地址。

再回到 main()函数，在第 46 行输出利用 malloc()分配的内存区域的地址。

接下来，让我们来观察一下实际被输出的地址。

乍一看，都是 0x80……、0xbf……这样的地址，有点晕。

将地址按照顺序重新排列，并且整理成表 2-1。

表2-1　地址一览表

地　址	内　容	地　址	内　容
0x8048414	函数func1()的地址	0x804965c	全局变量
0x8048440	函数func2()的地址	0x805b030	利用malloc()分配的内存区域
0x8048551	字符串常量	0xbfbfd9d8	func1()中的自动变量
0x8049650	函数内的static变量	0xbfbfd9d8	func2()中的自动变量
0x8049654	文件内static变量		

通过观察，我们发现“指向函数的指针”和“字符串常量”被配置在非常近的内存区域。此外，函数内 static 变量、文件内 static 变量、全局变量等这些静态变量，也是被配置在非常近的内存区域。接下来就是 malloc() 分配的内存区域，它看上去和自动变量的区域离得很远。最后你可以发现，func1()和 func2()的自动变量被分配了完全相同的内存地址。

如果使用图来说明，应该是下面这样的感觉（参照图 2-3）。

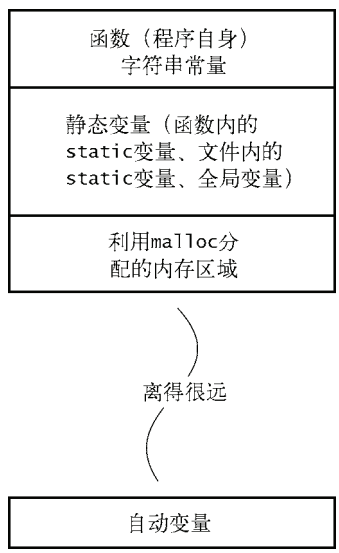

图 2-3 各种各样的地址

在后面的章节中，我们将会逐一对这些内存区域进行详细的说明。

2.3 函数和字符串常量

2.3.1 只读内存区域

在我的处理环境中，函数（程序自身）和字符串常量被配置在内存里相邻的地址上。

这并不是偶然的，如今的大多数操作系统都是将函数自身和字符串常量汇总配置在一个**只读内存区域**的。

由于函数本身不可能需要改写，所以它被配置在内存的只读区域。其实在很久以前，机器语言的程序改写自身程序代码的技术是被大量使用的*，但是现在的操作系统几乎都禁用了这种技术。

* 笔者就曾经有这样的经历。在 Z80 的情况下，只能通过直接指定地址的方式调用子程序……当然这些都是老皇历了。

那些能够修改自身代码的程序代码是非常晦涩难懂的。此外，如果执行程序是只读的，在同一份程序被同时启动多次的时候，通过在物理地址上共享程序能够节约物理内存。此外，由于硬盘上已经存放了可执行程序，就算内存不足，也不需要将程序交换到虚拟内存，相反可以将程序直接从内存中销毁*。

* 大体上，Windows、UNIX 等操作系统就是这样实现的。因此，在一部分 UNIX 中，如果在程序运行的时候将该程序改写（再编译/连接等），当前运行的程序就会崩溃。在如今的处理环境中，经常会对程序加锁。

根据处理环境的不同，字符串常量也有可能被配置在可改写的内存区域。像 DOS 这样不实施内存保护的操作系统也就罢了，可是在 UNIX 中，也有将字符串常量配置在非只读内存区域的情况。

假设有下面这样一个函数：

```
void func(void)
{
    char *str = "abc";

    printf("str..%s\n", str);

          ⋮  省略的很多逻辑
    str[0] = 'd';
}
```

一旦允许改写字符串常量，第一次调用函数输出“abc”，第二次调用函数却会输出“dbc”。可是根据代码的逻辑，给 str 赋值“abc”后，紧接着就会输出 str 的值。尽管如此，第二次调用还是输出了“dbc”，这就很让人头大。

2.3.2　指向函数的指针

函数可以在表达式中被解读成“指向函数的指针”，因此，正如代码清单 2-2 的实验那样，写成 func 就可以取得指向函数的指针。

“指向函数的指针”本质上也是指针（地址），所以可以将它赋给指针型变量。

比如有下面的函数原型：

```
int func(double d);
```

保存指向此函数的指针的变量的声明如下：

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

然后写成下面这样，就可以通过 func_p 调用 func，

```
int (*func_p)(double);    ←声明

func_p = func;    ←将 func 赋给 func_p

func_p(0.5);    ←此时，func_p 等同于 func
```

将“指向函数的指针”保存在变量中的技术经常被运用在如下场合：

- GUI 中的按钮控件记忆“当自身被按下的时候需要调用的函数”
- 根据“指向函数的指针的数组”对处理进行分配

后者的“指向函数的指针的数组”，像下面这样使用：

```
int (*func_table[])(double) = {
    func0,
    func1,
    func2,
    func3,
};
    .
    .
    .
func_table[i](0.5);    ←调用 func_table[i]的函数，参数为 0.5
```

使用上面的写法，不用写很长的 switch case，只需通过 i 的值就可以对处理进行分配。

哦？不明白为什么？

确实，像

```
int (*func_p)(double);    ←指向函数的指针
```

还有，

```
int (*func_table[])(double);    ←指向函数的指针的数组
```

这样的声明，是不能用普通的方法来读的。

关于这种声明的解读方式，会在第 3 章进行说明。

2.4　静态变量

2.4.1　什么是静态变量

静态变量是从程序启动到运行结束为止持续存在的变量。因此，静态变量总是在虚拟地址空间上占有固定的区域。

静态变量中有全局变量、文件内 static 变量和指定 static 的局部变量。因为这些变量的有效作用域各不相同，所以编译和连接时具有不同的意义，但是**运行的时候它们都是以相似的方式被使用的**。

2.4.2　分割编译和连接

在 C 语言中，一个程序可以由多个源代码文件构成，并且这些源代码文件在各自编译之后可以连接起来。这一点对于大规模的编程工作来说，是举足轻重的。难道不是吗？如果 100 个程序员一拥而上，一起捣鼓同一个源代码文件，那真是不可想象。

此外，关于函数和全局变量，如果**它们的名称相同**，即使它们跨了多个源代码文件也被作为相同的对象来对待。进行这项工作的是一个被称为“链接器”的程序。

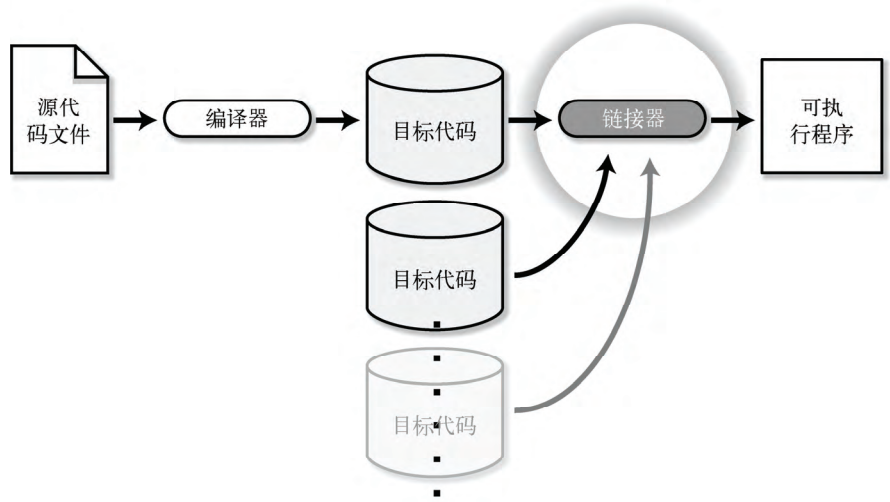

图 2-4　链接器

为了在链接器中将名称结合起来，各目标代码大多都具备一个**符号表**（symbol table）（详细内容需要依赖实现细节）。比如在 UNIX 中，可以使用 nm 这样的命令窥视符号表的内容。

不好意思，这里是 UNIX 特有的话题。为了测试，我们使用 cc -c 编译 print_address.c（参照代码清单 2-2），生成 print_addres.o 文件，之后对这个文件使用 nm 命令。在我的环境中，结果输出如下。

```
> cc -c print_address.c
> nm print_address.o
00000004 b file_static_variable
00000000 T func1
00000000 b func1_static_variable.4
0000002c T func2
00000000 t gcc2_compiled.
00000004 C global_variable
00000048 T main
         U malloc
         U printf
```

通过观察输出结果，我们很容易发现符号表中记录了文件内的 static 变量、局部 static 变量这些看上去不需要连接的对象。

如果命名空间不一样，确实不需要和其他文件的符号连接，但是对于静态变量来说，因为必须要给它们分配一些地址，所以符号表中记录了这些变量。可是同时我们也发现全局变量的标记有些特别。与全局变量使用 C 进行标记不同的是：和外部没有连接的符号，无论是局部的还是文件内部的 static 变量，都使用了 b 进行标记。

局部 static 变量 func1_static_variable 的后面被追加了.4 这样的标记，这是因为在同一个.o 文件中，局部 static 变量的名称有可能会发生重复，所以在它后面追加了识别标记。

函数名后面追加了 T 或者 U。如果函数是在当前文件中定义的，就在其函数名后加 T；如果函数定义在当前文件之外，只是在当前文件内部调用此函数，就在此函数后面加 U。

gcc2_compiled.是我们没有见过的符号，你可以把它当成处理环境随意加上的标记，把它放在一边。

链接器就是根据这些信息，给这些到目前为止还只是个“名称”的对象分配地址*。

* 现在对共享库做动态链接变成一件很自然的事，而现实中可没有这么单纯……

请注意自动变量完全没有出现在符号表中。这是因为自动变量的地址是在运行时被决定的，它属于链接器管辖范围以外的对象。关于这一点，我们在下一节阐述。

2.5　自动变量（栈）

2.5.1　内存区域的“重复使用”

通过代码清单 2-2 的实验，我们看到 func1()的自动变量 func1_variable 和 func2()的自动变量 func2_variable 存在于完全相同的地址上。

