Broadview®
www.broadview.com.cn

畅销书升级版

把手教你将四年的任务用四个月做好
C语言为载体，系统讲述计算机原理和程序原理
零基础开始学习编程，内容涵盖C语言入门及C语言本质

一站式学习C编程

北京亚嵌教育研究中心 组编
宋劲杉 编著

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

一站式学习C编程

上篇 C语言入门

- 程序的基本概念
- 简单函数
- 深入理解函数
- 结构体
- 编码风格
- 排序与查找
- 常量、变量和表达式
- 分支语句
- 循环语句
- 数组
- gdb
- 栈与队列

下篇 C语言本质

- 计算机中数的表示
- 运算符详解
- x86汇编程序基础
- 链接详解
- Makefile基础
- 函数接口
- 链表、二叉树和哈希表
- 数据类型详解
- 计算机体系结构基础
- 汇编与C之间的关系
- 预处理
- 指针
- C标准库

上架建议：程序设计 > C语言

ISBN 978-7-121-12982-7

9 787121 129827 >

定价：59.00元

责任编辑：李　冰
封面设计：侯士卿

一站式学习C编程

北京亚嵌教育研究中心 组编
宋劲杉 编著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

本书有两条线索，一条线索是以Linux平台为载体全面深入地介绍C语言的语法和程序的工作原理，另一条线索是介绍程序设计的基本思想和开发调试方法。本书分为两部分：第一部分讲解编程语言和程序设计的基本思想方法，让读者从概念上认识C语言；第二部分结合操作系统和体系结构的知识讲解程序的工作原理，让读者从本质上认识C语言。

本书适合做零基础的初学者学习C语言的第一本教材，帮助读者打下牢固的基础。有一定的编程经验但知识体系不够完整的读者也可以对照本书查缺补漏，从而更深入地理解程序的工作原理。本书最初是为北京亚嵌教育研究中心的嵌入式Linux系统工程师就业班课程量身定做的教材之一，也适合作为高等院校程序设计基础课程的教材。本书对于C语言的语法介绍得非常全面，对C99标准做了很多解读，因此也可以作为一本精简的C语言语法参考书。

图书在版编目（CIP）数据

一站式学习 C 编程 / 宋劲杉编著；北京亚嵌教育研究中心组编. —北京：电子工业出版社，2011.3
ISBN 978-7-121-12982-7

Ⅰ. ①一… Ⅱ. ①宋… ②北… Ⅲ. ①C语言—程序设计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字（2011）第027542号

责任编辑：李 冰
文字编辑：江 立
印 刷：北京天宇星印刷厂
装 订：三河市皇庄路通装订厂
出版发行：电子工业出版社
北京市海淀区万寿路173信箱 邮编：100036
开 本：787×1092 1/16 印张：30.75 字数：690千字
印 次：2011年3月第1次印刷
印 数：4000册 定价：59.00元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。

服务热线：（010）88258888。

前　言

本书最初是为北京亚嵌教育研究中心的嵌入式 Linux 系统工程师就业班课程量身定做的教材之一。该课程是为期四个月的全日制职业培训，要求学员毕业时具备非常 Solid 的 C 编程能力，能熟练地使用 Linux 系统，同时对计算机体系结构与指令集、操作系统原理和设备驱动程序都有较深入的了解。然而学员入学时的水平是非常初级而且参差不齐的：学历有专科、本科也有研究生，专业有和计算机相关的也有很不相关的（例如会计专业），以前从事的职业有和技术相关的也有完全不相关的（例如 HR），年龄从二十出头到三十五六岁的都有。这么多背景完全不同、基础完全不同、思维习惯和理解能力完全不同的人来听同一堂课，大家都迫切希望学会嵌入式开发技术，投身 IT 行业，这就是职业教育的特点，也是我写作本书时需要考虑的主要问题。

学习编程绝不是一件简单的事，尤其是对于零基础的初学者来说。大学的计算机专业有四年时间从零基础开始培养一个人，微积分、线性代数、随机数学、离散数学、组合数学、自动机、编译原理、操作系统、计算机组成原理等一堆基础课，再加上 C/C++、Java、数据库、网络、软件工程、计算机图形学等一堆专业课，最后培养出一个能找到工作的学生。很遗憾这最后一条很多学校没有做好，来亚嵌培训的很多学生这四年就是这么学过来的，但据我们考查他们的基础几乎为零，我不知道为什么。与之形成鲜明对比的是，只给我们四个月的时间，同样要求从零基础开始，最后培养出一个能找到工作的学生，而且还要保证他找到工作，这就是职业教育的特点。

为什么我说“只给我们四个月的时间”？我们倒是想教四年呢，但学时的长短我们做不了主，是由市场规律决定的。四年的任务要求四个月做好，要怎么完成这样一个几乎不可能的任务呢？有些职业教育给出的答案是“实用主义”，打出了“有用就学，没有用就不学”的口号，大肆贬低说大学里教的基础课都是过时的、无用的，只有他们教的技术才是实用的，这种炒作很不好，我认为大学里教的每一门课都是非常有用的，基础知识在任何时候都不会过时，倒是那些时髦的“实用技术”有可能很快就过时了。

四年的任务怎么才能用四个月做好？我们给出的答案是“优化”。现在大学里安排的课程体系最大的缺点就是根本不考虑优化。每个过来人都会有这样的感觉：大一大二学了好多数学课，却不知道都是干什么用的，也不懂为什么要学。连它有什么用都不知道怎么能有兴趣学好呢？然后到大三大四学专业课时，用到以前

目　录

上篇　C 语言入门

下篇 C 语言本质

上篇 C 语言入门

第 1 章

程序的基本概念

1.1 程序和编程语言

程序（Program）告诉计算机应如何完成一个计算任务，这里的计算可以是数学运算，比如解方程，也可以是符号运算，比如查找和替换文档中的某个单词。从根本上说，计算机是由数字电路组成的运算机器，只能对数字做运算，程序之所以能做符号运算，是因为符号在计算机内部也是用数字表示的。此外，程序还可以处理声音和图像，声音和图像在计算机内部必然也是用数字表示的，这些数字经过专门的硬件设备转换成人可以听到的声音和看到的图像。

程序由一系列指令（Instruction）组成，指令是指示计算机做某种运算的命令，通常包括以下几类：

输入（Input）

从键盘、文件或者其他设备获取数据。

输出（Output）

把数据显示到屏幕，或者存入一个文件，或者发送到其他设备。

基本运算

执行最基本的数学运算（加减乘除）和数据存取。

测试和分支

测试某个条件，然后根据不同的测试结果执行不同的后续指令。

循环

重复执行一系列操作。

对于程序来说，有上面这几类指令就足够了。你曾用过的任何一个程序，不管它有多么复杂，都是由这几类指令组成的。程序是那么的复杂，而编写程序可以用的指令却只有这么简单的几种，这中间巨大的落差就要由程序员去填补了，所以编写程序理应是一件相当复杂的工作。**编写程序可以说就是这样一个过程：把复杂的任务分解成子任务，把子任务再分解成更简单的任务，层层分解，直到最后简单得可以用以上指令来完成。**

编程语言（Programming Language）分为低级语言（Low-level Language）和高级语言（High-level Language）。机器语言（Machine Language）和汇编语言（Assembly Language）属于低级语言，直接用计算机指令编写程序。而 C、C++、Java、Python 等属于高级语言，用语句（Statement）编写程序，语句是计算机指令的抽象表示。举个例子，同样一个语句用 C 语言、汇编语言和机器语言分别表示如表 1.1 所示。

表 1.1　一个语句的三种表示

编程语言	表示形式
C 语言	a=b+1;
汇编语言	mov　　0x804a01c,%eax add　　$0x1,%eax mov　　%eax,0x804a018
机器语言	a1 1c a0 04 08 83 c0 01 a3 18 a0 04 08

计算机只能对数字做运算，符号、声音、图像在计算机内部都要用数字表示，指令也不例外，表 1.1 中的机器语言完全由十六进制数字组成。最早的程序员都是直接用机器语言编程，但是很麻烦，需要查大量的表格来确定每个数字表示什么意思，编写出来的程序很不直观，而且容易出错，于是有了汇编语言，把机器语言中一组一组的数字用助记符（Mnemonic）表示，直接用这些助记符写出汇编程序，然后让汇编器（Assembler）去查表把助记符替换成数字，也就把汇编语言翻译成了机器语言。从上面的例子可以看出，汇编语言和机器语言的指令是一一对应的，汇编语言有三条指令，机器语言也有三条指令，汇编器就是做一个简单的替换工作，例如在第一条指令中，把 movl ?,%eax 这种格式的指令替换成机器码 a1 ?，?表示一个地址，在汇编指令中是 0x804a01c，转换成机器码之后是 1c a0 04 08（这是指令中十六进制数的小端表示，小端表示将在第 16.2 节介绍）。

从上面的例子还可以看出，C 语言的语句和低级语言的指令之间不是简单的一一对应关系，一条 a=b+1;语句要翻译成三条汇编或机器指令，这个过程称为编译（Compile），由编译器（Compiler）来完成，显然编译器的功能比汇编器要复杂得多。用 C 语言编写的程序必须经过编译转成机器指令才能被计算机执行，编译需要花一些时间，这是用高级语言编程的一个缺点，然而更多的是优点。首先，用 C 语言编程更容易，写出来的代码更紧凑，可读性更强，出了错也更容易改正。其次，C 语言是可移植的（Portable）或者称为平台无关的（Platform Independent）。

“平台”这个词有很多种解释，可以指计算机体系结构（Architecture），也可以指操作系统，也可以指开发平台（编译器、链接器等）。不同的计算机体系结构有不同的指令集（Instruction Set），可以识别的机器指令格式是不同的，直接用某种体系结构的汇编或机器指令写出来的程序只能在这种体系结构的计算机上运行，然而各种体系结构的计算机都有各自的 C 编译器，可以把 C 程序编译成各种不同体系结构的机器指令，这意味着用 C 语言写的程序只需稍加修改甚至不用修

改就可以在各种不同的计算机上编译运行。各种高级语言都具有C语言的这些优点，所以绝大部分程序是用高级语言编写的，只有和硬件关系密切的少数程序（例如驱动程序）才会用到低级语言。还要注意一点，即使在相同的体系结构和操作系统下，用不同的C编译器（或者同一个C编译器的不同版本）编译同一个程序得到的结果也有可能不同，C语言有些语法特性在C标准中并没有明确规定，各编译器有不同的实现，编译出来的指令的行为特性也会有所不同，应该尽量避免使用不可移植的语法特性。

总结一下编译执行的过程，首先你用文本编辑器写一个C程序，然后保存成一个文件，例如program.c（通常C程序的文件名后缀是.c），这称为源代码（Source Code）或源文件，然后运行编译器对它进行编译，编译的过程并不执行程序，而是把源代码全部翻译成机器指令，再加上一些描述信息，生成一个新的文件，例如a.out，这称为可执行文件，可执行文件可以被操作系统加载运行，计算机执行该文件中由编译器生成的指令，如图1.1所示。

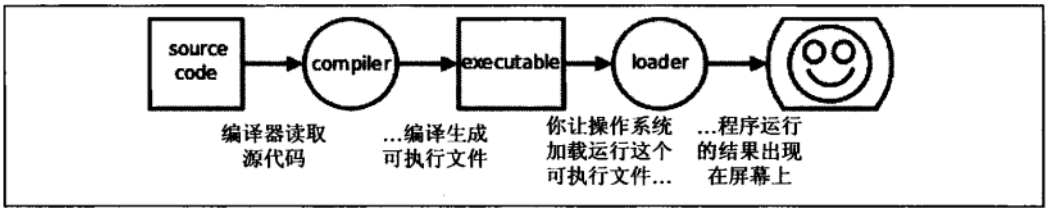


图1.1 编译执行的过程

有些高级语言以解释（Interpret）的方式执行，解释执行过程和C语言的编译执行过程很不一样。例如编写一个Shell脚本script.sh，内容如下：

```
#! /bin/sh
VAR=1
VAR=$(($VAR+1))
echo $VAR
```

定义Shell变量VAR的初始值是1，然后自增1，然后打印VAR的值。用Shell程序/bin/sh解释执行这个脚本，结果如下：

```
$ /bin/sh script.sh
2
```

这里的/bin/sh称为解释器（Interpreter），它把脚本中的每一行当做一条命令解释执行，而不需要先生成包含机器指令的可执行文件再执行。如果把脚本中的这三行当做三条命令直接敲到Shell提示符下，也能得到同样的结果：

```
$ VAR=1
$ VAR=$(($VAR+1))
$ echo $VAR
2
```

解释执行的过程如图1.2所示。

还有很多编程语言采用编译和解释相结合的方式执行，这种方式相当流行，Java、Python、Perl 等编程语言都采用这种方式。以 Python 为例，程序员写的源代码.py 文件首先被编译成.pyc 文件，称为字节码（Byte Code），然后字节码被 Python 虚拟机解释执行。字节码是 Python 虚拟机的指令而非机器指令，所以它是平台无关的，如果把字节码文件从一种平台拷贝到另一种平台上，只要另一种平台也安装了 Python 虚拟机，就能运行这个字节码文件。虚拟机执行过程如图 1.3 所示。