在声明自动变量的函数执行结束后，自动变量就不能被使用了。因此，func1()执行结束后，func2()重复使用相同的内存区域是完全没有问题的。

> 要　点
> **自动变量重复使用内存区域。**
> **因此，自动变量的地址是不一定的。**

2.5.2　函数调用究竟发生了什么

自动变量在内存中究竟是怎样被保存的？为了更加详细地了解这一点，还是让我们用下面这个测试程序做一下实验。

代码清单 2-3　auto.c

```
 1:  #include <stdio.h>
 2:
 3:  void func(int a, int b)
 4:  {
 5:      int c, d;
 6:
 7:      printf("func:&a..%p &b..%p\n", &a, &b);
 8:      printf("func:&c..%p &d..%p\n", &c, &d);
 9:  }
10:
11:  int main(void)
12:  {
13:      int a, b;
14:
15:      printf("main:&a..%p &b..%p\n", &a, &b);
16:      func(1, 2);
17:
18:      return 0;
19:  }
```

在我的环境中运行结果如下：

```
main:&a..0xbfbfd9e4 &b..0xbfbfd9e0
func:&a..0xbfbfd9d8 &b..0xbfbfd9dc
func:&c..0xbfbfd9cc &d..0xbfbfd9c8
```

如果把运行结果用图来说明，应该是图 2-5 这样的。

如果将 main()的局部变量、func()的局部变量以及 func()的形参的地址进行比较，func()相关的地址看上去相对小一些。

C 语言中，在现有被分配的内存区域之上以“堆积”的方式，为新的函数调用分配内存区域*。在函数返回的时候，会释放这部分内存区域供下一次函数调用使用。图 2-6 粗略地表现了这个过程。

* 图 2-3 中，自动变量的区域上方有一片广大的区域。在这片区域中，栈不断地增长。

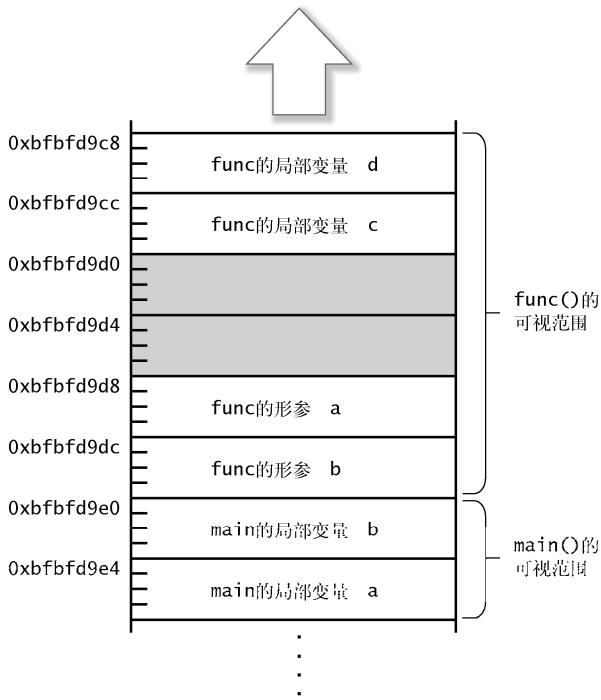

图 2-5 局部变量和参数的地址

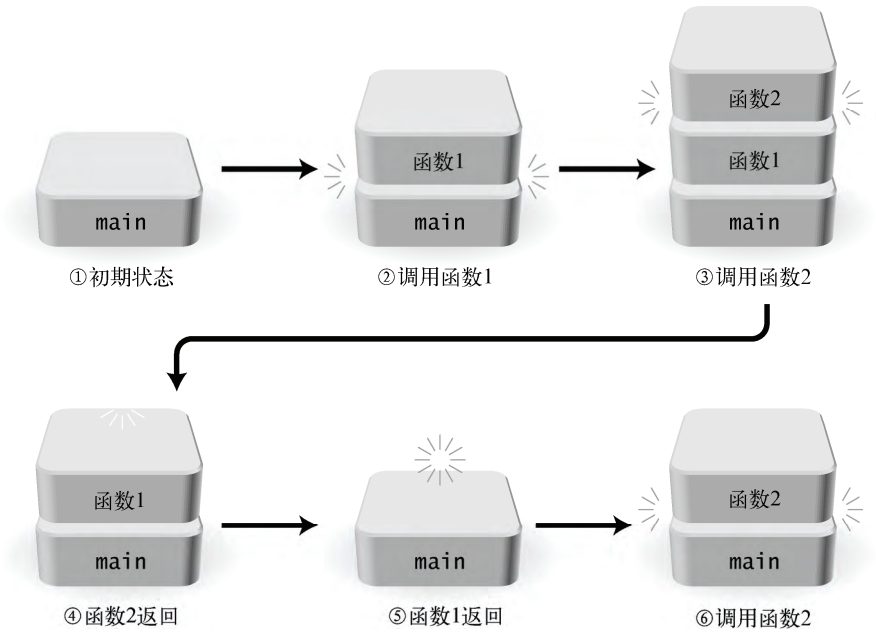

图 2-6　函数调用的概念图

对于像这样使用“堆积”方式的数据结构，我们一般称为**栈**。

程序员们有时候也使用数组等方式实现栈。但是，大部分的 CPU 中已经嵌入了栈的功能，C 语言通常直接使用。

> 要　点
> **C 语言中，通常将自动变量保存在栈中。**

通过将自动变量分配在栈中，内存区域可以被重复利用，这样可以节约内存。

此外，将自动变量分配在栈中，对于递归调用（参照 2.5.4 节）也具有重要的意义。

下面归纳了最简约的 C 语言函数调用的实现*：

❶ 在调用方，参数**从后往前**按顺序被堆积在栈中*。

❷ 和函数调用关联的返回信息（返回地址等）也被堆积在栈中（对应于图 2-5 的灰色部分）。所谓的“返回地址”，是指函数处理完毕后应该返回的地址。正因为返回地址被堆积在栈中，所以无论函数从什么地方被调用，它都能返回到调用点的下一个处理。

* 当然，最近的编译器做了各种优化，它们不一定完全以这里描述的方式动作。但是基本思想是一致的。

* 关于为什么参数是从后往前堆积的，请参照 2.5.3 节。

❸ 跳转到作为被调用对象的函数地址。

❹ 栈为当前函数所使用的自动变量增长所需大小的内存区域。❶到❹所增长的栈的区域成为当前函数的可引用区域。

❺ 在函数的执行过程中，为了进行复杂的表达式运算，有时候会将计算过程中的值放在栈中。

❻ 一旦函数调用结束，局部变量占用的内存区域就被释放，并且使用返回信息返回到原来的地址。

❼ 从栈中除去调用方的参数。

图 2-7 展示了 func(1, 2)被调用时，栈的使用情况。它与图 2-5 相吻合。

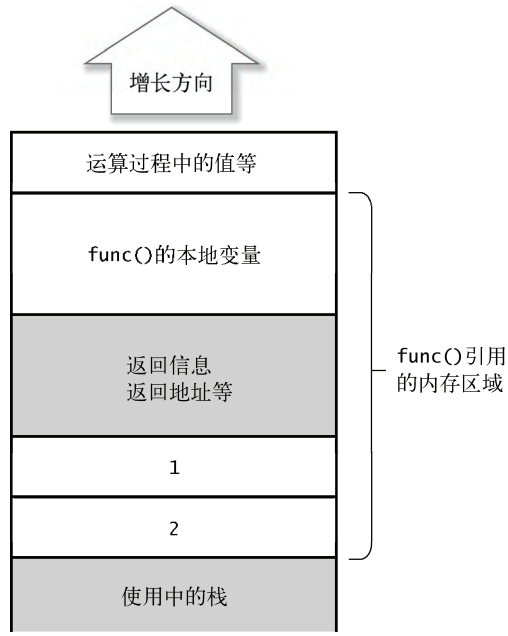

图 2-7 调用 func(1, 2)时栈的使用情况

此外，在调用函数时，请留意为形参分配新的内存区域。我们经常听说“C 的参数都是传值，向函数内部传递的是实参的副本”其中的复制动作，其实就是在这里发生的。

一旦函数执行结束，自动变量的内存区域就会被释放！

初学者经常写出下面这样的程序。

```
/*将 int 变换成字符串的程序*/
char *int_to_str(int int_value)
{
    char  buf[20];

    sprintf(buf, "%d", int_value);

    return buf;
}
```

恐怕……这个程序不能正常地跑起来*。

原因估计你也知道了：**自动变量 buf 的内存区域在函数执行结束后就会被释放**。

为了让这个程序先动起来，可以将 buf 声明成下面这样：

```
static char buf[20];
```

因为 buf 的内存区域一直会被静态地保持，所以即使函数执行结束，buf 的内存区域也不会被释放。

在这种方式下，当你连续两次调用这个函数，第二次函数调用会“悄悄地”修改第一次函数调用得到的字符串。

```
str1 = int_to_str(5);
str2 = int_to_str(10);
printf("str1..%s, str2..%s\n", str1, str2);  ←— 这里究竟会输出什么样的结果呢?
```

程序员当然希望输出的是 str1..5，str2..10。可惜最终事与愿违。

这样的函数引起了一个出乎意料的 bug。此外，在多线程编程的情况下，当前函数同样也是有问题的*。

借助于 malloc()动态地分配内存区域，可以解决上面的问题（参照 2.6 节），但是使用方需要同时使用 free()*。

其实，作为最终的解决方案，如果函数调用方事先知道数组的长度上限，可以在调用方声明一个数组，然后通过函数将结果输出到此数组中。对于 C 语言来说，这是一个最实用的方案。

* 在某些环境下可能也能跑起来，但这只是偶然。

* 标准库中有一个函数 strtok()，此函数也存在类似的问题，招致了很多抱怨。

* 由于 Java 具有垃圾回收机制，所以不需要我们显式地调用 free()。

一旦破坏了自动变量的内存区域

假设，通过自动变量声明下面的数组：

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

如果没有做数组长度检查，将数据写入了超过数组内存区域的地方，究竟又会引发怎样的悲剧呢？

如果只是超过一点点，可能也就是破坏相邻的自动变量的内容。但是，一旦将前方的内存区域径直地破坏下去……

自动变量是保存在栈中的，如果是数组，表示如下图（参照图 2-8），

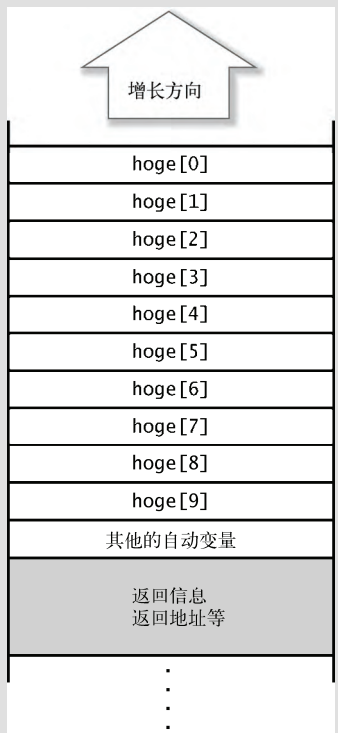

图 2-8 栈中的数组

在这里，如果大范围地超过数组内存区域写入数据，内存中直到存储函数的返回信息的区域都有可能被破坏。也就是说，这个函数**不能返回了**。

当我们在追踪一个 bug 的时候，发现函数处理已经走到了最后，可偏偏不能返回到调用方。此时，就应该质疑是不是已经不小心将数据写入了保存函数返回地址的内存领域。

这种情况下，就算是调试也经常不能定位程序发生错误的地方。因为调试也是使用堆积在栈中的信息，如果大量地破坏了栈的内存领域，是很难追踪到 bug 发生的具体地点的。

此外，由于越界向数组类型的自动变量写入数据，以至于返回信息（返回地址）被覆盖，甚至可能造成安全漏洞。

对于那些没有切实做好数组长度检查的程序，一些无良的骇客会故意地传递很大的数据，造成函数的返回地址被恶意的数据所改写*。也就是说，骇客完全可以让你的程序去运行任意的机器语言代码。

* 请参照“‘黑客不是骇客’协会”http://www-vacia.media.is.tohoku.zc.jp/~s-yamane/hackerML/。

标准库中有一个 gets()函数，它和 fgets()同样都可以从标准输入读取一行输入，但与 fgets()不同的是你不能向它传递缓冲的大小。因此，gets()是**不可能做数组长度检查**的。在 C 语言中，所谓将数组作为参数进行传递，只不过是传递指向数组初始元素的指针。作为被调用方，是完全不知道数组究竟有多长的。

gets()函数多用于将标准输入，也就是“从外部”来的输入保存在数组中。因此，在使用 gets()的程序中，通过故意将包含尺寸很大的行的数据传递给 gets()，就可以达到数组越界且改写返回地址的目的。

1988 年名震互联网的“互联网蠕虫”，就是利用了 gets()的这个弱点。

因为这些原因，gets()已经被视为落后于时代的函数。我使用的编译器（gcc version 2.7.2.1）在编译和连接过程中，一旦发现当前代码使用了 gets()，就会提示以下警告：

```
warning: this program uses gets(), which is unsafe.
```

如果你一意孤行还是要运行这个程序，还会提示同样的警告。

不只是 gets()，还有比如对于 scanf()这个函数，如果使用"%s"也会招致同样的结果。但是，你可以通过对 scanf()指定"%10s"来限制字符串的最大长度。

下面再给大家举一个有点恶搞的例子（参照代码清单 2-4）。

并不是所有的运行环境都是这样，但是至少在 Windows 98 和 FreeBSD（编译器是 gcc）下，hello()会被调用（而且不止一次）。

为什么会这样？——请读者自己思考。

另外，这个程序必然会以崩溃告终，所以请在具有内存保护的操作系统上运行。

代码清单 2-4 stackoverflow.c

```
 1:  #include <stdio.h>
 2:
 3:  void hello(void)
 4:  {
 5:      fprintf(stderr, "hello!\n");
 6:  }
 7:
 8:  void func(void)
 9:  {
10:      void        *buf[10];
11:      static int  i;
12:
13:      for (i = 0; i < 100; i++) {     ←越界!
14:          buf[i] = hello;
15:      }
16:  }
17:
18:  int main(void)
19:  {
20:      int buf[1000];
21:
22:      func();
23:
24:      return 0;
25:  }
```

2.5.3 可变长参数

大部分的C语言入门书籍往往在一开始就频繁地使用printf()这个输出文字信息的函数，利用这个函数，可以将可变个数的参数填充到字符串中的指定位置。因为这个奇怪的特征，在教初学者 C 语言的时候，经常会有这样**让人费解的说明**：

> 快看，在 C 中的输入输出并不是语言的一部分，而是作为函数库被提供的。因此，printf()这个函数和程序员平时写的函数没有什么不一样哦。

在这里姑且把这个问题先放一边。

正如 2.5.2 节中说明的那样，C语言的参数是从后往前被堆积在栈中的。

另外，在C语言中，应该是由调用方将参数从栈中除去。

顺便提一下，Pascal 和 Java 是从前往后将参数堆积在栈中的。这种方式能够从前面开始对参数进行处理，所以对于程序员来说比较直观。此外，将参数从栈中除去是被调用方应该承担的工作。大部分情况下，这种方式的效率还是比较高的。

为什么 C 故意采取和 Pascal、Java 相反的处理方式呢？其实就是为了实现可变长参数这个功能*。①

* Pascal 和 Java 中不能写带有可变长参数的函数。

比如，对于像

```
printf("%d, %s\n", 100, str);
```

这样的调用，栈的状态如图 2-9 所示。

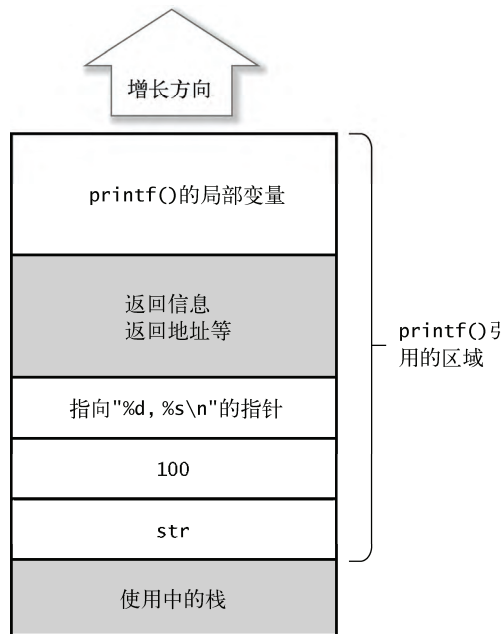

图 2-9　调用持有可变长参数的函数

重要的是，无论需要堆积多少个参数，**总能找到第一个参数的地址**。从图中我们可以看出，从 `printf()`的局部变量来看，第一个参数（指向`"%d, %s\n"`的指针）一定**存在于距离固定的场所**。如果从前往后堆积参数，就肯定不能找到第一个参数。

① Java 在 JDK1.5 之后的版本也开始支持可变长函数了。——译者注

之后，在printf()中，通过对找到的第一个参数"%d, %s\n"进行解析，就可以知道后面还有几个参数。

由于其余的参数是连续排列在第一个参数后面的，所以你可以顺序地将它们取出。

可是，现实中有一些部分是依赖运行环境的。ANSI C 为了提高可移植性，通过头文件 stdarg.h*提供了一组方便使用可变长参数的宏。

* ANSI C 以前的 C 使用 varargs.h 这个头文件，它和 stdarg.h 在使用上有很大的差异。

下面就让我们实际使用 stdarg.h 写一个具有可变长参数的函数。

我们考虑写一个山寨版的 printf()，取名为 tiny_printf()。

tiny_printf()的第一个参数指定后续的各参数的类型，第二个参数开始指定需要输出的值。

```
tiny_printf("sdd", "result..", 3, 5);
```

在这个例子中，通过第一个参数"sdd"，指定后续的参数类型为“字符串，int，int”（和 printf()一样，s 表示字符串，d 表示数字）。从第二个参数开始，分别向函数传递字符串"result"和两个整数。

函数执行后的结果如下：

```
result.. 3 5
```

和 printf()不同，因为指定换行字符的输出比较繁琐，tiny_printf()在默认情况下会主动进行换行。

代码如下（参照代码清单 2-5）。

代码清单 2-5 tiny_printf.c

```
 1:  #include <stdio.h>
 2:  #include <stdarg.h>
 3:  #include <assert.h>
 4:
 5:  void tiny_printf(char *format, ...)
 6:  {
 7:      int i;
 8:      va_list     ap;
 9:
10:      va_start(ap, format);
11:      for (i = 0; format[i] != '\0'; i++) {
12:          switch (format[i]) {
13:            case 's':
```

```
14:               printf("%s ", va_arg(ap, char*));
15:               break;
16:             case 'd':
17:               printf("%d ", va_arg(ap, int));
18:               break;
19:             default:
20:               assert(0);
21:           }
22:       }
23:       va_end(ap);
24:       putchar('\n');
25:   }
26:
27:   int main(void)
28:   {
29:       tiny_printf("sdd", "result..", 3, 5);
30:
31:       return 0;
32:   }
```

从第 5 行开始函数的定义。对于形参声明中的…这种写法，可能大家会感到有些陌生，但原型声明也是写成这样的。如果原型声明的参数中出现…，对于这部分的参数是不会做类型检查的。

在第 8 行，声明 va_list 类型的变量 ap。stdarg.h 文件中定义了 va_list 类型，我的环境中是这样定义的，

```
typedef char *va_list;
```

也就是说，在我的环境中，va_list 被定义成“指向 char 的指针”。

在第 10 行，va_start(ap, format);意味着“使指针 ap 指向参数 format 的下一个位置”。

因此，我们得到了可变长部分的第一个参数。在后面的第 14 行和第 17 行，给宏 va_arg()指定 ap 和参数类型，就可以顺序地取出可变长部分的参数。

第 23 行的 va_end()只不过是个“摆设”，只不过因为在标准里指出了对于具有 va_start()的函数需要写 va_end()。

顺便说一下，在我的环境中，宏 va_end()是这样被定义的：

```
#define       va_end(ap)
```

实际上，va_end()就是个空定义。

以上就是开发具有可变长参数的函数的方法。

这里必须要注意的是，因为无论怎样都是从函数内部顺序地读出各参数，所以要实现可变长参数的函数就必须知道参数的类型和个数。

printf()将输出内容整理得很利落，但是如果只是输出很少的内容，使用printf()就显得有点小题大做。这种情况下，Pascal的writeln()和BASIC的print语句倒是比较简洁。

如

```
writeln("a..", 10, " b..", 5);
```

输出结果为：

```
a..10 b..5
```

但是C语言是**无法提供这样的函数的**。这是因为在writeln()内部无法知道参数的类型和个数。

话说回来，一旦学会了可变长函数的开发方法，还是感觉自己挺了不起的，无论如何总是想找个机会秀一下。我就是这样的*。

可是，对于可变长参数的函数，是不能通过原型声明来校验参数的类型的。另外，函数的执行需要被调用方完全信任调用方传递的参数的正确性*。因此，对于使用了可变长参数的函数，调试会经常变得比较麻烦。一般只有在这种情况下，才推荐使用可变长参数的函数：如果不使用可变长参数得函数，程序写起来就会变得困难。