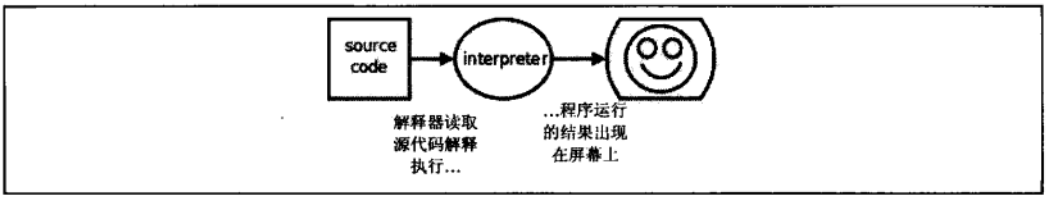


图 1.2　解释执行的过程

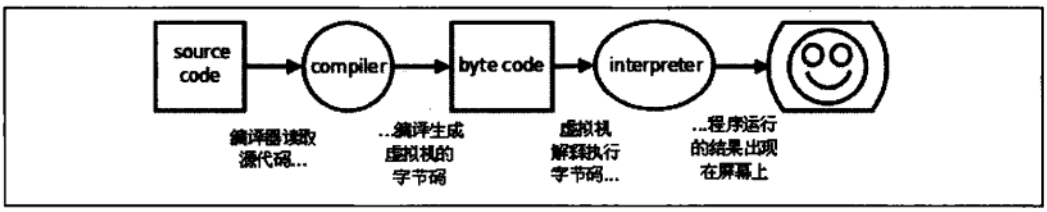


图 1.3　虚拟机执行过程

编程语言仍在发展演化。以上介绍的机器语言称为第一代语言（1GL，1st Generation Programming Language），汇编语言称为第二代语言（2GL，2nd Generation Programming Language），C、C++、Java、Python 等可以称为第三代语言（3GL，3rd Generation Programming Language）。目前已经有了 4GL（4th Generation Programming Language）和 5GL（5th Generation Programming Language）的概念。3GL 的编程语言虽然是用语句编程而不直接用指令编程，但语句也分为输入、输出、基本运算、测试分支和循环等几种，和指令有直接的对应关系。而 4GL 以后的编程语言更多是描述要做什么（Declarative）而不描述具体每一步怎么做（Imperative），具体步骤完全由编译器或解释器决定，例如 SQL（SQL，Structured Query Language，结构化查询语言）就是这样的例子。

习题

1．解释执行的语言相比编译执行的语言有什么优缺点？

这是我们的第一个思考题。本书的思考题通常要求读者系统地总结当前小节的知识，结合以前的知识，并经过一定的推理，然后做答。本书强调的是基本概念，读者应该抓住概念的定义和概念之间的关系来总结，比如本节介绍了很多概念：**程序**由**语句**或**指令**组成，计算机只能执行**低级语言**中的**指令**（汇编语言的指令要先转成机器码才能执行），**高级语言**要执行就必须先翻译成低级语言，翻译的方法有两种——**编译**和**解释**，虽然有这样的不便，但高级语言有一个好处是**平台无关性**。什么是**平台**？一种平台，就是一种**体系结构**，就是一种**指令集**，就是一种

机器语言，这些都可看作是一一对应的，上文并没有用“一一对应”这个词，但读者应该能推理出这个结论，而高级语言和它们不是一一对应的，因此高级语言是**平台无关**的，概念之间像这样的数量对应关系尤其重要。那么编译和解释的过程有哪些不同？主要的不同在于什么时候翻译和什么时候执行。

现在回答这个思考题，根据编译和解释的不同原理，你能否在执行效率和平台无关性等方面做一下比较？

希望读者掌握**以概念为中心的阅读思考习惯**，每读一节就总结一套概念之间的关系图并画在书上空白处。如果读到后面某一节看到一个讲过的概念，但是记不清在哪一节讲过了，没关系，书后的索引可以帮你找到它是在哪一节定义的。

1.2 自然语言和形式语言

自然语言（Natural Language）就是人类讲的语言，比如汉语、英语和法语。这类语言不是人为设计（虽然有人试图强加一些规则）而是自然进化的。形式语言（Formal Language）是为了特定应用而人为设计的语言。例如数学家用的数字和运算符号、化学家用的分子式等。编程语言也是一种形式语言，是专门设计用来表达计算过程的形式语言。

形式语言有严格的语法（Syntax）规则，例如，3+3=6 是一个语法正确的数学等式，而 3=+6$则不是，$H_2O$ 是一个正确的分子式，而 $_2Zz$ 则不是。语法规则是由符号（Token）和结构（Structure）的规则所组成的。Token 的概念相当于自然语言中的单词和标点、数学式中的数和运算符、化学分子式中的元素名和数字，例如 3=+6$的问题之一在于$不是一个合法的数也不是一个事先定义好的运算符，而 $_2Zz$ 的问题之一在于没有一种元素的缩写是 Zz。结构是指 Token 的排列方式，3=+6$还有一个结构上的错误，虽然加号和等号都是合法的运算符，但是不能在等号之后紧跟加号，而 $_2Zz$ 的另一个问题在于分子式中必须把下标写在化学元素名称之后而不是前面。关于 Token 的规则称为词法（Lexical）规则，而关于结构的规则称为语法（Grammar）规则[①]。

当阅读一个自然语言的句子或者一种形式语言的语句时，你不仅要搞清楚每个词（Token）是什么意思，而且必须搞清楚整个句子的结构是什么样的（在自然语言中你只是没有意识到，但确实这样做了，尤其是在读外语时你肯定也意识到了）。这个分析句子结构的过程称为解析（Parse）。例如，当你听到“The other shoe fell.”这个句子时，你理解 the other shoe 是主语而 fell 是谓语动词，一旦解析完成，你就搞懂了句子的意思，如果知道 shoe 是什么东西，fall 意味着什么，这句话是在

① 很不幸，Syntax 和 Grammar 通常都翻译成“语法”，这让初学者非常混乱，Syntax 的含义其实包含了 Lexical 和 Grammar 的规则，还包含一部分语义的规则（例如在 C 程序中变量应先声明后使用）。即使在英文的文献中 Syntax 和 Grammar 也经常混用，在有些文献中 Syntax 的含义不包括 Lexical 规则，只要注意上下文就不会误解。另外，本书在翻译容易引起混淆的时候通常直接用英文名称，例如 Token 没有十分好的翻译，直接用英文名称。

什么上下文（Context）中说的，你还能理解这个句子主要暗示的内容，这些属于语义（Semantic）的范畴。

虽然形式语言和自然语言有很多共同之处，包括 Token、结构和语义，但是也有很多不一样的地方。

歧义性（Ambiguity）

自然语言充满歧义，人们通过上下文的线索和自己的常识来解决这个问题。形式语言的设计要求是清晰的、毫无歧义的，这意味着每个语句都必须有确切的含义而不管上下文如何。

冗余性（Redundancy）

为了消除歧义减少误解，自然语言引入了相当多的冗余。结果是自然语言经常说得啰里啰唆，而形式语言则更加紧凑，极少有冗余。

与字面意思的一致性

自然语言充斥着成语和隐喻（Metaphor），我在某种场合下说“The other shoe fell”，可能并不是说谁的鞋掉了。而形式语言中字面（Literal）意思基本上就是真实意思，也会有一些例外，例如第 2 章要讲的 C 语言转义序列，但即使有例外也会明确规定哪些字面意思不是真实意思，它们所表示的真实意思又是什么。

说自然语言长大的人（实际上没有人例外），往往有一个适应形式语言的困难过程。某种意义上，形式语言和自然语言之间的不同正像诗歌和说明文的区别，当然，前者之间的区别比后者更明显。

诗歌

词语的发音和意思一样重要，全诗作为一个整体创造出一种效果或者表达一种感情。歧义和非字面意思不仅是常见的而且是刻意使用的。

说明文

词语的字面意思显得更重要，并且结构能传达更多的信息。诗歌只能看一个整体，而说明文更适合逐字逐句分析，但仍然充满歧义。

程序

计算机程序是毫无歧义的，字面和本意高度一致，能够完全通过对 Token 和结构的分析加以理解。

现在给出一些关于阅读程序（包括其他形式语言）的建议。首先请记住形式语言远比自然语言紧凑，所以要多花点时间来读。其次，结构很重要，从上到下从左到右读往往不是一个好办法，而应该学会在大脑里解析：识别 Token，分解结构。

最后，请记住细节的影响，诸如拼写错误和标点错误这些在自然语言中可以忽略的小毛病会把形式语言搞得面目全非。

1.3 程序的调试

编程是一件复杂的工作，因为是人做的事情，所以难免经常出错。据说有这样一个典故：早期的计算机体积都很大，有一次一台计算机不能正常工作，工程师们找了半天原因最后发现是一只虫子（Bug）钻进计算机中造成的。从此以后，程序中的错误被叫做Bug，而找到这些Bug并加以纠正的过程就叫做调试（Debug）。有时候调试是一项非常复杂的工作，要求程序员概念明确、逻辑清晰、性格沉稳，还需要一点运气。调试的技能我们在后续的学习中慢慢培养，但首先我们要清楚程序中的Bug分为哪几类。

编译时错误

编译器只能翻译语法正确的程序，否则将导致编译失败，无法生成可执行文件。对于自然语言来说，一点语法错误不是很严重的问题，因为我们仍然可以读懂句子。而编译器就没那么宽容了，哪怕只有一个很小的语法错误，编译器也会输出一条错误提示信息然后罢工，你就得不到想要的结果。虽然大部分情况下编译器给出的错误提示信息就是你出错的代码行，但也有个别时候编译器给出的错误提示信息帮助不大，甚至会误导你。在开始学习编程的前几个星期，你可能会花大量的时间来纠正语法错误。等到有了一些经验之后，还是会犯类似的错误，不过会少得多，而且你能更快地发现错误原因。等到经验更丰富之后你就会觉得，语法错误是最简单最低级的错误，编译器的错误提示也就那么几种，即使错误提示是有误导的也能够立刻找出真正的错误原因是什么。相比下面两种错误，语法错误解决起来要容易得多。

运行时错误

编译器检查不出这类错误，仍然可以生成可执行文件，但在运行时会出错而导致程序崩溃。对于我们接下来的几章将编写的简单程序来说，运行时错误很少见，到了后面的章节你会遇到越来越多的运行时错误。读者在以后的学习中要时刻**注意区分编译时和运行时（Run-time）这两个概念**，不仅在调试时需要区分这两个概念，在学习C语言的很多语法时都需要区分这两个概念，有些事情在编译时做，有些事情则在运行时做。

逻辑错误和语义错误

第三类错误是逻辑错误和语义错误。如果程序里有逻辑错误，编译和运行都会很顺利，看上去也不产生任何错误信息，但是程序没有干它该干的事情，而是干了别的事情。当然不管怎么样，计算机只会按你写的程序去做，问题在于你写的程序不是你真正想要的，这意味着程序的意思（即语义）是错的。找到逻辑错误在哪需要十分清醒的头脑，要通过观察程序的输出回过头来判断它到底在做什么。

通过本书你将掌握的最重要的技巧之一就是调试。调试的过程可能会让你感到一些沮丧，但调试也是编程中最需要动脑的、最有挑战和乐趣的部分。从某种角度看调试就像侦探工作，根据掌握的线索来推断是什么原因和过程导致了你所看到的结果。调试也像是一门实验科学，每次想到哪里可能有错，就修改程序然后再试一次。如果假设是对的，就能得到预期的正确结果，就可以接着调试下一个 Bug，一步一步逼近正确的程序；如果假设错误，只好另外找思路再做假设。“当你把不可能的全部剔除，剩下的——即使看起来再怎么不可能——就一定是事实。”（即使你没看过福尔摩斯也该看过柯南吧）。

也有一种观点认为，编程和调试是一回事，编程的过程就是逐步调试直到获得期望的结果为止。你应该总是从一个能正确运行的小规模程序开始，每做一步小的改动就立刻进行调试，这样的好处是总有一个正确的程序做参考：如果正确就继续编程，如果不正确，那么一定是刚才的小改动出了问题。例如，Linux 操作系统包含了成千上万行代码，但它也不是一开始就规划好了内存管理、设备管理、文件系统、网络等大的模块，一开始它仅仅是 Linus Torvalds 用来琢磨 Intel 80386 芯片而写的小程序。据 Larry Greenfield 说，“Linus 的早期工程之一是编写一个交替打印 AAAA 和 BBBB 的程序，这玩意儿后来进化成了 Linux。”（引自 The Linux User's Guide Beta1 版）在后面的章节中会给出更多关于调试和编程实践的建议。