要 点

在决定开发可变长参数的函数之前，有必要讨论是否真的需要这么做。

补充 assert()

代码清单2-5中，有assert(0);这样的语句。

assert()是在assert.h中定义的宏，使用方式如下：

```
assert(条件表达式);
```

若条件表达式的结果为真，什么也不会发生；若为假，则会输出相关信息并且**强制终止程序**。

我们经常会看到如下注释：

* 以前年少无知，看到XView的函数使用方法，还觉得挺帅的。

* 偶尔，对于printf，编译器有时也会亲切地给出“格式定义符和实际参数的类型不一致”的警告。其实这只不过是编译器特别照顾了printf()。毕竟printf()是一个被“超”频繁使用的函数。

```
/*这里的 str[i]必定为'\0' */
```

虽然这种方式可以提高程序的可读性，但也不是说你可以什么检查也不做。这种情况下，如果使用，

```
assert(str[i] == '\0');
```

可以帮我们挑出 bug。

代码清单 2-5 的 assert(0)的参数为 0（假），因此只要程序执行经过这里就会被强制终止。如果程序自身没有 bug，程序是不会走到过程语句 switch 的 default 部分的，我认为这种编程方式是值得肯定的。

对于使用“强制终止程序运行”的方式来进行异常处理，很多人是持有反对意见的。确实，对于用户的一些奇怪的操作，比如传入一个奇怪的文件等行为，突然草率地终止程序会让用户感到不知所措。

可是，如果不是由于这样的“外部因素”，只要**程序自身没有 bug 就绝对不会发生异常**的情况下，我想还是应该果断地终止程序。使用“通过返回值返回错误状态”等冗长的处理方式，一旦调用方偷懒没做检查*，就会放过很多本来可以很容易就发现的 bug。

像 C 这样可以导致内存区域破坏的语言，一旦发现很明显的 bug，就表明当前程序已经无法保证正常运行了。如果栈被破坏了，就算你想返回错误状态，也有可能无济于事，因为当前的函数可能连 return 也完成不了*。

让我们在那些潜在 bug 还没有给你带来更大的麻烦之前，麻利地把它们消灭吧!*

* 这个经常有。☹

* 如果是 Java 这样能够保证内存区域不会破坏，并且具备完善的异常处理机制的语言，将例外返回反而是一个正确的做法。

* 如果是在编辑重要的数据，应该采取紧急安全措施。当然了，文件名称需要和普通情况下的不一样。

* “拜托你使用调试好吗？”我们经常会听到这样的声音。实际上由于这样那样的原因，我们不能完全否定“printf()调试”这样的做法。

尝试开发一个用于调试的函数吧

经常看到很多人为了调试方便，使用 printf()输出变量的值*。

但如果直接使用 printf()或者 fprintf()，当调试结束后要删除它们可就麻烦了。

```
#ifdef DEBUG
    printf(...);
#endif /* DEBUG */
```

在有的书籍中推荐了上面的写法，但如果代码中充斥了大量这样的东西，就会变得很难读懂。

我们可能更需要下面这样类似于 printf()的写法：

```
debug_write("hoge..%d, piyo..%d\n", hoge, piyo);
```

可是，因为 printf()是持有可变长参数的函数，无法实现通过 debug_write()函数重新调用 printf()的功能。所以说，自己实现 printf()还是有些难度的。啊，怎么办呢？

这种情况下，标准库中提供了 vprintf()和 vfprintf()这两个函数。

```
void debug_write(char *fmt, ...)
{
    va_list   ap;
    va_start(ap, fmt);

    vfprintf(stderr, fmt, ap);    ←— 参数中传递 ap 的地方是指针

    va_end(ap);
}
```

此外，如果在此函数中引用一个全局变量，就可以用来控制是否输出调试信息。但伴随着 debug_write()调用而导致的开销是无法避免的。

如果是宏，在编译时是可以完全回避这部分系统开销的。但可惜的是，我们无法向 C 的宏传递可变长参数。

在下面，我们先定义一个宏：

```
#ifdef DEBUG
#define DEBUG_WRITE(arg)  debug_write arg
#else
#define DEBUG_WRITE(arg)
#endif
```

然后使用如下技巧：

```
DEBUG_WRITE(("hoge..%d\n", hoge));
```

这里必须要加两重括号，这可是个难点哟。

C 语言的常见问题集《C 语言编程 FAQ》[6]中介绍了，在非调试模式时将 DEBUG_WRITE 定义为(void)的技巧（p.151）*。

通过这个技巧，上面的语句会展开成(void)("hoge..%d\n", hoge)的形式，它意味着“将逗号运算符连接的表达式强制转型成 void”。优秀的编译器会通过优化完全忽略这条语句。

随便提一下，在 ISO C99 中的宏已经可以取得可变长参数。不过好像几乎都不是专门用于调试的。

或者，对于比较简单的输出，如果事先定义这样一个针对 int、double、char*的宏，可以使后面的开发变得简便一些。

```
#define SNAP_INT(arg) fprintf(stderr, #arg "...%d\n", arg)
```

* 非常遗憾，《C 语言编程 FAQ》的日语版（英文版名为 Cprogramming FAQS），已经绝版了。（第 8 次印刷注：由新纪元出版社重新出版。ISBN：4775302507）

这个宏可以像下面这样使用（如果有兴趣知道原理，你可以调查一下预处理器手册）：

```
SNAP_INT(hoge);
```

输出结果为：

```
hoge...5
```

此外还有一点需要补充。使用 printf()输出调试信息，由于输出被缓冲，所以在程序异常结束的时候会发生关键信息没有被输出的情况。通过 fprintf()将调试信息输出到 stderr 或者文件时*，建议事先使用 setbuf()停止缓冲。

* 依照标准，stderr 是不进行缓冲的。

2.5.4　递归调用

C 语言通常在栈中分配自动变量的内存区域。这除了可以重复使用该区域来节省内存以外，还可以实现递归调用。

递归调用就是函数对自身的调用。

可是很多程序员对递归都感到比较纠结。

纠结的原因，除了递归调用本身的难度之外，也有“不明白究竟有什么作用”这一点。

世间的 C 语言入门书籍也经常用阶乘运算等作为递归调用的例题，我觉得这是有点不太合适的。为什么呢？阶乘什么的，**用循环来写不是更简单易懂吗**？

关于现实中递归调用的使用，就拿我来说，遍历树结构、图形结构中各元素的情况是占绝大多数的，但由于篇幅的限制，就不用这些应用作为例子了。

这里举一个“如果不用递归调用，程序就不太好写”的例子：快速排序。

所谓排序，就是将很多数据以一定的顺序（比如从小到大等）进行排列。作为排序的手法，冒泡算法、插入算法、堆排序等很多方法都为大家所知。顾名思义，快速排序是其中最快速的排序算法*。

* 实际上，利用像本书中举的这么简单的程序来应对特殊的案例，似乎显得过于幼稚。尽管如此，几乎在所有的情况下，处理速度还是比冒泡算法等要快很多。

快速排序的基本思路如下：

❶ 从需要排序的数据中，找到一个**适当的**基准值（pivot）。

❷ 将需要排序的数据按照小于 pivot 和大于 pivot 进行分类。
❸ 对分类后的两类数据各自再次进行上述的❶和❷的处理。

如果排列对象是数组，上面的第 2 个步骤就点麻烦。下面是在不使用大的临时内存区域的情况下，对数组进行分类的思路（假设是升序排序）：

❶ 从左向右，检索比 pivot 大的数据。
❷ 从右向左，检索比 pivot 小的数据。
❸ 如果两个方向都能检索到数据，将找到的数据交换。
❹ 重复进行❶～❸的操作，直到从左开始检索的下标和从右开始检索的下标发生冲突为止。

可能有人会对这种算法的敏捷性表示怀疑，但是这种算法的确很高效。在我的运行环境中，测试对 5 万个随机整数进行排序，冒泡算法排序花了 117 秒，快速排序算法用了 65 毫秒就完成了工作（1 毫秒=千分之一秒）。

要 点
根据算法不同，程序的处理速度会有天壤之别。所以，选择合适的算法是非常重要的。

代码清单 2-6 是实现快速排序的源代码。

代码清单 2-6 快速排序的程序

```
 1:  #define SWAP(a, b) {int temp; temp = a; a = b; b = temp;}
 2:
 3:  void quick_sort_sub(int *data, int left, int right)
 4:  {
 5:      int left_index = left;
 6:      int right_index = right;
 7:      int pivot = data[(left + right) / 2];
 8:
 9:      while (left_index <= right_index) {
10:          for ( ; data[left_index] < pivot; left_index++)
11:              ;
12:          for ( ; data[right_index] > pivot; right_index--)
13:              ;
14:
15:          if (left_index <= right_index) {
16:              SWAP(data[left_index], data[right_index]);
17:              left_index++;
18:              right_index--;
19:          }
20:      }
21:
22:      if (right_index > left) {
```

```
23:             quick_sort_sub(data, left, right_index);
24:         }
25:         if (left_index < right) {
26:             quick_sort_sub(data, left_index, right);
27:         }
28:     }
29:
30:     void quick_sort(int *data, int data_size)
31:     {
32:         quick_sort_sub(data, 0, data_size - 1);
33:     }
```

这个例程是对 int 的数组进行排序。

向函数 quick_sort()传递 int 型数组 data 和数组的长度，数组中的元素会被升序排序。

quick_sort_sub()将数组 data 的元素从 data[left]到 data[right]的部分进行排序（包括 data[left]和 data[right]）。

quick_sort_sub()首先决定 pivot 的值，然后把数组的元素分成大于 pivot 和小于 pivot 两个类别*。

* 尽管这里的算法占了整篇代码的一大半，但为了不跑题，笔者对算法不进行详细的说明，还是请读者自己去解读。

然后，对于分类后还存在两个以上元素的数组，**递归地调用自身**（第 23 行和第 26 行）。

因为程序优先对数组的左侧进行处理，所以数组将会被排序成图 2-10 中的样子。

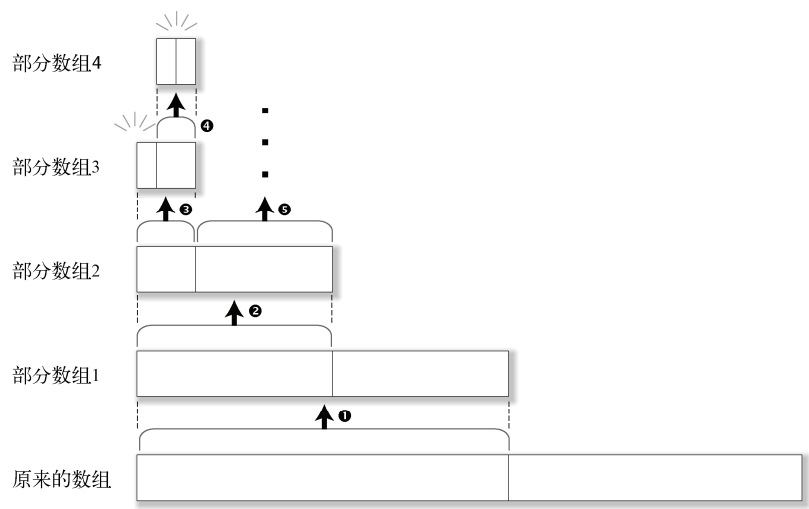

图 2-10　快速排序的概念图

首先，原始的数组被分类成大于 pivot 和小于 pivot 两个部分。对分类后的左侧（部分数组 1）递归调用自身再次进行分类（❶）。然后对分类的结果的左侧再分类，再分类……就这样不断地递归下去，一直进行到左侧还剩下一个元素（❹），再去处理右侧。就这样顺次地连续返回，对残留的部分进行处理（❺）。