1.4　第一个程序

在开始写程序之前首先要搭建开发环境，安装编译器、头文件、库文件、开发文档等。在 Linux 系统下如何安装软件包和搭建开发环境不是本书的重点，这些问题需要读者自己解决，但我在这里简单列出需要安装的软件包供读者参考（假定你用的是 Debian 或 Ubuntu 发行版）：

- `gcc: The GNU C compiler`
- `libc6-dev: GNU C Library: Development Libraries and Header Files`
- `manpages-dev: Manual pages about using GNU/Linux for development`
- `manpages-posix-dev: Manual pages about using a POSIX system for development`
- `binutils: The GNU assembler, linker and binary utilities`
- `gdb: The GNU Debugger`
- `make: The GNU version of the "make" utility`

本书所有代码都在 Ubuntu 10.04 LTS（32 位 x86 平台）发行版上编译测试通过。读者如果用其他 Linux 发行版，或者不使用发行版提供的软件包而是用自己从源代码编译出的软件包，则编译运行本书的代码得到的结果会有些不同，但不影响学习。

在 Windows 平台上使用微软的开发环境也可以编译运行本书的大部分代码。从 http://www.microsoft.com/express/Downloads/可以下载免费版的 Visual C++ 2010

Express，建议选择英文版下载安装，因为很多编译选项术语根本没有准确的中文翻译。安装之后打开开始菜单→所有程序→Microsoft Visual Studio 2010 Express→Visual Studio Command Prompt (2010)，进入命令行编译和运行程序，不要使用IDE，我在前言中已经解释过理由了。

在Windows平台上编译运行C程序也可以使用MinGW（GNU开发工具的Windows版本）、Cygwin（在Windows系统中模拟的Linux环境）或者Intel的编译器，本书不做详细介绍。

通常一本教编程的书中第一个例子都是打印“Hello, World.”，这个传统源自参考文献[3]，用C语言写这个程序可以这样写：

例1.1 Hello World

```
#include <stdio.h>

/* main: generate some simple output */

int main(void)
{
    printf("Hello, world.\n");
    return 0;
}
```

在Linux平台上，将这个程序保存成主目录下的main.c，然后编译运行：

```
$ gcc main.c
$ ./a.out
Hello, world.
```

gcc是Linux平台的C编译器，编译后在当前目录下生成可执行文件a.out[②]，直接在命令行输入这个可执行文件的路径就可以执行它。如果不想把文件名叫a.out，可以用gcc的-o参数自己指定文件名：

```
$ gcc main.c -o main
$ ./main
Hello, world.
```

在Windows平台上，将这个程序保存成C:\目录下的main.c，打开Visual Studio命令行，用cl命令编译，然后运行：

```
Setting environment for using Microsoft Visual Studio 2010 x86 tools.

C:\Program Files\Microsoft Visual Studio 10.0\VC>cd c:\

C:\>cl main.c
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version
16.00.30319.01 for 80x86
Copyright (C) Microsoft Corporation. All rights reserved.
```

② “a.out”是“Assembler Output”的缩写，实际上一个C程序要先被编译器翻译成汇编程序，再被汇编器翻译成机器指令，最后还要经过链接器的处理才能成为可执行文件，详见第18.2节。

```
main.c
Microsoft (R) Incremental Linker Version 10.00.30319.01
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:main.exe
main.obj

C:\>main.exe
Hello, world.
```

虽然这只是一个很小的程序，但我们目前暂时还不具备相关的知识来完全理解这个程序，比如程序的第一行，还有程序主体的 int main(void){...return 0;}结构，这些部分我们暂时不详细解释，读者现在只需要把它们看成是每个程序按惯例必须要写的部分（Boilerplate）。但要注意 main 是一个特殊的名字，C 程序总是从 main 里面的第一条语句开始执行的，在这个程序中是指 printf 这条语句。

第 3 行的/* ... */结构是一个注释（Comment），其中可以写一些描述性的话，解释这段程序在做什么。注释只是写给程序员看的，编译器会忽略从/*到*/的所有字符，所以写注释没有语法规则，爱怎么写就怎么写，并且不管写多少都不会被编译进可执行文件中。

printf 语句的作用是把消息打印到屏幕。注意语句的末尾以;号（Semicolon）结束，下一条语句 return 0;也是如此。

C 语言用{}括号（Brace 或 Curly Brace）把语法结构分成组，在上面的程序中 printf 和 return 语句套在 main 的{}括号中，表示它们属于 main 的定义之中。我们看到这两句相比 main 那一行都缩进（Indent）了一些，在代码中可以用若干个空格（Blank）和 Tab 字符来缩进，缩进不是必需的，但这样使我们更容易看出这两行是属于 main 的定义之中的，要写出漂亮的程序必须有整齐的缩进，第 9.1 节将介绍推荐的缩进写法。

正如前面所说，编译器对于语法错误是毫不留情的，如果你的程序有一点拼写错误，例如第一行写成了 stdoi.h，在编译时会得到错误提示：

```
$ gcc main.c
main.c:1:19: error: stdoi.h: No such file or directory
...
```

这个错误提示非常紧凑，初学者往往不容易看明白出了什么错误，即使知道这个错误提示说的是第 1 行有错误，很多初学者对照着书看好几遍也看不出自己这一行哪里有错误，因为他们对符号和拼写不敏感（尤其是英文较差的初学者），他们还不知道这些符号是什么意思又如何能记住正确的拼写？对于初学者来说，最想看到的错误提示其实是这样的：“在 main.c 程序第 1 行的第 19 列，您试图包含一个叫做 stdoi.h 的文件，可惜我没有找到这个文件，但我却找到了一个叫做 stdio.h 的文件，我猜这个才是您想要的，对吗？”可惜没有任何编译器会友善到这个程度，大多数时候你所得到的错误提示并不能直接指出谁是犯人，而只是一个线索，你需要根据这个线索做一些侦探和推理。

有些时候编译器的提示信息不是error而是warning，例如把上例中的printf("Hello, world.\n");改成printf(1);然后编译运行：

```
$ gcc main.c
main.c: In function 'main':
main.c:7: warning: passing argument 1 of 'printf' makes pointer from
integer without a cast
...
$ ./a.out
Segmentation fault
```

这个警告信息是说类型不匹配，但勉强还能配得上。警告信息不是致命错误，编译仍然可以继续，如果整个编译过程只有警告信息而没有错误信息，仍然可以生成可执行文件。但是，警告信息也是不容忽视的。出警告信息说明你的程序写得不够规范，可能有Bug，虽然能编译生成可执行文件，但程序的运行结果往往是不正确的，例如上面的程序运行时出了一个段错误，这属于运行时错误。各种警告信息的严重程度不同，像上面这种警告几乎一定表明程序中有Bug，而另外一些警告只表明程序写得不够规范，一般还是能正确运行的，有些不重要的警告信息gcc默认是不提示的，但这些警告信息也有可能表明程序中有Bug。**一个好的习惯是打开gcc的-Wall选项，让gcc提示所有的警告信息，不管是严重的还是不严重的，然后把这些问题从代码中全部消灭**。比如把上例中的printf("Hello, world.\n");改成printf(0);然后编译运行：

```
$ gcc main.c
$ ./a.out
```

编译既不报错也不报警告，一切正常，但是运行程序什么也不打印。如果打开-Wall选项编译就会报警告了：

```
$ gcc -Wall main.c
main.c: In function 'main':
main.c:7: warning: null argument where non-null required (argument 1)
```

如果printf中的0是你不小心写上去的（例如错误地使用了编辑器的查找替换功能），这个警告就能帮助你发现错误。虽然本书的命令行为了突出重点通常省略-Wall选项，但是强烈建议你写每一个编译命令时都加上-Wall选项。

习题

1. 尽管编译器的错误提示不够友好，但仍然是学习过程中一个很有用的工具。你可以像上面那样，从一个正确的程序开始每次改动一小点，然后编译看是什么结果，如果出错了，就尽量记住编译器给出的错误提示并把改动还原。因为错误是你改出来的，你已经知道错误原因是什么了，所以能很容易地把错误原因和错误提示信息对应起来记住，这样下次你在毫无防备的情况下撞到这个错误提示时就会很容易想到错误原因是什么了。这样反复练习，有了一定的经验积累之后面对编译器的错误提示就会从容得多了。

第 2 章

常量、变量和表达式

2.1 继续 Hello World

在第 1.4 节中，读者应该已经尝试对 Hello world 程序做各种改动看编译运行结果，其中有些改动会导致编译出错，有些改动会影响程序的输出，有些改动则没有任何影响，下面我们总结一下。首先，注释可以跨行，也可以穿插在程序之中，看下面的例子。

例 2.1 带更多注释的 Hello World

```
#include <stdio.h>

/*
 * comment1
 * main: generate some simple output
 */

int main(void)
{
    printf(/* comment2 */"Hello, world.\n"); /* comment3 */
    return 0;
}
```

第一个注释跨了四行，头尾两行是注释的界定符（Delimiter）/*和*/，中间两行开头的*号（Asterisk）并没有特殊含义，只是为了看起来整齐，这不是语法规则而是大家都遵守的 C 代码风格（Coding Style）之一，代码风格将在第 9 章详细介绍。

使用注释需要注意以下两点。

1．注释不能嵌套（Nest）使用，就是说一个注释的文字中不能再出现/*和*/了，例如/* text1 /* text2 */ text3 */是错误的，编译器只把/* text1 /* text2 */看成注释，后面的 text3 */无法解析，因而会报错。

2．有的 C 代码中有类似// comment 的注释，两个/斜线（Slash）表示从这里直到该行末尾的所有字符都属于注释，这种注释不能跨行，也不能穿插在一行代码中间。这是从 C++借鉴的语法，在 C99 中被标准化。

> **提示：C 语言标准**
>
> C 语言的发展历史大致上分为三个阶段：Old Style C、C89 和 C99。Ken

Thompson和Dennis Ritchie最初发明C语言时有很多语法和现在最常用的写法并不一样，但为了向后兼容性（Backward Compatibility），这些语法仍然在C89和C99中保留下来了，本书不详细讲Old Style C，但在必要的地方会加以说明。C89是最早的C语言规范，于1989年提出，1990年首先由ANSI（美国国家标准委员会，American National Standards Institute）推出，后来被采纳为ISO国际标准（ISO/IEC 9899:1990），因而有时也称为C90，最经典的C语言教材参考文献[3]就是基于这个版本的，C89是目前最广泛采用的C语言标准，大多数编译器都完全支持C89。C99标准（ISO/IEC 9899:1999）是在1999年推出的，加入了许多新特性，但目前仍没有得到广泛支持，在C99推出之后相当长的一段时间里，连gcc也没有完全实现C99的所有特性。C99标准详见参考文献[8]。本书讲C的语法以C99为准，但示例代码通常只使用C89语法，很少使用C99的新特性。

C标准的目的是为了精确定义C语言，而不是为了教别人怎么编程，C标准在表达上追求准确和无歧义，却十分不容易看懂，参考文献[4]和参考文献[5]是对C89及其修订版本的阐释（可惜作者没有随C99更新这两本书），比C标准更容易看懂，另外，参考文献[6]也有助于加深对C标准的理解。

像"Hello, world.\n"这种由双引号（Double Quote）引起来的一串字符称为字符串字面值（String Literal），或者简称字符串。注意，程序的运行结果并没有双引号，printf打印出来的只是里面的一串字符Hello, world.，因此双引号是字符串字面值的界定符，夹在双引号中间的一串字符才是它的内容。注意，打印出来的结果也没有\n这两个字符，这是为什么呢？在第1.2节中提到过，C语言规定了一些转义序列（Escape Sequence），这里的\n并不表示它的字面意思，也就是说并不表示\和n这两个字符本身，而是合起来表示一个换行符（Line Feed）。例如我们写三条打印语句：

```
printf("Hello, world.\n");
printf("Goodbye, ");
printf("cruel world!\n");
```

运行的结果是第一条语句单独打到第一行，后两条语句都打到第二行。为了节省篇幅突出重点，以后的例子通常省略#include和int main(void) { ... }这些Boilerplate，但读者在练习时需要加上这些构成一个完整的程序才能编译通过。C标准规定的转义字符有以下几种，如表2.1所示。

表2.1 C标准规定的转义字符

\'	单引号'（Single Quote或Apostrophe）
\"	双引号"
\?	问号?（Question Mark）
\\	反斜线\（Backslash）
\a	响铃（Alert或Bell）

续表

\b	退格（Backspace）
\f	分页符（Form Feed）
\n	换行（Line Feed）
\r	回车（Carriage Return）
\t	水平制表符（Horizontal Tab）
\v	垂直制表符（Vertical Tab）

如果在字符串字面值中要表示单引号和问号，既可以使用转义序列\'和\?，也可以直接用字符'和?，而要表示\或"则必须使用转义序列，因为\字符表示转义而不表示它的字面含义，"表示字符串的界定符而不表示它的字面含义。可见转义序列有两个作用：一是把普通字符转义成特殊字符，例如把字母 n 转义成换行符；二是把特殊字符转义成普通字符，例如\和"是特殊字符，转义后取它的字面值。