那么在这里，处理完数组某部分的左侧之后，之所以处理还能够转移到右侧，是因为 C 将自动变量分配在栈中。

这个程序将数组分成两部分之后，再进行递归调用。此时，**在栈中分配了新的用于当前部分数组的内存区域**。另外，当某部分数组的数据处理完毕，函数调用返回的时候，栈会收缩，调用前的状态恰好处于栈的最初位置。正因为如此，处理才这样不断地向右移动。

如果这个程序不写成递归，程序写起来会变得有点麻烦。

拒绝偏见，请大家慢慢习惯去运用递归。

ANSI C 以前的 C，为什么不能初始化自动变量的聚合类型？

ANSI C 以前的 C，只有标量才能在声明的同时被初始化（参照 1.1.8 节）。

因此，在如今能这样写的代码：

```
void func(void)
{
    int array[] = {1, 2, 3, 4, 5};
        :
        :
```

曾经只能写成下面这样：

```
void func(void)
{
    static int array[] = {1, 2, 3, 4, 5};
        :
        :
```

就像现在看到的这样，局部变量的内存区域是在函数被调用时，也就是**在执行中**被分配的。在执行中，如果想要初始化数组等聚合类型，编译

器必须在现场生成一些代码，编译器就被复杂化。你应该还记得前面提到过“原本 C 是只能使用标量的语言”吧。

对于这这种情况，如果加上 static 作为静态变量分配内存区域，就可以在程序的**执行前**被完全初始化。

从 ANSI C 开始，聚合类型的自动变量也可以在声明的时候进行初始化。可是，尽管这样还是会花费相应的成本，在可以使用静态数组的情况下，如果添加 static 可以获得令人满意的运行效率。如果同时加上 const，效率可能会更好。

此外，就算是 ANSI C，在聚合类型的初始化运算符中也只能写常量。比如，

```
void func(double angle)
{
    double hoge = sin(angle);
          ⋮
```

这样写是没有问题的，但

```
void func(double angle)
{
    double hoge[] = {sin(angle), cos(angle)};
          ⋮
```

这么写就违反语法规范了。

Rationale 的 3.5.7 中说明了理由。下面的代码似乎会引起理解上的混乱，

```
int x[2] = { f(x[1]), g(x[0]) };
```

由于聚合类型的初始化运算符中只能写常量，所以编译器的处理会变得简单一些。因为只要事先在某个地方分配一个填充初始值的内存区域，然后在进入程序块的时候复制这个区域就可以了。

2.6　利用 malloc()来进行动态内存分配（堆）

2.6.1　malloc()的基础

C 语言中可以使用 malloc()进行动态内存分配。

malloc()根据参数指定的尺寸来分配内存块，它返回指向内存块初始位置的指针，经常被用于动态分配结构体的内存领域、分配执行前还不知道大

小的数组的内存领域等。

```
p = malloc(size);
```

一旦内存分配失败（内存不足），malloc()将返回 NULL。利用 malloc()分配的内存被结束使用的时候，通过 free()来释放内存。

```
free(p);   ←释放 p 指向的内存区域
```

以上是 malloc()的基本使用方式。

像这样能够动态地（运行时）进行内存分配，并且可以通过任意的顺序释放的记忆区域，称为**堆**（heap）*。

* 它不是 C 语言定义的术语。

英语中“heap”这个单词是指像山一样堆得高高的事物（比如干草等）。malloc()就好像从内存的山上分出一部分内存，为我们“从堆中取来所需的内存区域”。

malloc()主要有以下的使用范例：

❶ 动态分配结构体

藏书家可能会用计算机管理自己的藏书。刚从书店买回来一本书，却发现：“啊！原来这本书我已经有了！”特别是漫画书，经常会有重复购买的现象，有吧？（难道只有我是这样？）

因此，最好做一个“藏书管理程序”。

假设用下面这样的结构体管理一本书的数据：

```
typedef struct {
    char  title[64];  /*书名*/
    int   price;      /*价格*/
    char  isbn[32];   /*ISBN*/
          ⋮
          ⋮
    } BookData;
```

对于藏书家来说，他一定要管理堆积如山的 BookData。

在这种情况下，当然可以通过一个巨大的数组来管理大量的 BookData，但是在 C 中必须明确定义数组的长度，究竟需要定义多大的数组是一件让人头疼的事情。如果申请过大的数据空间浪费内存，但如果申请的大小刚刚好，随着书的增加，数组的空间又会变得不足。

通过下面的方式，就可以在运行时分配 BookData 的内存区域。

```
BookData *book_data_p;

/*分配一个结构体 BookData 的内存区域*/
book_data_p = malloc(sizeof(BookData));
```

如果使用**链表**来管理，就可以保持任意个数的 BookData。当然，只要你内存足够。

关于链表的使用方法，在这里只是稍稍提一下，在第 5 章会详细说明。

首先让我们向结构体 BookData 中追加一个指向 BookData 类型的指针成员。

```
typedef struct BookData_tag {
    char  title[64];  /*书名*/
    int   price;      /*价格*/
    char  isbn[32];   /* ISBN */
      :
      :
    struct BookData_tag *next;     ←追加这一行
} BookData;
```

在这个例子中，利用 typedef 为 struct BookData_tag 定义同义词 BookData（不用一次次地写 struct 了）。尽管如此，由于成员 next 声明时 typedef 还没有结束，所以请注意这里必须要写成 struct BookData_tag。tag 不能省略。

通过 next 持有“指向下一个 BookData 的指针”，像图 2-11 这样形成串珠的数据结构，就能够保存大量的 BookData*。

* 以后，图中的●→表示指针，⊠表示 NULL。

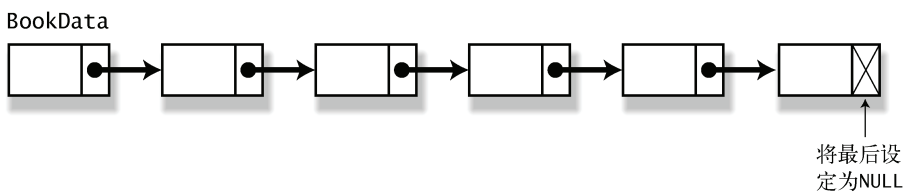

图 2-11　链表

这种被称为“链表”的数据结构运用非常广泛。

❷ 分配可变长数组

在刚才的 BookData 类型中，书名的地方写成下面这样：

```
char  title[64];  /*书名*/
```

可是有的情况下，书名也很变态，比如，

京都露天混浴澡堂蒸气谋杀事件——美女大学生 3 人组所见！
谜一样的旅馆女主人死亡和 10 年前的观光客失踪事件的背后所潜藏的真相是？

（这还是书名吗？）

此时，char title[64]就放不下这么长的书名了。可是，也不是所有的书都有这么长的书名，所以准备太长的数组也是浪费。

这里，将 title 的声明写成：

```
char  *title;  /*书名*/
```

然后，

```
BookData *book_data_p;
     ⋮
/*这里 len 为标题字符数+1（空字符部分的长度） */
book_data_p->title = malloc(sizeof(char) * len);
```

就能够只给标题字符串分配它必要的内存区域了。

此时，如果想要引用 title 中某个特定的字符，当然可以写成

```
book_data_p->title[i]
```

大家想必还记得 p[i]是*(p + i)的语法糖吧。

malloc()的返回值的类型为 void*

ANSI C 以前的 C，因为没有 void*这样的类型，所以 malloc()返回值的类型就被简单地定义成 char*。char*是不能被赋给指向其他类型的指针变量的，因此在使用 malloc()的时候，必须要像下面这样将返回值进行强制转型：

```
book_data_p = (BookData*)malloc(sizeof(BookData));
```

ANSI C 中，malloc()的返回值类型为 void*，void*类型的指针可以不强制转型地赋给所有的指针类型变量。因此，像上面的强制转型现在已经不需要了。

尽管如此，现在还有人时常在这里使用强制转型。我认为，不写多余的强制转型代码，对于顺畅地读懂代码是有利的。

此外，假设忘记了#include stdlib.h，一旦笨拙地对返回值进行强制转型，编译器很可能不会输出警告。

C 语言默认地将没有声明的函数的返回值解释成 int 类型*，那些运气好、目前还能跑起来的程序，如果被迁移到 int 和指针长度不同的处理环境中，就会突然跑不动了。

* 如果可能，应该提高警告的级别，以便让警告可以输出。

因此，我们不要对 malloc()的返回值进行强制转型。因为 C 不是 C++。

另外，C++中可以将任意的指针赋给 void*类型的变量，但不可以将 void*类型的值赋给通常的指针变量。所以在 C++中，malloc()的返回值必须要进行强制转型。但是，如果是 C++，通常使用 new 来进行动态内存分配（也应该这样）。

2.6.2　malloc()是“系统调用”吗

这里有一点离题。

C 的函数库中为我们准备了很多的函数（printf()等）。另外，标准库的一部分函数最终会调用“系统调用”。所谓**系统调用***，就是请求操作系统来帮我们做一些特殊的函数群。标准函数通过 ANSI C 进行了标准化，但是不同操作系统上的系统调用的行为却经常会有差别。

* 这原本是 UNIX 的术语。

比如在 UNIX 操作系统中，printf()最终调用 write()这样的系统调用。不只是 printf()，putchar()和 puts()在最终也是调用 write()。

由于 write()只能输出指定的字节串，为了让应用开发人员更方便地使用和提高可移植性，C 语言给 write()披上了标准库的“皮”。

话说回来，malloc()是系统调用？还是标准库函数？

可能很多人会认为它是系统调用。恰恰相反，malloc()实际上属于标准库函数，它不是系统调用。

> 要　点
> **malloc()不是系统调用。**

2.6.3 malloc()中发生了什么

malloc()大体的实现是，从操作系统一次性地取得比较大的内存，然后将这些内存“零售”给应用程序。

根据操作系统的不同，从操作系统取得内存的手段也是不一样的，在UNIX的情况下使用brk()*的系统调用*。

* break的略称。
* 在最近的实现中，有时也会使用mmap()（后面会提到）。使用malloc()分配小区域的时候，还是使用brk()。

请大家回想一下，在图2-3中，“利用malloc分配的内存区域”的下面，是一片范围很大的空间。系统调用brk()就是通过设定这个内存区域的末尾地址，来伸缩内存空间的函数的。

调用函数的时候，栈会向地址较小的一方伸长。多次调用malloc()时，会调用一次brk()，内存区域会向地址较大的一方伸长。

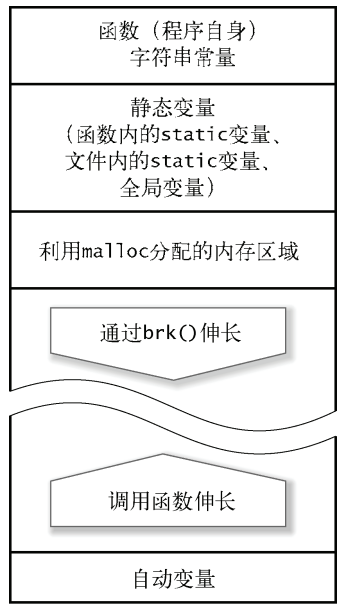

图2-12 内存区域的伸长

“啥？就算可以通过这种方式分配内存区域，但是做不到以任意的顺序释放内存吧？”

也许会有很多人存在以上的想法。**实际上，这是有道理的。**

如果这么说，大家自然会想问：“那么free()又是什么？”那么下面来说说malloc()和free()的基本原理。

现实中的 malloc()的实现，在改善运行效率上下了很多工夫。这里我们只考虑最单纯的实现方式——通过链表实现。

顺便提一下，*K&R* 中也记载了通过链表实现 malloc()的例程。

最朴素的实现就是如图 2-13，在各个块之前加上一个管理区域，通过管理区域构建一个链表。

malloc()遍历链表寻找空的块，如果发现尺寸大小能够满足使用的块，就分割出来将其变成使用中的块，并且向应用程序返回紧邻管理区域的后面区域的地址。free()将管理区域的标记改写成“空块”，顺便也将上下空的块合并成一个块。这样可以防止块的碎片化*。

* 这个操作被称为合并（coalescing）。也有不实现这个操作的，但这样会加剧碎片化（参照 2.6.5 节）。

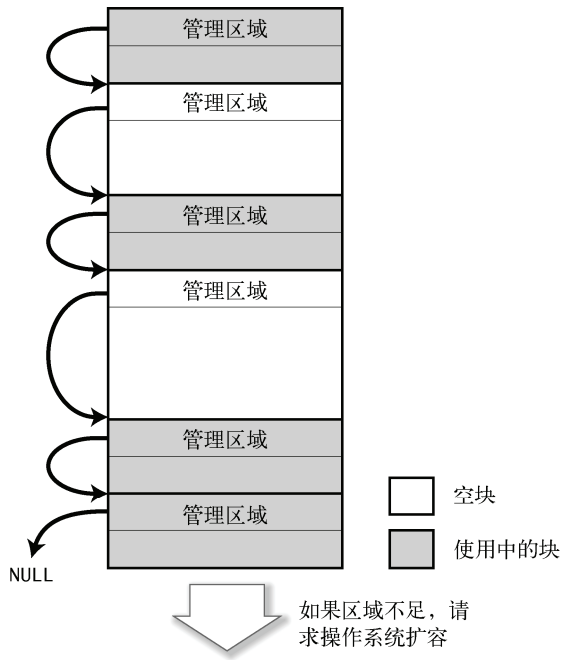

图 2-13　通过链表实现 malloc()的例子

如果不存在足够大的空块，就请求操作系统对空间进行扩容（UNIX 下使用 brk()系统调用）。

那么，在这种内存管理方式的运行环境中，一旦数组越界检查发生错误，越过 malloc()分配的内存区域写入了数据，又会发生什么呢？

此刻将会破坏下一个块的管理区域，所以从此以后的 malloc()和 free()

调用中出现程序崩溃的几率会非常高。这种情况下，不能因为程序是从malloc()中崩溃的就一口咬定“这是库的 bug!!!”这么做只会自取其辱。

现实中的处理环境，是不会这样单纯地实现 malloc()功能的。

比如，作为内存管理方法，除了这里说明的链表方式之外，还有一个被大家广泛熟知的“buddy block system”方法。这种将大的内存逐步对半分开的方式，虽然速度很快，但会造成内存的使用效率低下。

此外，让管理区域和传递给应用程序的区域相邻也是比较危险的，所以有的实现中会将它们分开存放。我的环境（FreeBSD3.2）就是这样的（现行的实现中也有很多是相邻存放的）。

不过，我在这里只是想说明“malloc()绝对不是一个魔法函数”。

随着 CPU 和操作系统的不断进化，也许有一天 malloc()会成为真正的魔法函数。但目前你肯定不能将 malloc()看成魔法函数。对于 malloc()的动作原理，如果不是非常了解，就很有可能陷入程序不能正常进行调试的窘境，或者常常写出非常低效的程序。

所以我建议大家先充分理解 malloc()后再去使用它，不然它只会给你带来危险。

> 要 点
> **malloc()绝对不是魔法函数。**

2.6.4 free()之后，对应的内存区域会怎样

正如刚才所说，malloc()管理从操作系统一次性地被分配的内存，然后零售给应用程序，这是它大致的实现方式。

因此，一般来说调用 free()之后，**对应的内存区域是不会立刻返还给操作系统的**。让我们通过代码清单 2-7 来做个实验。

代码清单 2-7 free.c

```
1:  #include <stdio.h>
2:  #include <stdlib.h>
3:
4:  int main(void)
5:  {
6:      int *int_p;
7:
```

```
 8:      int_p = malloc(sizeof(int));     ←在利用 malloc()分配的
                                              内存区域中……
 9:
10:      *int_p = 12345;     ←写入
11:
12:      free(int_p);     ←调用 free()之后
13:
14:      printf("*int_p..%d\n", *int_p);     ←输出对应的内容!
15:
16:      return 0;
17:  }
```

在我的环境中执行这个程序，会输出 12345。之后随着某次 malloc()调用，恰好将这片区域重新进行分配后，才会发生这部分内容的改写。

但是，C 标准不能保证情况总是这样。因此，调用 free()之后，**是不能引用对应的内存区域的**。

这里之所以特地举出这个例子，是因为“调用 free()之后，对应的内存内容不会被马上破坏”。这样的特性给程序调试中的原因查明带来了困难。

如图 2-14，某内存区域被两个指针同时引用。使用指针 A 引用这个区域的程序员认为当前区域对他来说已经不需要了，于是稀里糊涂地调用了 free()。实际上，在远离当前这段代码的地方，指针 B 还在引用当前这片区域。此时，会发生什么呢*?

* 大型的程序常常会出现这种问题。

仓促地调用 free()是有问题的，就算调用了 free()，指针 B 引用的内存区域也不会立刻被破坏，暂时还保持着以前的值。直到在某个地方执行 malloc()，随着当前内存区域被重新分配，内容才开始被破坏。这样的 bug，从原因产生到 bug 被发现之间周期比较长，因此给程序调式带来很大困难。

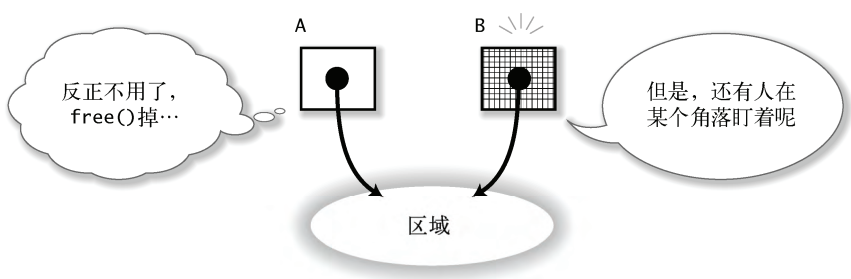

图 2-14　一个内存区域被两个指针引用……

为了避免这个问题，如果是大型程序，可以做一个函数给 free()披一张皮，并且使程序员们只能调用这个函数，在区域被释放之前故意将区域破坏

（可以胡乱地填充一个像 0xCC 这样的值）。可惜的是，我们无法知道当前指针指向的区域的大小*。如果偏要这么做，可以考虑也给 malloc()批上一张皮，每次分配内存的时候可以多留出一点空间，然后在最前面的部分设定区域的大小信息。

* 如果使用 malloc()分配内存，标准库肯定是知道此内存的大小的。遗憾的是，ANSI C 没有提供公开内存大小的函数。

当然，这种手法只能用于程序的调试版本。在去掉调式选项对程序进行编译的时候，这些代码就会消失。这样就不会影响发行版的程序的运行效率。

这么做虽然有点麻烦，但是对于大规模的程序来说，这种手法还是非常有效的。

2.6.5 碎片化

某些处理环境对 malloc()的实现，和 2.6.3 节中描述的没有大的差别，但是以随机的顺序分配、释放内存。此时，又会发生什么问题呢？

此时，内存被零零碎碎分割，会出现很多细碎的空块。并且，这些区域**事实上是无法被利用的**。

这种现象，我们称为碎片化（fragmentation）（参照图 2-15）。

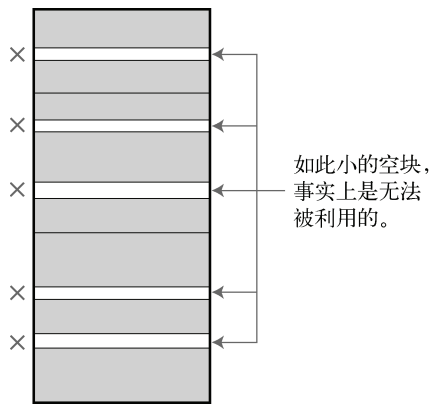

图 2-15 碎片化

将块向前方移动，缩小块之间的距离，倒是可以整合零碎的区域并将它们组合成较大的块*。可是 C 语言将虚拟地址直接交给了应用程序，库的一方是不能随意移动内存区域的。

* 这样的操作称为压缩（compaction）。

在 C 中，只要使用 malloc()的内存管理过程，就无法根本回避碎片化问题。但若利用 realloc()函数（在下一节说明），倒可以让问题得到一些改善。

2.6.6　malloc()以外的动态内存分配函数

本书介绍 malloc()以外的动态内存分配函数，首先介绍 calloc()函数。

```
#include <stdlib.h>

void *calloc(size_t nmemb, size_t size)
```

calloc()使用和 malloc()相同的方式分配 nmemb × size 大小的内存区域，并且将该区域清零返回。也就是说，calloc()和下面的代码具有同等效果。

```
p = malloc(nmemb * size);
memset(p, 0, nmemb * size);
```