C 语言规定了几个控制字符，不能用键盘直接输入，因此采用\加字母的转义序列表示。\a 是响铃字符，在字符终端下显示这个字符的效果是 PC 喇叭发出嘀的一声，在图形界面终端下的输出效果取决于终端的配置。在终端下显示\b 和按下退格键的效果相同。\f 是分页符，主要用于控制打印机在打印源代码时提前分页，这样可以避免一个函数跨两页打印。\n 和\r 分别表示 Line Feed 和 Carriage Return，这两个词来自老式的英文打字机，Line Feed 是跳到下一行（进纸，喂纸，有个喂的动作所以是 feed），Carriage Return 是回到本行开头（Carriage 是卷着纸的轴，随着打字慢慢左移，打完一行就一下子移回最右边，如果你看过欧美的老电影应该能想起来这是什么）。用老式打字机打完一行之后需要这么两个动作，\r\n，所以现在 Windows 平台的文本文件用\r\n 做换行符，许多应用层网络协议（如 HTTP）也用\r\n 做换行符，而 Linux 和各种 UNIX 平台的文本文件只用\n 做换行符。在终端下显示\t 和按下 Tab 键的效果相同，用于在终端下定位表格的下一列，\v 用于在终端下定位表格的下一行。\v 比较少用，\t 比较常用，以后将“水平制表符”简称为“制表符”或 Tab。请读者用 printf 语句试试这几个控制字符的作用。

注意"Goodbye, "末尾的空格，字符串字面值中的空格也算一个字符，也会出现在输出结果中，而程序中别处的空格和 Tab 多一个少一个往往是无关紧要的，不会对编译的结果产生任何影响，例如不缩进不会影响程序的结果，main 后面多几个空格也没影响，但是 int 和 main 之间至少要有一个空格分隔开：

```
int main    (void)
{
printf("Hello, world.\n");
return 0;
}
```

不仅空格和 Tab 是无关紧要的，换行也是如此，我甚至可以把整个程序写成一行，但是 include 必须单独占一行：

```
#include<stdio.h>
int main(void){printf("Hello, world.\n");return 0;}
```

这样也行，但肯定不是好的代码风格，去掉缩进已经很影响可读性了，写成现在

这个样子可读性更差。如果编译器说第2行有错误，也很难判断是第2行的哪个语句出的错。所以，**好的代码风格要求缩进整齐，每个语句一行，适当留空行。**

2.2 常量

常量（Constant）是程序中最基本的元素，有字符（Character）常量、整数（Integer）常量、浮点数（Floating Point）常量和枚举常量。枚举常量将在第7.3节介绍。下面看一个例子：

```
printf("character: %c\ninteger: %d\nfloating point: %f\n", '}', 34,
3.14);
```

字符常量要用单引号括起来，例如上面的'}'，注意单引号只能括一个字符而不能像双引号那样括一串字符，字符常量也可以是一个转义序列，例如'\n'，这时虽然单引号括了两个字符，但实际上只表示一个字符。和字符串字面值中使用转义序列有一点区别，如果在字符常量中要表示双引号"和问号?，既可以使用转义序列\"和\?，也可以直接用字符"和?，而要表示'和\则必须使用转义序列[①]。

在计算机中整数和小数的内部表示方式不同（将在第13章详细介绍），因而在C语言中是两种不同的类型（Type），通常小数在计算机中的表示方式称为浮点数，详见第13.4节。上例的34和3.14分别是整数常量和浮点数常量。

上例的printf语句输出结果和Hello world那个例子不太一样，字符串"character: %c\ninteger: %d\nfloating point: %f\n"并不是按原样打印输出的，而是输出成这样：

```
character: }
integer: 34
floating point: 3.140000
```

printf中的第一个字符串称为格式化字符串（Format String），它规定了后面几个常量以何种格式插入到这个字符串中，在格式化字符串中%号（Percent Sign）后面加上字母c、d、f分别表示字符型、整型和浮点型的转换说明（Conversion Specification），转换说明只在格式化字符串中占个位置，并不出现在最终的打印结果中，这种用法通常叫做占位符（Placeholder）。这也是一种字面意思与真实意思不同的情况，但是转换说明和转义序列又有区别，**转义序列是在编译时处理的，而转换说明是在运行时调用printf函数处理的：**

- 源文件中的字符串字面值是"character: %c\ninteger: %d\nfloating point: %f\n"，\n占两个字符；

① 读者可能会奇怪，为什么需要规定一个转义序列\?呢？因为C语言规定了一些三连符（Trigraph），在某些特殊的终端上缺少某些字符，需要用Trigraph输入，例如用??=表示#字符。Trigraph极不常用，极不常用的C语法在本书中通常不会介绍，介绍这个只是为了让读者理解转义序列的作用，即特殊字符转普通字符、普通字符转特殊字符，?号也是一种特殊字符，要表示其字面意思也需要用转义序列，但如果?号单独出现，不会被误认为是三连符，也可以不用转义序列。

- 编译之后保存在可执行文件中的字符串是 character： %c 换行 integer: %d 换行 floating point: %f 换行，\n 已经被替换成一个换行符，而%c 这两个字符不变；
- 在运行时这个字符串被传给 printf，printf 再把其中的%c、%d、%f 解释成转换说明。

有时候不同类型的数据很容易弄混，例如"5"、'5'、5，如果你注意了它们的界定符就会很清楚，第一个是字符串字面值，第二个是字符，第三个是整数，看了本章后面几节你就知道为什么一定要严格区分它们之间的差别了。

习题

1、我们知道，用\斜线表示转义序列和在 printf 格式化字符串中用%号表示占位符是两种不同的机制，前者在编译时处理，后者在运行时处理。但两者在语法上具有类似的规律，想想在 printf 格式化字符串中怎么表示一个%字符？写个小程序试验一下。

2.3 变量

变量（Variable）是编程语言最重要的概念之一，在程序中变量是一个名字，而这个名字代表的是计算机存储器中的一块空间，可以在里面保存一个值（Value），保存的值是可以随时变的，比如这次存个字符'a'，变量的值就是'a'，下次存个字符'b'，变量的值就变成'b'，正因为变量的值可以随时变所以才叫变量。

常量有不同的类型，变量也有不同的类型，变量的类型决定了它所占的存储空间的大小。在 C 语言中用声明（Declaration）来规定变量的名字和类型，例如下面有四条声明，规定了四个变量 fred、bob、jimmy 和 tom 的类型分别是字符型、整型、单精度浮点型、双精度浮点型：

```
char fred;
int bob;
float jimmy;
double tom;
```

提示：声明和定义

C 语言中的声明有变量声明、函数声明和类型声明三种。本节只讲变量声明，下一章会讲到函数声明，从第 7 章开始我们会看到类型声明。

从另一个角度来看，声明分为“是定义（Definition）的声明”和“不是定义的声明”，那么什么样的声明同时也是定义呢？简单地说，**分配存储空间的声明同时也是定义，不分配存储空间的声明不是定义**。

- 如果一个变量声明要求编译器为它分配存储空间，那么这个声明同时也是变量的定义。本章和接下来几章的示例代码中的变量声明都是要

分配存储空间的，因而都是定义；等学到第 19.2 节我们会看到有些变量声明不分配存储空间，因而不是定义。

- 如果一个函数声明带有函数体，要求编译器为它生成指令（当然也需要分配存储空间来保存这些指令），那么这个声明同时也是函数的定义。在下一章我们会看到带函数体的声明和不带函数体的声明，不带函数体的声明不是函数定义。
- 类型声明总是不分配存储空间的，所以严格来说只有类型声明而没有类型定义，但通常我们习惯说“定义了某种类型”，所以在本书中“类型定义”和“类型声明”表示相同的含义，不加区分。

声明也是以;号结尾，这一点和语句类似，但是在语法上声明和语句是有区别的，语句只能出现在函数体中，而声明既可以出现在函数体中也可以出现在所有函数之外。

浮点型有三种：float 是单精度浮点型；double 是双精度浮点型；long double 是精度更高的浮点型。它们之间的区别和转换规则将在第 14 章详细介绍，在随后的几章中我们只使用 double 类型，上一节介绍的常量 3.14 是 double 类型的常量，printf 的%f 也是 double 型的转换说明（注意%f 不是 float 型的转换说明）。给变量起名不能太随意，上面四个变量的名字就不够好，我们猜不出这些变量是用来存什么的，像下面这样起名就比较有意义：

```
char firstletter;
char lastletter;
int hour, minute;
```

在这个例子中我们还看到两个相同类型的变量（同样是 int 类型的 hour 和 minute）可以一起声明。

给变量起名有一定的限制，C 语言规定必须以字母或下划线_（Underscore）开头，后面可以跟若干个字母、数字、下划线，但不能有其他字符。例如这些是合法的变量名：Abc、__abc__、_123。但这些是不合法的变量名：3abc、ab$。其实这条规则不仅适用于变量名，也适用于所有可以由程序员起名的语法元素，例如以后要讲的函数名、宏定义、结构体成员名等，在 C 语言中这些统称为标识符（Identifier）。

另外要注意，表示类型的 char、int、float、double 等虽然符合上述规则，但也不能用作标识符。在 C 语言中有些单词有特殊意义，不允许用作标识符，这些单词称为关键字（Keyword）或保留字（Reserved Word）。通常用于编程的文本编辑器都会高亮显示（Highlight）这些关键字，所以只要小心一点通常不会误用作标识符。C99 规定的关键字有：

```
auto  break  case  char  const  continue  default  do  double
else  enum  extern  float  for  goto  if  inline  int  long
register  restrict  return  short  signed  sizeof  static  struct
switch  typedef
union  unsigned  void  volatile  while  _Bool  _Complex  _Imaginary
```

还有一点要注意，**一般来说应避免使用以下划线开头的标识符**，以下划线开头的标识符只要不和C语言关键字冲突都是合法的，但是往往被编译器用作一些功能扩展（比如第18.4节讲到gcc的__attribute__语法），C标准库也定义了很多以下划线开头的标识符留作内部使用，所以除非你对编译器的特性和C标准库的实现特别清楚，一般应避免使用这种标识符，以免造成命名冲突。

请记住：**理解一个概念不是把定义背下来就行了，一定要理解它的外延和内涵，也就是什么情况属于这个概念，什么情况不属于这个概念，什么情况虽然属于这个概念但一般推荐的做法（Best Practice）是要尽量避免这种情况，这才算是真正理解了。**

2.4 赋值

定义了变量之后，我们要把值存到它们所表示的存储空间里，可以用赋值（Assignment）语句实现：

```
char firstletter;
int hour, minute;
firstletter = 'a';   /* give firstletter the value 'a' */
hour = 11;           /* assign the value 11 to hour */
minute = 59;        /* set minute to 59 */
```

注意变量一定要先声明后使用，编译器必须先看到变量声明，才知道firstletter、hour和minute是变量名，各自代表一块存储空间。另外，变量声明中的类型表明这个变量代表多大的一块存储空间，这样编译器才知道如何读写这块存储空间。还要注意，这里的等号不表示数学里的相等关系，和1+1=2的等号是不同的，这里的等号表示赋值。在数学上不会有i=i+1这种等式成立，而在C语言中这条语句表示把变量i的存储空间中的值取出来，再加上1，得到的结果再存回i的存储空间中。再比如，在数学上a=7和7=a是一样的，而在C语言中后者是不合法的。总结一下：**定义一个变量，就是分配一块存储空间并给它命名；给一个变量赋值，就是把一个值保存到这块存储空间中。**

变量的定义和赋值也可以一步完成，这称为变量的初始化（Initialization），例如要达到上面代码的效果也可以这样写：

```
char firstletter = 'a';
int hour = 11, minute = 59;
```

其中等号右边的值叫做Initializer，例如上面的'a'、11和59。注意，**初始化是一种特殊的声明，而不是一种赋值语句**。就目前来看，先定义一个变量再给它赋值和定义这个变量的同时给它初始化所达到的效果是一样的，C语言的很多语法规则既适用于赋值也适用于初始化，但在以后的学习中你也会了解到它们之间的不同，请在学习过程中注意总结赋值和初始化的相同和不同之处。

如果在纸上“跑”一个程序 ，可以用一个框表示变量的存储空间，在框的外边标上变量名，在框里记上它的值，如图 2.1 所示。

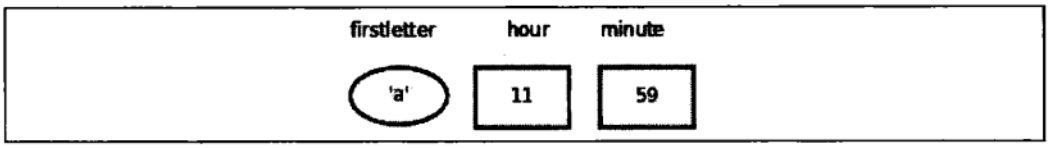


图 2.1 在纸上表示变量

你可以用不同形状的框表示不同类型的变量，这样可以提醒你给变量赋的值必须符合它的类型。如果所赋的值和变量的类型不符会导致编译器报警告或报错（这是一种语义错误），例如：

```
int hour, minute;
hour = "Hello.";        /* WRONG ! */
minute = "59";          /* WRONG !! */
```