老实说，我对 C 语言提供这个函数的意图**完全无法理解**。通过两个参数传递区域的尺寸，实际上只是在内部做乘法运算，这点就让人费解。将区域清零返回这一点也让人感觉别扭。因为就算通过 memset()清零，在浮点数和指针的情况下它们的值也不一定为 0（也许是空指针）*。

* 可是，在现行的大部分处理环境中，浮点数的值被初始化为浮点数 0，指针被初始化为空指针。这无疑是多此一举地将问题复杂化。

calloc()对于避免难以再现的 bug 是有效的。我一般是给 malloc()披一张皮，但不是清零，而是填充一个无意义的值 0xCC。对于在那些没有好好做初始化的程序中发现 bug，我的方式肯定更好一些。

此外，在 *K&R* 的第一版中有一个 cfree()函数*，看上去好像可以利用这个函数释放由 calloc()分配的内存。其实这个函数和 free()做的事完全一样。现在，为了维持向后的兼容性，在大部分的环境中仍然可以使用 cfree()。但是 **ANSI C 的标准中是不存在这个函数的**。就算你使用了 calloc()，在释放内存的时候也请使用 free()。

* 随便提一下，实际上在 *K&R* 的第 1 版中，只记载了 calloc()和 cfree()的配对函数，并**没有记载** malloc()。

> *要　点*
> **不要使用** cfree()。

还有一个内存分配函数是 realloc()。

此函数用于改变已经通过 malloc()分配的内存的尺寸。

```
#include <stdlib.h>

void *realloc(void *ptr, size_t size);
```

realloc()将 ptr 指定的区域的尺寸修改成 size，并且返回指向新的内存区域的指针。

话虽这么说，但正如前面说明的那样，malloc()不是一个魔法函数，

realloc()同样如此。通常，我们使用 realloc()来扩展内存区域。如果通过 ptr 传递的区域后面有足够大小的空闲空间，就直接实施内存区域扩展。但是，如果后面的区域没有足够多的空闲空间，就分配其他新的空间，然后将内容复制过去。

我们经常需要对数组顺次追加元素，这种情况下，如果每追加一个元素都利用 realloc()进行内存区域扩展，将会发生什么呢?

如果手气不错，后面正好有足够大的空地儿，这样还好。如果不是这样，就需要频繁地复制区域中的内容，这样自然会影响运行效率。另外，不断地对内存进行分配、释放的操作，也会引起内存碎片化。此时不妨考虑一下这种手法：假设以 100 个元素为单位，一旦发现空间不足，就一次性进行内存扩展分配*。

* Java的 java.util.Vector 类就是采用了这个手法。但是本质上，Java 并不背离堆的机制。

此外，一旦利用 realloc()扩展巨大的内存区域，除了在复制上花费很多时间之外，也会造成堆中大量的空间过分地活跃。如果想要动态地为大量的元素分配内存空间，最好不要使用连续的内存区域，而是应该积极使用链表。

要 点

请谨慎使用 realloc()。

另外，如果通过 ptr 向 realloc()传入 NULL，realloc()的行为就和 malloc()完全相同。偶尔会见到像下面这样的代码，

```
if (p == NULL) {
    p = malloc(size);
} else {
    p = realloc(p, size);
}
```

完全可以简洁地写成下面这样，

```
p = realloc(p, size);
```

（姑且先把返回 NULL 的问题放在一边*）。

* 对于这种写法，realloc()返回 NULL 的时候，p 会永远被丢失，这是一个问题。

随便说一下，如果通过 size 向 realloc()传入 0，realloc()的行为就和 free()完全相同。

补充 malloc()的返回值检查

内存分配一旦失败，malloc()会返回 NULL。

大部分 C 的书籍都歇斯底里地提醒大家“如果你调用了 malloc()，就**必须**做返回值检查”，关于这一点，本书倒是要对此唱唱反调。不是吗？那么做**太麻烦了**。

完美地对应内存不足的问题，

```
p = malloc(size);
if (p == NULL) {
    return OUT_OF_MEMORY_ERROR;
}
```

并不是像上面这样机械地写一段代码那么单纯的。

当然我们在构造某些数据结构的时候，既需要确保数据结构自身没有任何矛盾，还要确保函数能够返回，随之而来的测试也会变得更加复杂。假设能够做到这些，却又因为分配已达上限字节数的内存区域而失败，此时我们又该怎么办呢？

- 通过对话框通知用户“内存不足”？此时，也许程序自身已经没有能力弹出对话框了……
- 为了不丢失写到一半的文档数据，姑且将文件保持打开状态……能做到吗？
- 安全起见，通过递归的方式遍历深层次的树结构……可以分配栈的空间吗？
- 还是先保护硬盘数据吧……对于 Windows 这样存放了交换文件的文件系统，如果此时恰恰只有一个分区，硬盘数据又会出现什么状况呢？

其实并不是只有显式的 malloc()调用才会导致内存不足，深度的递归调用也会引起栈内存不足。此外，根据不同情况，printf()内部有时也会调用 malloc()。不光是 malloc()调用，操作系统往内存中写入数据的时候也会进行内存区域分配，对于这种情况，调用 malloc()时是无法发现内存不足的。

开发通用性极高的库程序的时候，“切实地进行返回值检查”的确是必须的。但是如果是开发普通的应用程序，只需要给 malloc()披上一张皮，一旦发生内存不足，当场输出错误信息并且终止该程序，这种做法在很多时候也是可行的。主张

> “一旦调用 malloc()，绝对要对返回值进行检查，并且进行完善的处理。”

的人，一般在使用 Java 时，一定会在合适的层次上使用 catch，当然他们也不会去使用 Perl 等等的 Shell 脚本……

程序结束时必须调用 free()吗?

网上的新闻组 fj.comp.lang.c 曾经针对下面的主题发生过一次激烈的口水战。

程序结束之前，一定需要释放 malloc()分配的内存吗?

这是一个让人头疼的问题。在现实中，如果是普通 PC 上使用的操作系统，在进程结束时，肯定会释放曾经分配给当前进程的内存空间。

其实 C 语言标准却并没有规定必须要这么做，只是正经的操作系统都主动提供这个功能。此外，在写推荐内存为 128MB 的程序时，你不会去考虑以后还要将它移植到电视机遥控器的嵌入式芯片上吧。也就是说，在程序结束之前，没有必要调用 free()。

可是，对于进行“读取一个文件→处理→输出结果→结束”这样的处理，如果要扩展成可以连续处理多个文件，一旦原来的程序没有调用 free()，后面的人那可真的遭罪了。

为了提前发现内存溢出（忘了调用 free()）的漏洞，最近出现了一些工具，它可以报告那些没有在结束时被实施 free()的内存区域的列表。此时，“故意不调用 free()的区域”和“忘记调用 free()区域”被混同在一起出现在报告中，让人难以区分。对于不能使用这些工具的开发环境，可以采用“将 malloc()和 free()披上一张皮，然后计算它们被调用的次数，并且确认程序结束时次数是否一致”这样简单可行的方法，并且这种方法对于检查内存泄漏非常有效。

从这一点上来看，我认为“对于调用 malloc()分配的内存区域，在程序结束前一定要调用 free()”这样的原则也是相当合理的。

那么到底应该怎么做？答案是“具体问题具体分析”（我倒！跟没说一样）。

我倒是不太喜欢“必定 free()派”的观点，其实之所以“内定 free()”，是因为他们认为：

- 使用 malloc()之后写上对应的 free()是一种**谨慎的编程风格**
- 程序员就应该**留意**将 malloc()和 free()对应起来
- “调用了 exit()，就没有必要调用 free()了”这种想法是**不负责任并且恶劣的**编程风格

不管怎么说程序员也是人（瞧这话说的），对于人来说，恐怕会犯错的地方必定犯错。明明如此，你还去标榜什么“写程序要谨慎”，我觉得有点自讨没趣。

“谨慎地”编程有那么了不起吗？我认为那些能尽力让自己摆脱“麻烦事”的程序员才是优秀的。该脱手时就脱手，尽可能依赖工具去完成检查工作而不是总去目测。就算在那些无论如何也要依靠手动去应对的情况下，也暗自发誓“总有一天我把它做成自动化”。此类程序员才是人才！

2.7　内存布局对齐

稍微转换一下话题……

假设有下面这样的一个结构体：

```
typedef struct {
    int     int1;
    double  double1;
    char    char1;
    double  double2;
} Hoge;
```

在我的环境中，sizeof(int)的结果为 4，sizeof(double)的结果为 8，随便说一下，根据 C 标准，sizeof(char)的结果必定为 1*。敢问阁下，这个结构体的尺寸是多大？

* 比如，即使 char 为 9 bit 的处理环境（如果真的存在），sizeof(char)值也是 1。标准就是这么定义的。

4 + 8 + 1 + 8 = 21 个字节——几乎在所有的情况下，这个答案都是错误的。在我的处理环境中，答案是 24 个字节。

还是通过程序做个实验吧（参照代码清单 2-8）。

声明一个 Hoge 类型的变量，然后将各成员的地址输出*。

* 如果需要获得结构体成员距离初始位置的偏移量，一般使用 stddef.h 中定义的宏 offsetof()。使用这个宏，不需要声明哑变量（dummy）也可以获取偏移量。

代码清单 2-8　alignment.c

```
 1:  #include <stdio.h>
 2:
 3:  typedef struct {
 4:      int     int1;
 5:      double  double1;
 6:      char    char1;
 7:      double  double2;
 8:  } Hoge;
 9:
10:  int main(void)
11:  {
12:      Hoge        hoge;
13:
14:      printf("hoge size..%d\n", sizeof(Hoge));
```

```
15:
16:      printf("hoge   ..%p\n", &hoge);
17:      printf("int1   ..%p\n", &hoge.int1);
18:      printf("double1..%p\n", &hoge.double1);
19:      printf("char1  ..%p\n", &hoge.char1);
20:      printf("double2..%p\n", &hoge.double2);
21:
22:      return 0;
23:  }
```