注意最后一个语句，把"59"赋给 minute 看起来像是对的，但是类型不对，字符串字面值不能赋给整型变量。

既然可以为变量的存储空间赋值，就应该可以把值取出来用，现在我们取出这些变量的值用 printf 打印：

```
printf("Current time is %d:%d\n", hour, minute);
```

变量名用在等号左边表示赋值，而用在 printf 中表示把它的存储空间中的值取出来替换在那里。不同类型的变量所占的存储空间大小是不同的，数据表示方式也不同，变量的最小存储单位是字节（Byte），在 C 语言中 char 型变量占一个字节，其他类型的变量占多少字节在不同平台上有不同的规定，将在第 14 章详细讨论。

2.5 表达式

常量和变量都可以参与加减乘除运算，例如 1+1、hour-1、hour * 60 + minute、minute/60 等。这里的+ - * /称为运算符（Operator），而参与运算的常量和变量称为操作数（Operand），上面四个由运算符和操作数所组成的算式称为表达式（Expression）。

和数学上规定的一样，hour * 60 + minute 这个表达式应该先算乘再算加，也就是说运算符是有优先级（Precedence）的，*和/是同一优先级，+和-是同一优先级，*和/的优先级高于+和-。对于同一优先级的运算从左到右计算，如果不希望按默认的优先级计算则要加()括号（Parenthesis）。例如(3+4)*5/6 应先算 3+4，再算*5，再算/6。

② 在纸上跑程序是每个初学编程的人都要练的一项基本功，你应该能自己算出程序的运行结果，从而对它的结果有一个预期，如果你自己都不知道这个程序该出什么结果，那交给计算机跑出来的结果是对是错你如何判断？

前面讲过打印语句和赋值语句，现在我们定义：在任意表达式后面加个;号也是一种语句，称为表达式语句。例如：

```
hour * 60 + minute;
```

这是个合法的语句，但这个语句在程序中起不到任何作用，把 hour 的值和 minute 的值取出来相加和相乘，得到的计算结果却没有保存，白算了一通。再比如：

```
int total_minute;
total_minute = hour * 60 + minute;
```

这个语句就很有意义，把计算结果保存在另一个变量 total_minute 里。事实上等号也是一种运算符，称为赋值运算符，赋值语句就是一种表达式语句，等号的优先级比+和*都低，所以先算出等号右边的结果然后才做赋值操作，整个表达式 total_minute = hour * 60 + minute 加个;号构成一个语句。

任何表达式都有值和类型两个基本属性。hour * 60 + minute 的值是由三个 int 型的操作数计算出来的，所以这个表达式的类型也是 int 型。同理，表达式 total_minute = hour * 60 + minute 的类型也是 int，它的值是多少呢？C 语言规定等号运算符的计算结果就是等号左边被赋予的那个值，所以这个表达式的值和 hour * 60 + minute 的值相同，也和 total_minute 被赋值之后的值相同。

等号运算符还有一个和+ - * /不同的特性，如果一个表达式中出现多个等号，不是从左到右计算而是从右到左计算，例如：

```
int total_minute, total;
total = total_minute = hour * 60 + minute;
```

计算顺序是先算 hour * 60 + minute 得到一个结果，然后算右边的等号，就是把 hour * 60 + minute 的结果赋给变量 total_minute，这个结果同时也是整个表达式 total_minute = hour * 60 + minute 的值，再算左边的等号，即把这个值再赋给变量 total。如果一个操作数的左右两侧各有一个相同优先级的运算符，这个操作数与左边的运算符结合还是与右边的运算符结合取决于运算符的结合性（Associativity），相同优先级的运算符应该具有相同的结合性，+ -和* /是左结合的，而等号是右结合的。在上面的表达式中，操作数 total_minute 的左右两边都有等号，应该和右边的等号结合，相当于 total = (total_minute = hour * 60 + minute)，而不是(total = total_minute) = hour * 60 + minute。

现在我们总结一下到目前为止学过的语法规则：

表达式 → 标识符

表达式 → 常量

表达式 → 字符串字面值

表达式 → (表达式)

表达式 → 表达式 + 表达式

表达式 → 表达式 - 表达式

表达式 → 表达式 * 表达式

表达式 → 表达式 / 表达式

表达式 → 表达式 = 表达式

语句 → 表达式;

语句 → printf(表达式, 表达式, 表达式, ...);

变量声明 → 类型 标识符 = Initializer, 标识符 = Initializer, ...;

（= Initializer 的部分可以不写）

注意，**本书所列的语法规则都是简化过的，是不准确的，目的是为了便于初学者理解**，比如上面所列的语法规则并没有描述运算符的优先级和结合性。完整的 C 语法规则请查看参考文献[8]的 Annex A。

表达式可以是单个的常量或变量，也可以是根据以上规则组合而成的更复杂的表达式。以前我们用 printf 打印常量或变量的值，现在可以用 printf 打印更复杂的表达式的值，例如：

```
printf("%d:%d is %d minutes after 00:00\n", hour, minute, hour * 60 + minute);
```

编译器在翻译这条语句时，首先根据上述语法规则把这个语句解析成如图 2.2 所示的语法树，然后再根据语法树生成相应的指令。

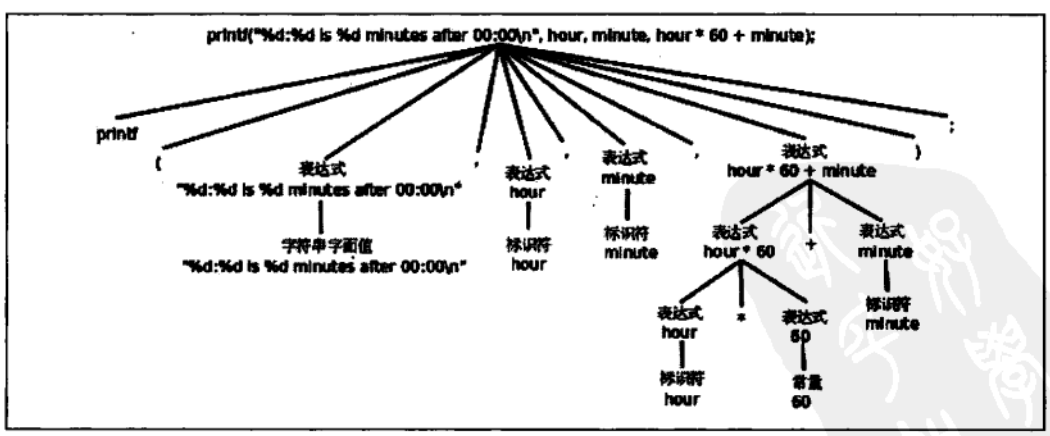


图 2.2 语法树

语法树的每一步分解利用一条语法规则，直到分解成 Token 为止，所以语法树的末端全部是 Token。语法解析的过程十分复杂，我们不深入讨论如何分解，而是

反过来从组合的角度来理解语法规则，比如表达式 hour * 60 + minute 是这样组合而成的：

1．hour 是标识符，根据规则“表达式 → 标识符”，它也是表达式；

2．60 是常量，根据规则“表达式 → 常量”，它也是表达式；

3．既然 hour 和 60 都是表达式，根据规则“表达式 → 表达式 * 表达式”，hour * 60 可以组合成表达式；

4．minute 是标识符，根据规则“表达式 → 标识符”，它也是表达式；

5．既然 hour * 60 和 minute 都是表达式，根据规则“表达式 → 表达式 + 表达式”，hour * 60 + minute 可以组合成表达式。

根据这些语法规则进一步组合可以写出更复杂的语句，比如在一条语句中完成计算、赋值和打印功能：

```
printf("%d:%d is %d minutes after 00:00\n", hour, minute, total_minute
= hour * 60 + minute);
```

理解组合（Composition）规则是理解语法规则的关键所在，正因为可以根据语法规则任意组合，我们才可以用简单的常量、变量、表达式、语句和声明搭建出任意复杂的程序，以后我们学习新的语法规则时会进一步体会到这一点。从上面的例子可以看出，表达式不宜过度组合，否则会给阅读和调试带来困难。

根据语法规则组合出来的表达式在语义上并不总是正确的，例如：

```
minute + 1 = hour;
```

等号左边的表达式要求表示一个存储位置而不是一个值，这是等号运算符和+ - * / 运算符的又一个显著的不同。有的表达式既可以表示一个存储位置也可以表示一个值，而有的表达式只能表示值，不能表示存储位置，例如 minute + 1 这个表达式就不能表示存储位置，放在等号左边是语义错误。表达式所表示的存储位置称为左值（lvalue），允许放在等号左边，而之前我们所说的表达式的值也称为右值（rvalue），只能放在等号右边。上面的话换一种说法就是：**有的表达式既可以做左值也可以做右值，而有的表达式只能做右值**。目前我们学过的表达式中只有变量可以做左值，可以做左值的表达式还有几种，以后会讲到。

我们看一个有意思的例子，如果定义三个变量 int a, b, c;，表达式 a = b = c 是合法的，先求 b = c 的值，再把这个值赋给 a，而表达式(a = b) = c 是不合法的，先求(a = b)的值没问题，但(a = b)这个表达式不能再做左值了，因此放在= c 的等号左边是错的。

关于整数除法运算有一点特殊之处：

```
hour = 11;
minute = 59;
```

```
printf("%d and %d hours\n", hour, minute / 60);
```

执行结果是 11 and 0 hours，也就是说 59/60 得 0，这是因为两个 int 型操作数相除的表达式仍为 int 型，只能保存计算结果的整数部分，即使小数部分是 0.98 也要舍去。

向下取整的运算称为 Floor，用数学符号「⌋表示；向上取整的运算称为 Ceiling，用数学符号「⌉表示。例如：

```
⌊59/60⌋=0
⌈59/60⌉=1
⌊-59/60⌋=-1
⌈-59/60⌉=0
```

在 C 语言中整数除法取的既不是 Floor 也不是 Ceiling，无论操作数是正是负总是把小数部分截掉，在数轴上向零的方向取整（Truncate towards Zero），或者说当操作数为正的时候相当于 Floor，当操作数为负的时候相当于 Ceiling。回到先前的例子，要得到更精确的结果可以这样：

```
printf("%d hours and %d percent of an hour\n", hour, minute * 100 / 60);
printf("%d and %f hours\n", hour, minute / 60.0);
```

在第二个 printf 中，表达式是 minute / 60.0，60.0 是 double 型的，/运算符要求左右两边的操作数类型一致，而现在并不一致。C 语言规定了一套隐式类型转换规则，在这里编译器自动把左边的 minute 也转成 double 型来计算，整个表达式的值也是 double 型的，在格式化字符串中应该用%f 转换说明与之对应。本来编程语言作为一种形式语言要求有简单而严格的规则，自动类型转换规则不仅很复杂，而且使 C 语言的形式看起来也不那么严格了，C 语言这么设计是为了书写程序简便而做的折中，有些事情编译器可以自动做好，程序员就不必每次都写一堆繁琐的转换代码。然而 C 语言的类型转换规则非常难掌握，本书的前几章会尽量避免类型转换，到第 14.3 节再集中解决这个问题。

习题

1、假设变量 x 和 n 是两个正整数，我们知道 x/n 这个表达式的结果要取 Floor，例如 x 是 17，n 是 4，则结果是 4。如果希望结果取 Ceiling 应该怎么写表达式呢？例如 x 是 17，n 是 4，则结果是 5；x 是 16，n 是 4，则结果是 4。

2.6 字符类型与字符编码

字符常量或字符型变量也可以当做整数参与运算，例如：

```
printf("%c\n", 'a'+1);
```

执行结果是 b。

我们知道，符号在计算机内部也用数字表示，每个字符在计算机内部用一个整数

表示，称为字符编码（Character Encoding），目前最常用的是 ASCII 码（American Standard Code for Information Interchange），详见表 A.1。表中每一栏的最后一列是字符，前三列分别是用十进制（Dec）、十六进制（Hx）和八进制（Oct）表示的字符编码，各种进制之间的换算将在第 13.2 节介绍。从十进制那一列可以看出 ASCII 码的取值范围是 0～127。表中的很多字符是不可见字符（Non-printable Character）和空白字符（Whitespace），不能像字母 a 这样把字符本身填在表中，而是用一个名字来描述该字符，例如 CR（Carriage Return）、LF（NL Line Feed，Newline）、DEL 等。作为练习，请读者查一查表 2.1 中的字符在 ASCII 码表中的什么位置。

回到刚才的例子，在 ASCII 码中字符 a 是 97，字符 b 是 98。计算'a'+1 这个表达式，应该按 ASCII 码把'a'当做整数值 97，然后加 1，得到 98，然后 printf 把 98 这个整数值当做 ASCII 码来解释，打印出相应的字符 b。

之前我们说“整型”是指 int 型，而现在我们知道 char 型本质上就是整数，只不过取值范围比 int 型小，所以**以后我们把 char 型和 int 型统称为整数类型（Integer Type）或简称整型**，后面我们还要学习几种类型也属于整型，将在第 14.1 节详细介绍。

字符'a'～'z'、'A'～'Z'、'0'～'9'的 ASCII 码都是连续的，因此表达式'a'+25 和'z'的值相等，'0'+9 和'9'的值也相等。注意'0'～'9'的 ASCII 码是十六进制的 30～39，和整数值 0～9 是不相等的。

字符也可以用 ASCII 码转义序列表示，这种转义序列由\加上 1～3 个八进制数字组成，或者由\x 加上 1～2 个十六进制数字组成，可以用在字符常量或字符串字面值中。例如'\0'表示 NUL 字符（Null Character），'\11'或'\x9'表示 Tab 字符，"\11"或"\x9"表示由 Tab 字符组成的字符串。注意'0'的 ASCII 码是 48，而'\0'的 ASCII 码是 0，两者是不同的。

③ 空白字符在不同的上下文中有不同的含义，在 C 语言中空白字符定义为空格、水平 Tab(\t)、垂直 Tab（\v）、换行（\r 和\n）和分页符（\f），本书在使用“空白字符”这个词时会明确说明在当前上下文中空白字符指的是哪些字符。

第 3 章

简单函数

3.1 数学函数

在数学中我们用过 sin 和 ln 这样的函数，例如 sin(π/2)=1，ln1=0 等，在 C 语言中也可以使用这些函数（数学函数 sin 在 C 标准库中就是 sin 函数，而数学函数 ln 在 C 标准库中对应的是 log 函数）。

例 3.1 在 C 语言中使用数学函数

```
#include <math.h>
#include <stdio.h>

int main(void)
{
        double pi = 3.1416;
        printf("sin(pi/2)=%f\nln1=%f\n", sin(pi/2), log(1.0));
        return 0;
}
```

编译运行这个程序，结果如下：

```
$ gcc main.c -lm
$ ./a.out
sin(pi/2)=1.000000
ln1=0.000000
```

在数学中写一个函数有时候可以省略括号，而 C 语言要求一定要加上括号，例如 log(1.0)。在 C 语言的术语中，1.0 是参数，log 是函数（Function），log(1.0)是函数调用（Function Call）。sin(pi/2)和 log(1.0)这两个函数调用在我们的 printf 语句中处于什么位置呢？在上一章讲过，这应该是写表达式的位置，因此函数调用也是一种表达式。log(1.0)这个表达式由操作数 log 和函数调用运算符()括号组成，函数调用运算符是一种后缀运算符（Postfix Operator），()括号及其中的参数是操作数 log 的后缀。操作数 log 是一个函数名（Function Designator），它的类型是一种函数类型（Function Type）。log(1.0)这个表达式的值是取自然对数运算的结果，类型是 double 型，在 C 语言中函数调用表达式的值称为函数的返回值（Return Value）。总结一下我们新学的语法规则：

表达式 → 函数名

表达式 → 表达式(参数列表)

参数列表 → 表达式, 表达式, ...

现在我们可以完全理解 printf 语句了：原来 printf 也是一个函数，上例中的 printf("sin(pi/2)=%f\nln1=%f\n", sin(pi/2), log(1.0))是带三个参数的函数调用，而函数调用也是一种表达式，因此 printf 语句也是表达式语句的一种。但是 printf 感觉不像一个数学函数，为什么呢？因为像 log 这种函数，我们传进去一个参数会得到一个返回值，我们调用 log 函数就是为了得到它的返回值，至于 printf，通常我们并不关心它的返回值（事实上它也有返回值，表示实际打印的字符数），我们调用 printf 不是为了得到它的返回值，而是为了利用它所产生的副作用（Side Effect）——打印。**C 语言的函数可以有 Side Effect，这一点是它和数学函数在概念上的根本区别。**

Side Effect 这个概念也适用于运算符组成的表达式。比如 a + b 这个表达式也可以看成一个函数调用，把运算符+看作函数，它的两个参数是 a 和 b，返回值是两个参数的和，传入两个参数，得到一个返回值，并没有产生任何 Side Effect。而赋值运算符是有 Side Effect 的，如果把 a = b 这个表达式看成函数调用，返回值就是所赋的值，既是 b 的值也是 a 被赋予的值，但除此之外还产生了 Side Effect，就是变量 a 被改变了，改变计算机存储单元里的数据或者做输入输出操作都算 Side Effect。

回想一下我们的学习过程，一开始我们说赋值是一种语句，后来学了表达式，我们说赋值语句是表达式语句的一种；一开始我们说 printf 是一种语句，现在学了函数，我们又说 printf 也是表达式语句的一种。随着我们一步步的学习，把原来看似不同类型的语句统一成一种语句了。学习的过程总是这样，初学者一开始接触的很多概念从严格意义上说是错的，但是很容易理解，随着一步步学习，在理解原有概念的基础上不断纠正，不断泛化（Generalize）。比如一年级老师说小数不能减大数，其实这个概念是错的，后来引入了负数就可以减了，后来引入了分数，原来的正数和负数的概念就泛化为整数，上初中学了无理数，原来的整数和分数的概念就泛化为有理数，再上高中学了复数，有理数和无理数的概念就泛化为实数。坦白说，到目前为止本书的很多说法都是不完全正确的，但这是学习理解的必经阶段，到后面的章节都会逐步纠正的。

程序第一行的#号（Pound Sign，Number Sign 或 Hash Sign）和 include 表示包含一个头文件（Header File），后面尖括号（Angle Bracket）中就是文件名（这些头文件通常位于/usr/include 目录下）。头文件中声明了我们程序中使用的库函数，根据先声明后使用的原则，要使用 printf 函数必须包含 stdio.h，要使用数学函数必须包含 math.h，如果什么库函数都不使用就不必包含任何头文件，例如写一个程序 int main(void){int a;a=2;return 0;}，不需要包含头文件就可以编译通过，当然这个程序什么也做不了。

使用 math.h 中声明的库函数还有一点特殊之处，gcc 命令行必须加-lm 选项，因为数学函数位于 libm.so 库文件中（这些库文件通常位于/lib 目录下），-lm 选项

告诉编译器，我们程序中用到的数学函数要到这个库文件里找。注意库文件名是libm，但使用-l 选项指定库文件时省略 lib，只写成-lm。本书用到的大部分库函数（例如 printf）位于 libc.so 库文件中，使用 libc.so 中的库函数在编译时不需要加-lc 选项，当然加了也不算错，因为这个选项是 gcc 的默认选项。关于头文件和库函数目前理解这么多就可以了，到第 19 章再详细解释。

> **提示：C 标准库和 glibc**
>
> C 标准主要由两部分组成，一部分描述 C 的语法，另一部分描述 C 标准库。C 标准库定义了一组标准头文件，每个头文件中包含一些相关的函数、变量、类型声明和宏定义。要在一个平台上支持 C 语言，不仅要实现 C 编译器，还要实现 C 标准库，这样的实现才算符合 C 标准。不符合 C 标准的实现也是存在的，例如很多单片机的 C 语言开发工具中只有 C 编译器而没有完整的 C 标准库。
>
> 在 Linux 平台上最广泛使用的 C 函数库是 glibc，其中包括 C 标准库的实现，也包括本书第三部分介绍的所有系统函数。几乎所有 C 程序都要调用 glibc 的库函数，所以 glibc 是 Linux 平台 C 程序运行的基础。glibc 提供一组头文件和一组库文件，最基本、最常用的 C 标准库函数和系统函数在 libc.so 库文件中，几乎所有 C 程序的运行都依赖于 libc.so，有些做数学计算的 C 程序除了 libc.so 之外还依赖于 libm.so，还有很多 C 程序依赖于 glibc 的其他库文件。以后我说 libc 时专指 libc.so 这个库文件，而说 glibc 时指的是 glibc 提供的所有库文件。
>
> glibc 并不是 Linux 平台唯一的基础 C 函数库，也有人在开发别的 C 函数库，比如适用于嵌入式系统的 uClibc。

3.2 自定义函数

我们不仅可以调用 C 标准库提供的函数，也可以定义自己的函数，事实上我们已经这么做了：我们定义了 main 函数。例如：

```
int main(void)
{
        int hour = 11;
        int minute = 59;
        printf("%d and %d hours\n", hour, minute / 60);
        return 0;
}
```

main 函数的特殊之处在于执行程序时它自动被操作系统调用，操作系统就认准了 main 这个名字，除了名字特殊之外，main 函数和别的函数没有区别。我们对照着 main 函数的定义来看语法规则：

函数定义 → 返回值类型 函数名(参数列表) 函数体

函数体 → { 语句列表 }

语句列表 → 语句列表项 语句列表项 ...

语句列表项 → 语句

语句列表项 → 变量声明、类型声明或非定义的函数声明

非定义的函数声明 → 返回值类型 函数名(参数列表);

我们稍后再详细解释“函数定义”和“非定义的函数声明”的区别。从第 7 章开始我们才会看到类型声明，所以现在暂不讨论。

给函数命名也要遵循上一章讲过的标识符命名规则。由于我们定义的 main 函数不带任何参数，参数列表应写成 void。函数体可以由若干条语句和声明组成，C89 要求所有声明写在所有语句之前（本书的示例代码都遵循这一规定），而 C99 的新特性允许语句和声明按任意顺序排列，只要每个标识符都遵循先声明后使用的原则就行。main 函数的返回值是 int 型的，return 0;这个语句表示返回值是 0，main 函数的返回值是返回给操作系统看的，因为 main 函数是被操作系统调用的，通常程序执行成功就返回 0，在执行过程中出错就返回一个非零值。比如我们将 main 函数中的 return 语句改为 return 4;再执行它，执行结束后可以在 Shell 中看到它的退出状态（Exit Status）：

```
$ ./a.out
11 and 0 hours
$ echo $?
4
```

$?是 Shell 中的一个特殊变量，表示上一条命令的退出状态。关于 main 函数需要注意两点：

1. 参考文献[3]上的 main 函数定义写成 main(){...}的形式，不写返回值类型也不写参数列表，这是 Old Style C 的风格。Old Style C 规定不写返回值类型就表示返回 int 型，不写参数列表就表示参数类型和个数没有明确指出。这种宽松的规定使编译器无法检查程序中可能存在的 Bug，增加了调试难度，不幸的是现在的 C 标准为了兼容旧的代码仍然保留了这种语法，但读者绝不应该继续使用这种语法。

2. 其实操作系统在调用 main 函数时是传参数的，main 函数最标准的形式应该是 int main(int argc, char *argv[])，在第 22.6 节详细介绍。C 标准也允许 int main(void) 这种写法，如果不使用系统传进来的两个参数也可以写成这种形式。但除了这两种形式之外，定义 main 函数的其他写法都是错误的或不可移植的。

关于返回值和 return 语句我们将在第 5.1 节详细讨论，我们先从不带参数也没有返回值的函数开始学习定义和使用函数。

例 3.2 最简单的自定义函数

```
#include <stdio.h>

void newline(void)
```

```
{
        printf("\n");
}

int main(void)
{
        printf("First Line.\n");
        newline();
        printf("Second Line.\n");
        return 0;
}
```

执行结果是：

```
First Line.

Second Line.
```

我们定义了一个newline函数给main函数调用，它的作用是打印一个换行，所以执行结果中间多了一个空行。newline函数不仅不带参数，也没有返回值，返回值类型为void表示没有返回值 ，这说明我们调用这个函数完全是为了利用它的Side Effect。如果我们想要多次插入空行就可以多次调用newline函数：

```
int main(void)
{
        printf("First Line.\n");
        newline();
        newline();
        newline();
        printf("Second Line.\n");
        return 0;
}
```

如果我们总需要三个三个地插入空行，我们可以再定义一个threelines函数每次插入三个空行。

例3.3 较简单的自定义函数

```
#include <stdio.h>

void newline(void)
{
```

① 敏锐的读者可能会发现一个矛盾：如果函数newline没有返回值，那么表达式newline()不就没有值了吗？然而上一章讲过任何表达式都有值和类型两个基本属性。其实这正是设计void这么一个关键字的原因：首先从语法上规定没有返回值的函数调用表达式是void类型的，有一个void类型的值，这样任何表达式都有值，不必考虑特殊情况，编译器的语法解析比较容易实现；然后从语义上规定void类型的表达式不能参与运算，因此newline() + 1这样的表达式不能通过语义检查，从而兼顾了语法上的一致和语义上的不矛盾。在C语言中这个问题通过语义检查来解决，而在别的编程语言中又有不同的解决办法，比如Pascal语言从语法上区分有返回值和没有返回值的函数，有返回值的称为函数，没有返回值的称为过程（Procedure），在本书中函数和过程表示相同的含义。

```
        printf("\n");
}

void threelines(void)
{
        newline();
        newline();
        newline();
}

int main(void)
{
        printf("Three lines:\n");
        threelines();
        printf("Another three lines.\n");
        threelines();
        return 0;
}
```