我的环境中的运行结果如下：

```
hoge size..24
hoge   ..0xbfbfd9d0
int1   ..0xbfbfd9d0
double1..0xbfbfd9d4
char1  ..0xbfbfd9dc
double2..0xbfbfd9e0
```

观察运行结果可以发现，char1 的后面空出来一块。

这是因为根据硬件（CPU）的特征，对于不同数据类型的可配置地址受到一定限制。或者，即使可以配置，某些 CPU 的效率也会降低。此时，编译器会适当地进行边界调整（布局对齐），在结构体内插入合适的填充物。

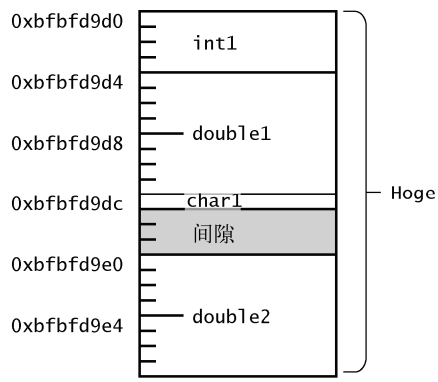

图 2-16 布局对齐

根据这个实验，在我的环境中，int 和 double 被配置在 4 的倍数的地址上。

布局对齐处理有时候也在结构体的末尾进行，这是由于有时候需要构造结构体数组的缘故。针对这样的结构使用 sizeof 运算符，会返回包含末尾对

齐字节的结构体长度。将结果和元素个数相乘，就可以获得整个数组的大小。

此外，malloc()会充分考虑到各种类型的长度，返回调整后最优化的地址。局部变量等也会被配置到优化调整后的地址上。

布局对齐操作是根据 CPU 的情况进行的。因此，**根据 CPU 的不同，布局对齐填充的方式也不同**。在我的环境中，double 可以被配置在 4 的倍数的地址上，但在很多 CPU 上，double 只能被配置在 8 的倍数的地址上。

偶尔，也会有人比较讨厌布局对齐方式对硬件的依赖，通过手工调整边界来提高可移值性。

```
typedef struct {
    int     int1;
    char    pad1[4];    ←— 通过手工填充
    double  double1;
    char    char1;
    char    pad2[7];    ←— 这里也是
    double  double2;
} Hoge;
```

可是，这么做究竟有什么作用呢?

即使不这么做,编译器也会根据 CPU 的情况帮我们进行适当的边界调整。如果只是引用成员名，就根本没有必要去理会布局对齐方式。

如果需要将结构体照原样（通过 fwrite()）输出到文件中，由于 CPU 的不同，在其他机器上想要读取这个结构体的时候，对齐方式的不同可能导致问题。那么，通过手工方式调整边界，说不定某台机器上输出的数据，也能被其他机器读取。可是无论怎样，这只不过是**偶尔**才可以拿出来说的例子。

在上面的例子中，pad1 的尺寸为 4，pad2 的尺寸为 7。**究竟这些数字是怎么冒出来呢**？连标准也不能保证 sizeof(int)为 4，sizeof(double)为 8。将这些数字直接写在程序中，还说什么“为了提高可移植性”……

也就是说，手工插入填充物的方法，即使可以让不同机器的数据交换成为可能，也只不过是敷衍逃避。原型开发也许会允许使用这种方式，但是如果在现实中进行数据交换，**将结构体按照原样写入到文件的方式本身就是个错误**。

另外，即使是 sizeof(int)为 4 的处理环境，其内部表现也不一定相同。关于这一点，下一节会进行说明。

要 点
即使手工进行布局对齐，也不能提高可移植性。

2.8 字节排序

我的环境是在普通 PC（CPU 为 Celeron）上安装了 FreeBSD3.2。sizeof(int)为 4，但在这 4 个字节中，整数究竟是以什么样的形式存放的呢？

这里依然使用一个程序来验证问题（代码清单 2-9）。

代码清单 2-9 byteorder.c

```
 1:  #include <stdio.h>
 2:
 3:  int main(void)
 4:  {
 5:      int         hoge = 0x12345678;
 6:      unsigned char *hoge_p = (unsigned char*)&hoge;
 7:
 8:      printf("%x\n", hoge_p[0]);
 9:      printf("%x\n", hoge_p[1]);
10:      printf("%x\n", hoge_p[2]);
11:      printf("%x\n", hoge_p[3]);
12:
13:      return 0;
14:  }
```

程序中将 int 型变量强制赋给 unsigned char *型变量 hoge_p，因此，我们可以使用 hoge_p[0]~hoge_p[3] 以字节为单位引用 hoge 的内容。

我的环境中，程序的执行结果如下：

```
78
56
34
12
```

对于我的环境，“0x12345678”在内存中好像是逆向存放的呢。

可能有读者会感到意外。其实 Intel 的 CPU（包括 AMD 等兼容 CPU），

都是像这样将整数**颠倒过来**存放的。这种配置方式一般称为小端（little-endian）字节序。

此外，对于工作站等的 CPU，经常将“0x12345678”这样的值以“12，34，56，78”的顺序存放，这种配置方式称为大端（big-endian）字节序。

那么，小端和大端这样的字节排列方式就称为字节排序（Byte Order）。

小端和大端哪一种方式会更好？这个问题如同辩论宗教取向一样令人纠结，在这里我们就不做深入讨论了。事实是它们各有所长。在纸和笔的时代，人类在做加法运算时都是从低位开始的，可是对于 CPU 来说采用小端方式会更轻松。当然，对于人类来说，大端的方式也许更容易理解。

问题就在于，连以上这样整数类型的数据**在内存中的配置方式都因 CPU 的不同而不同**。

事实上，有的 CPU 还会采用“将两个字节编成一组再反向排列”这样差异性更大的字节排列方式。此外，很多环境使用 IEEE 754 规定的形式处理浮点类型，但 C 的标准并没有规定这一点*。即使是采用 IEEE 754 的处理环境，Intel 的 CPU 也还是逆向排列字节的。

* Java 也规定了这一点。因此，如果硬件不支持 IEEE 754，情况就会变得复杂一点。

也就是说，根据环境的不同，内存中的二进制映像的形式也不尽相同，所以那些**试图将内存的内容直接输出到硬盘，或者通过网络进行传输以便不同的机器读取等想法都是不可取的**。

如果要考虑数据兼容性，建议自定义一些数据格式，然后遵循这些格式来输出数据。UNIX 的 XDR 等工具可以在这一点上为我们提供帮助。

> 要　点
> **无论是整数还是浮点小数，内存上的表现形式都随环境的不同而不同。**

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

直到这里，都是在我的环境中实际地运行程序，然后结合输出结果进行各种各样的说明。

但是，C 标准并不是根据实现方式制定的，而是根据对开发语言的需求来制定的。

比如，如今大多的PC和工作站的操作系统都为我们实现了虚拟地址功能，但是对于那些不具备此功能的操作系统，C 语言编写的应用程序同样能跑得很欢。

此外，前面介绍了“C 语言将自动变量分配在栈中”。其实，这一点并没有在标准中做出规定。因此，即使在每次进入函数的时候将自动变量配置到堆中，也不算违反标准。只是这种实现非常慢，没有人会愚蠢到这种地步罢了。

至于 malloc()的实现，在不同的处理环境中也存在很大的差异。除了brk()这样 UNIX 特有的系统调用之外，最近也出现了这样的分配方式：在分配较大的内存空间时，使用系统调用 mmap()，之后通过 free()将内存返回给操作系统*。

* 默认地不一定被关联到那里。

进一步说，第 1 章的内容都是以“指针就是地址”为前提进行说明的。可是关于这一点，**在标准中连一个字也没有提到**。正如前面摘录的内容一样，标准中只提到“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”。也就是说，如果连实体都能够被引用，即使你不使用虚拟地址也不违反标准。

在标准中，&运算符被称为“地址运算符”，这就让人有点糊涂，具体的说明是：

> 一元&（地址）运算符的结果是指向由其操作数所表示对象或函数的指针。

无论怎样，此运算符返回的是“指针”，而不是“地址”。

通过%p，printf()的输出结果被定义成：

> 该指针的值将以实现定义的方式转换为一系列可打印的字符。

由于 C 语言经常通过指针的形式让我们可以直接接触到地址，所以人们常常误以为：

- 如果没有养成经常关注内容状态的习惯，是不适合进行 C 语言编程的
- C 语言其实就是结构化的汇编
- C 语言其实是低级语言

但对于开发应用程序的普通程序员，其实**完全没有必要去理会指针就是地址**这件事。

可能 C 语言确实就是低级语言。也许有人想说：

> 明明想使用高级语言，只是在不得已用了 C 语言的情况下，还装腔作势刻意地来一句“啥？C 难道不是低级语言吗”，并以此来标榜自己能够使用低级语言编程，这样可以让他人高看自己一眼。

其实不是这样的。你明知处理环境中指针就是地址，却对此**选择性失明**，并且坚持抱着“如果将 C 当成高级语言，C 就可以像高级语言那样使用”的想法。那么，等待你的将是成群的 bug。

本章也是在强调“指针就是地址”的基础之上，展开各部分内容的。

比起唠唠叨叨地进行那些抽象的说明，具体地将地址表现出来的方式是不是更直观？将自动变量的地址表示出来，你一下子就能看到栈是如何随着函数调用的发生而不断增长的。如果你没有理解这一点，你同样也无法理解递归调用的原理。

此外，在大部分的运行环境中，C 语言不做运行时检查，这已经是无法回避的现实。因此，如果没有在某种程度上掌握内存的使用方式，正常的调试工作就会遇到麻烦。

另外，遇到不明白的地方，应该通过实验来确认。俗话说得好，实践出真知！

对于本章的例程，请大家一定亲手在自己的环境中尝试运行一下。

但是，一旦大家通过实验接受了“指针就是地址”这个观点，就应该对此提高警惕，否则你就会顺手写出一些抽象度和移植度低下的程序。

此外，一旦出现 bug，请带着“指针就是地址”的观点去解决它——这种姿态在解决 bug 上是恰到好处的。

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

查看更多图书信息请关注新浪微博@图灵教育

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

ISBN 978-7-115-30121-5
定价：49.00元