通过这个简单的例子可以体会到：

1．同一个函数可以被多次调用；

2．可以用一个函数调用另一个函数，后者再去调用第三个函数；

3．通过自定义函数可以给一组复杂的操作起一个简单的名字，例如 threelines。对于 main 函数来说，只需要通过 threelines 这个简单的名字来调用就行了，不必知道打印三个空行具体怎么做，所有的复杂操作都被隐藏在 threelines 这个名字后面；

4．使用自定义函数可以使代码更简洁，main 函数在任何地方想打印三个空行只需调用一个简单的 threelines()，而不必每次都写三个 printf("\n")。

读代码和读文章不一样，按从上到下从左到右的顺序读代码未必是最好的。比如上面的例子，按源文件的顺序应该是先看 newline 再看 threelines 再看 main。如果你换一个角度，按代码的执行顺序来读也许会更好：首先执行的是 main 函数中的语句，在一条 printf 语句之后调用了 threelines，这时再去看 threelines 的定义，其中又调用了 newline，这时再去看 newline 的定义，newline 里面有一条 printf 语句，执行完成后返回 threelines，这里还剩下两次 newline 调用，效果也都一样，执行完之后返回 main，接下来又是一条 printf 和一条 threelines，如图 3.1 所示。

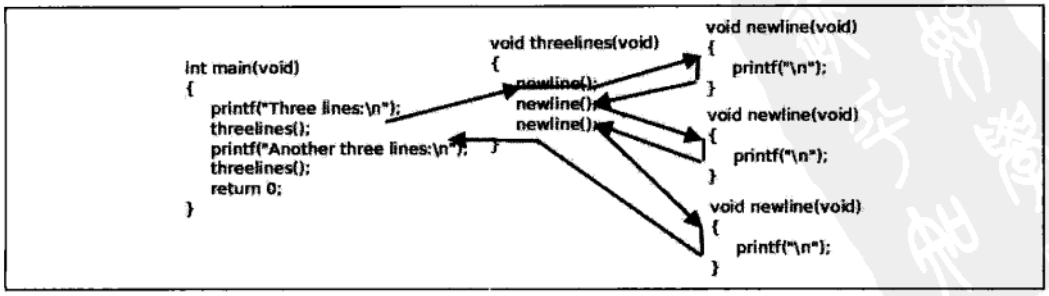


图 3.1　函数调用的执行顺序

读代码的过程就是模仿计算机执行程序的过程，我们不仅要记住当前读到了哪一行代码，还要记住现在读的代码是被哪个函数调用的，这段代码返回后应该从上一个函数的什么地方接着往下读。

现在澄清一下函数声明、函数定义、函数原型（Prototype）这几个概念。比如 void threelines(void)这一行，声明了一个函数的名字、参数类型和个数、返回值类型，这称为函数原型。在代码中也可以单独写一个函数原型后面加;号结束，而不写函数体，例如：

```
void threelines(void);
```

这种写法只能叫函数声明而不能叫函数定义，上一章讲过，只有带函数体的声明才叫函数定义，因为编译器只有见到函数体才能生成指令，并且分配存储空间来保存这些指令。那么没有函数体的函数声明有什么用呢？它为编译器提供了有用的信息，编译器见到函数原型（不管带不带函数体）就明确了这个函数的名字、参数类型和返回值，之后编译器碰到函数调用代码就知道该生成什么样的指令来实现函数调用了，所以函数原型应该出现在函数调用之前，这也是遵循“先声明后使用”的原则。

在上面的例子中，main 调用 threelines，threelines 再调用 newline，要保证每个函数的原型出现在调用之前，就只能按先 newline 再 threelines 再 main 的顺序定义了。如果使用不带函数体的声明，则可以改变函数的定义顺序：

```
#include <stdio.h>

void newline(void);
void threelines(void);

int main(void)
{
        ...
}

void newline(void)
{
        ...
}

void threelines(void)
{
        ...
}
```

这样仍然遵循了先声明后使用的原则。

由于有 Old Style C 语法的存在，并非所有函数声明都包含完整的函数原型，例如 void threelines();这个声明并没有明确指出参数类型和个数，所以不算函数原型，这个声明提供给编译器的信息只有函数名和返回值类型。如果在这样的声明之后调用函数，编译器不知道参数的类型和个数，就不会做语法检查，所以很容易引入 Bug。读者需要了解这个知识点以便维护别人用 Old Style C 风格写的代码，但

绝不应该按这种风格写新的代码。

如果在调用函数之前没有声明会怎么样呢？有的读者也许碰到过这种情况，我可以解释一下，但绝不推荐这种写法。比如按上面的顺序定义这三个函数，但是把开头的两行声明去掉：

```
#include <stdio.h>

int main(void)
{
        printf("Three lines:\n");
        threelines();
        printf("Another three lines.\n");
        threelines();
        return 0;
}

void newline(void)
{
        printf("\n");
}

void threelines(void)
{
        newline();
        newline();
        newline();
}
```

编译时会报警告：

```
$ gcc main.c
main.c:17: warning: conflicting types for 'threelines'
main.c:6: note: previous implicit declaration of 'threelines' was
here
```

但也能编译通过，运行结果也对。这里涉及的语法规则称为函数的隐式声明（Implicit Declaration）。

- 由于在 main 函数中调用 threelines()之前并没有声明它，编译器只能根据函数调用来猜测它的原型，比如根据调用 foo(2.1, 3.3)可以猜测 foo 函数有两个 double 型的参数，而我们调用 threelines()没有传任何参数，所以编译器认为 threelines 函数的参数类型是 void，另外，编译器认为所有隐式声明的返回值类型都是 int，所以 threelines 函数的隐式声明是 int threelines (void);，这样参数和返回值类型都确定下来了，编译器可以根据这些信息为 threelines()调用生成相应的指令。
- 然后编译器接着往下看，看到 threelines 函数的原型是 void threelines(void)，与先前建立的隐式声明不一致（返回值类型不同），所以报警告。好在我们也没用到这个函数的返回值，执行结果仍然正确。假如我们在 main 函数中写 int i = threelines();就错了，但编译也能通过，也给出同样的警告信息。

注意区分以下两个概念：

- 如果调用函数时参数列表为空，并且缺少函数原型，则编译器根据隐式声明规则认为参数类型是 void；
- 如果声明函数时参数列表为空，则这个声明属于 Old Style C 语法，不算函数原型，编译器认为参数类型和个数没有明确指出。

3.3 形参和实参

下面我们定义一个带参数的函数，我们需要在函数定义中指明参数的个数和每个参数的类型，定义参数就像定义变量一样，需要为每个参数指明类型，参数的命名也要遵循标识符命名规则。

例 3.4 带参数的自定义函数

```
#include <stdio.h>

void print_time(int hour, int minute)
{
        printf("%d:%d\n", hour, minute);
}

int main(void)
{
        print_time(23, 59);
        return 0;
}
```

如果写一个非定义的函数声明，可以只写参数类型而不写参数名，例如：

```
void print_time(int, int);
```

这样的声明仍然包含了函数名、参数类型和个数、返回值类型的完整信息，所以也算函数原型。注意，定义变量时可以把相同类型的变量列在一起，而定义参数却不可以，例如下面这样的定义是错的：

```
void print_time(int hour, minute)
{
        printf("%d:%d\n", hour, minute);
}
```

初学者肯定都乐意看到这句话：“变量是这样定义的，参数也是这样定义的，一模一样”，这意味着不用专门去记住参数应该怎么定义了。谁也不愿意看到这句话：“定义变量可以这样写，而定义参数却不可以”。C 语言的设计者也不希望自己设计的语法规则里到处都是例外，一个容易被用户接受的设计应该遵循最少例外原则（Rule of Least Surprise）。其实这条规定也不算十分例外，也是可以理解的，我们看参数列表的语法规则：

参数列表 → 列表项, 列表项, ...

列表项 → 类型 标识符

参数列表中的,号（Comma）是列表项之间的分隔符，如果允许写 void foo(int hour, minute, char c)这样的声明，那就是允许一部分列表项有类型一部分列表项没有类型，还要规定没有类型的列表项和前一列表项的类型相同，那编译器实现起来就复杂了。

另外一个问题是，如果仿照变量声明把参数列表的语法规则改成下面这样不是很好吗？

参数列表 → 列表项; 列表项; ...

列表项 → 类型 标识符, 标识符, ...

按照这样的语法规则，函数声明就可以写成 void foo(int hour, minute; char c)，但为什么 C 语言没有这样规定呢？这也是从 Old Style C 继承下来的，Old Style C 是这样声明参数的：

```
void foo(x, y, z)
int x;
char z;
{
        ...
}
```

现在的 C 编译器仍然支持这种语法。上例中不写类型的参数 y 默认是 int 型。

学习编程语言不应该死记各种语法规定，如果能够想清楚设计者这么规定的原因（Rationale），不仅有助于记忆，而且会有更多收获。本书在必要的地方会解释一些 Rationale，或者启发读者自己去思考，例如上一节在脚注中解释了 void 关键字的 Rationale。参考文献[6]是随 C99 标准一起发布的，值得参考。

总的来说，C 语言的设计是非常优美的，只要理解了少数基本概念和基本原则就可以根据组合规则写出任意复杂的程序，很少有例外的规定说这样组合是不允许的，或者那样类推是错误的。相反，C++的设计就非常复杂，充满了例外，全世界没几个人能把 C++的所有规则都牢记于心，因而 C++的设计一直饱受争议，这个观点在参考文献[7]中有详细阐述。

在本书中，凡是提醒读者注意的地方都是多少有些 Surprise 的地方，初学者如果按常理来想很可能要想错，所以需要特别提醒一下。而初学者容易犯的另外一些错误，完全是因为没有掌握好基本概念和基本原理，或者根本无视组合规则而全凭自己主观臆断所致，对这一类问题本书不会做特别的提醒，例如有的初学者看完第 2 章之后会这样打印 π 的值：

```
double pi=3.1416;
printf("pi\n");
```

之所以会犯这种错误，一是不理解 Literal 的含义，二是自己想当然地把变量名组

合到字符串里去，而事实上根本没有这条语法规则。如果连这样的错误都需要在书上专门提醒，就好比提醒小孩吃饭一定要吃到嘴里，不要吃到鼻子里，更不要吃到耳朵里一样。

回到正题。我们调用 print_time(23, 59)时，函数中的参数 hour 就代表 23，参数 minute 就代表 59。确切地说，当我们讨论函数中的 hour 这个参数时，我们所说的“参数”是指形参（Parameter），当我们讨论传一个参数 23 给函数时，我们所说的“参数”是指实参（Argument），但我习惯都叫参数而不习惯总把形参、实参这两个文绉绉的词挂在嘴边（事实上大多数人都不习惯），读者可以根据上下文判断我说的到底是形参还是实参。记住这条基本原理：**形参相当于函数中定义的变量，调用函数传递参数的过程相当于定义形参变量并且用实参的值来初始化**。例如这样调用：

```
void print_time(int hour, int minute)
{
        printf("%d:%d\n", hour, minute);
}

int main(void)
{
        int h = 23, m = 59;
        print_time(h, m);
        return 0;
}
```

相当于在函数 print_time 中执行了以下代码：

```
int hour = h;
int minute = m;
printf("%d:%d\n", hour, minute);
```

main 函数的变量 h 和 print_time 函数的参数 hour 是两个不同的变量，只不过它们的存储空间中都保存了相同的值 23，因为变量 h 的值赋给了参数 hour。同理，变量 m 的值赋给了参数 minute。C 语言的这种传递参数的方式称为 Call by Value。在调用函数时，每个参数都需要得到一个值，函数定义中有几个形参，在调用时就要传几个实参，不能多也不能少，每个参数的类型也必须对应上。

肯定有读者注意到了，为什么我们每次调用 printf 传的实参个数都不一样呢？因为 C 语言规定了一种特殊的参数列表格式，用命令 man 3 printf 可以查看到 printf 函数的原型：

```
int printf(const char *format, ...);
```

第一个参数是 const char *类型的，后面的...可以代表 0 个或任意多个参数，这些参数的类型也是不确定的，这称为可变参数（Variable Argument），我们将在第 23.6 节详细讨论这种格式。总之，每个函数的原型都明确规定了返回值类型以及参数的类型和个数，即使像 printf 这样规定为“不确定”也是一种明确的规定，调用函数时要严格遵守这些规定，有时候我们把函数叫做接口（Interface），调

用函数就是使用这个接口，使用接口的前提是必须和接口保持一致。

提示：Man Page

Man Page 是 Linux 开发最常用的参考手册，由很多页面组成，每个页面描述一个主题，这些页面被组织成若干个 Section。FHS（Filesystem Hierarchy Standard）标准规定了 Man Page 各 Section 的含义如表 3.1 所示。

表 3.1　Man Page 的 Section

Section	描述
1	用户命令，例如 ls(1)
2	系统调用，例如_exit(2)
3	库函数，例如 printf(3)
4	特殊文件，例如 null(4)描述了设备文件/dev/null 和/dev/zero 的作用，这个页面也叫 zero(4)
5	系统配置文件的格式，例如 passwd(5)描述了系统配置文件/etc/passwd 的格式
6	游戏
7	其他杂项，例如 bash-builtins(7)描述了 bash 的各种内建命令
8	系统管理命令，例如 ifconfig(8)

注意区分用户命令和系统管理命令，用户命令通常位于/bin 和/usr/bin 目录，系统管理命令通常位于/sbin 和/usr/sbin 目录，一般用户可以执行用户命令，而执行系统管理命令经常需要 root 权限。系统调用和库函数的区别将在第 18.2 节说明。

Man Page 中有些页面有重名，比如敲 man printf 命令看到的并不是 C 函数 printf，而是位于第 1 个 Section 的命令 printf，要查看位于第 3 个 Section 的 printf 函数应该敲 man 3 printf，也可以敲 man -k printf 命令搜索哪些页面的主题包含 printf 关键字。本书会经常出现类似 printf(3)这样的写法，括号中的 3 表示 Man Page 的第 3 个 Section，或者表示“我这里想说的是 printf 库函数而不是 printf 命令”。

习题

1. 定义一个函数 increment，它的作用是把传进来的参数加 1。例如：

```
void increment(int x)
{
        x = x + 1;
}

int main(void)
{
        int i = 1, j = 2;
        increment(i); /* i now becomes 2 */
```

```
        increment(j); /* j now becomes 3 */
        return 0;
}
```

我们在 main 函数中调用 increment 增加变量 i 和 j 的值，这样能奏效吗？为什么？

2．说出以下代码哪些属于函数声明，哪些属于函数定义，哪些属于函数原型。

- main() {}
- int foo();
- int bar(void) {}
- void baz(int i, int);

3.4 全局变量、局部变量和作用域

我们把函数中定义的变量称为局部变量（Local Variable），由于形参相当于函数中定义的变量，所以形参也是一种局部变量。在这里“局部”有两层含义：

1．一个函数中定义的变量不能被另一个函数使用。例如 print_time 中的 hour 和 minute 在 main 函数中没有定义，不能使用，同样 main 函数中的局部变量也不能被 print_time 函数使用。如果这样定义：

```
void print_time(int hour, int minute)
{
        printf("%d:%d\n", hour, minute);
}

int main(void)
{
        int hour = 23, minute = 59;
        print_time(hour, minute);
        return 0;
}
```

main 函数中定义了局部变量 hour，print_time 函数中也有参数 hour，虽然它们名称相同，但仍然是两个不同的变量，代表不同的存储单元。main 函数的局部变量 minute 和 print_time 函数的参数 minute 也是如此。

2．每次调用函数时局部变量都表示不同的存储空间。局部变量在每次函数调用时分配存储空间，在每次函数返回时释放存储空间，例如调用 print_time(23, 59)时分配 hour 和 minute 两个变量的存储空间，在里面分别存上 23 和 59，函数返回时释放它们的存储空间，下次再调用 print_time(12, 20)时又分配 hour 和 minute 的存储空间，在里面分别存上 12 和 20。

与局部变量的概念相对的是全局变量（Global Variable），全局变量定义在所有的函数体之外，它们在程序开始运行时分配存储空间，在程序结束时释放存储空间，在任何函数中都可以访问全局变量。

例 3.5 全局变量

```
#include <stdio.h>

int hour = 23, minute = 59;

void print_time(void)
{
        printf("%d:%d in print_time\n", hour, minute);
}

int main(void)
{
        print_time();
        printf("%d:%d in main\n", hour, minute);
        return 0;
}
```

正因为全局变量在任何函数中都可以访问，所以在程序运行过程中全局变量被读写的顺序从源代码中是看不出来的，源代码的书写顺序并不能反映函数的调用顺序。程序出现了 Bug 往往就是因为在某个不起眼的地方对全局变量的读写顺序不正确，如果代码规模很大，这种错误是很难找到的。而对局部变量的访问不仅局限在一个函数内部，而且局限在一次函数调用之中，从函数的源代码中很容易看出访问的先后顺序是怎样的，所以比较容易找到 Bug。因此，**虽然全局变量用起来很方便，但一定要慎用，能用函数传参代替的就不要用全局变量**。

如果全局变量和局部变量重名了会怎么样呢？如果上面的例子改为：

例 3.6 作用域

```
#include <stdio.h>

int hour = 23, minute = 59;
int x = 10;

void print_time(void)
{
        printf("%d:%d in print_time\n", hour, minute);
}

int main(void)
{
        int hour = 0, minute = 30;
        print_time();
        printf("%d:%d in main\n", hour, minute);
        printf("x=%d\n", x);
        return 0;
}
```

则第一次调用 print_time 打印的是全局变量的值，第二次直接调用 printf 打印的则是 main 函数局部变量的值。在 C 语言中每个标识符都有特定的作用域，全局变量是定义在所有函数体之外的标识符，它的作用域从定义的位置开始直到源文件结束，而 main 函数局部变量的作用域仅限于 main 函数之中。如例 3.6 中方框所示，设想整个源文件是一张大纸，也就是全局变量的作用域，而 main 函数是盖在这张大纸上的一张小纸，也就是 main 函数局部变量的作用域。在小纸上用到标识

符 hour 和 minute 时应该参考小纸上的定义，因为大纸（全局变量的作用域）被盖住了，如果在小纸上用到某个标识符却没有找到它的定义，那么再去翻看下面的大纸上有没有定义，例如例 3.6 中的变量 x。

到目前为止我们在初始化一个变量时都是用常量做 Initializer，其实也可以用表达式做 Initializer，但要注意一点：**局部变量可以用类型相符的任意表达式来初始化，而全局变量只能用常量表达式（Constant Expression）来初始化**。例如，全局变量 pi 这样初始化是合法的：

```
double pi = 3.14 + 0.0016;
```

但这样初始化是不合法的：

```
double pi = acos(-1.0);
```

然而局部变量这样初始化却是合法的。

为什么要这样规定呢？因为在程序运行一开始（在还没有执行 main 函数中的任何语句之前）就要用初始值来初始化全局变量，这样，main 函数的第一条语句就可以取全局变量的初始值来做计算。要做到这一点，初始值必须保存在编译生成的可执行文件中，因此要求初始值必须在**编译时**就计算出来，然而上面第二种 Initializer 的值必须在程序**运行时**调用 acos 函数才能得到，所以不能用来初始化全局变量。请注意区分编译时和运行时这两个概念。

由于编译器负责计算全局变量的初始值，为了简化编译器的实现，C 语言从语法上规定全局变量只能用常量表达式来初始化。比如有这样的初始化：

```
int minute = 360 - 10;
int hour = minute / 60;
```

把 minute 初始化成 360-10 是合法的，编译器并不会生成一个系统指令来描述 360-10 的计算过程并把这些指令保存到可执行文件中，而是直接把计算结果 350 保存到可执行文件中。

然而，把 hour 初始化成 minute / 60 是不合法的。虽然在编译时计算出 hour 的初始值是可能的（先算出 minute 的初始值再据此算出 hour 的初始值），但 minute / 60 不是常量表达式，不符合语法规定，所以编译器直接报错退出，而不去算这个初始值。

如果全局变量在定义时不初始化则初始值是 0，如果局部变量在定义时不初始化则初始值是不确定的。所以，**局部变量在使用之前一定要先赋值**，如果基于一个不确定的值做后续计算肯定会引入 Bug。

如何证明“局部变量的存储空间在每次函数调用时分配，在函数返回时释放”？当我们想要确认某些语法规则时，可以查教材，也可以查 C99，但最快捷的办法就是编个小程序验证一下。

例 3.7 验证局部变量存储空间的分配和释放

```
#include <stdio.h>

void foo(void)
{
        int i;
        printf("%d\n", i);
        i = 777;
}

int main(void)
{
        foo();
        foo();
        return 0;
}
```

第一次调用 foo 函数，分配变量 i 的存储空间，然后打印 i 的值，由于 i 未初始化，打印的应该是一个不确定的值，然后把 i 赋值为 777，函数返回，释放 i 的存储空间。第二次调用 foo 函数，分配变量 i 的存储空间，然后打印 i 的值，由于 i 未初始化，打印的应该又是一个不确定的值，如果确实如此，就证明了“局部变量的存储空间在每次函数调用时分配，在函数返回时释放”。分析完了，我们运行程序看看是不是像我们分析的这样：

```
$ ./a.out
-1077716184
777
$ ./a.out
-1077466584
777
```

结果出乎意料，第一次调用打印的 i 值确实是个不确定值，第二次调用打印的 i 值正是第一次调用末尾赋给 i 的值 777。

如何分析这个结果呢？有一种初学者是这样，原本就没有把这条语法规则记牢，或者对自己的记忆力没信心，看到这个结果就会想：哦那肯定是我记错了，改过来记吧，应该是“函数中的局部变量具有一直存在的固定的存储空间，每次函数调用时使用它，返回时也不释放，再次调用函数时它应该还能保持上次的值”。还有一种初学者是怀疑论者或不可知论者，看到这个结果就会想：教材上明明说“局部变量的存储空间在每次函数调用时分配，在函数返回时释放”，那一定是教材写错了，教材也是人写的，是人写的就难免出错，哦，连 C99 也这么写的啊，C99 也是人写的，也难免出错，或者 C99 也许没错，但是反正运行结果就是错了，计算机这东西真靠不住，太容易受电磁干扰和宇宙射线影响了，我的程序写得再正确也有可能被干扰得不能正确运行。

这是初学者最常见的两种心态。不从客观事实和逻辑推理出发分析问题的真正原因，而仅凭主观臆断胡乱给问题定性，“说你有罪你就有罪”。先不要胡乱怀疑，我们再做一次实验，在两次 foo 函数调用之间插一个别的函数调用，结果就大不

相同了[②]：

```
int main(void)
{
        foo();
        printf("hello\n");
        foo();
        return 0;
}
```

结果是：

```
$ ./a.out
134513801
hello
4009972
$ ./a.out
134513801
hello
8503284
```

这一回，两次调用 foo 打印的 i 值看起来都挺乱，但似乎第一次调用打印的 i 值总是不变，那它到底是个确定值还是不确定值呢？

关键的一点：我说“未初始化的局部变量的初值是不确定值”，并没有说每次运行程序时这个不确定值不能相同，也没有说这个不确定值不能是上次调用函数时赋给该局部变量的值。在这里“不确定”的准确含义是：**每次调用这个函数时该局部变量的初值可能不一样，运行环境不同，函数的调用次序不同，都会影响到局部变量的初值**。在运用逻辑推理时一定要注意，**不要把必要条件（Necessary Condition）当充分条件（Sufficient Condition）**，这一点在 Debug 时尤其重要，看到错误现象不要轻易断定原因是什么，一定要考虑再三，找出它的真正原因。例如，不要看到第二次调用打印 777 就下结论“函数中的局部变量具有一直存在的固定的存储空间，每次函数调用时使用它，返回时也不释放，再次调用函数时它应该还能保持上次的值”，这个结论倒是能推出 777 这个结果，但反过来由 777 这个结果却不能推出这样的结论。所以说 777 这个结果是该结论的必要条件，但不是充分条件。至于为什么这个不确定值有时刚好是 777，有时又不是，等学到第 18.1 节就能解释这些现象了。

从第 3.2 节介绍的语法规则可以看出，非定义的函数声明也可以写在局部作用域中，例如：

```
int main(void)
{
        void print_time(int, int);
        print_time(23, 59);
        return 0;
}
```

② 也许在你的机器上跑不出这个结果，因为你的编译器、操作系统、库函数的实现和我所用的不同，不过道理是类似的。

这样声明的标识符 print_time 具有局部作用域，只在 main 函数中是有效的函数名，出了 main 函数就不存在 print_time 这个标识符了。

写非定义的函数声明时参数可以只写类型而不起名，例如上面代码中的 void print_time(int, int);，只要告诉编译器参数类型是什么，编译器就能为 print_time(23, 59)函数调用生成正确的指令。另外注意，虽然在一个函数体中可以声明另一个函数，但不能定义另一个函数，C 语言不允许嵌套定义函数[③]。

③ 但 gcc 的扩展特性允许嵌套定义函数，本书不做详细讨论。

第 4 章

分支语句

4.1 if 语句

目前我们写的简单函数中可以有多条语句，但这些语句总是从前到后顺序执行的。除了顺序执行之外，有时候我们需要检查一个条件，然后根据检查的结果执行不同的后续代码，在 C 语言中可以用分支语句实现，比如：

```
if (x != 0) {
        printf("x is nonzero.\n");
}
```

其中 x != 0 表示“x 不等于 0”的条件，这个表达式称为控制表达式（Controlling Expression）如果条件成立，则{}中的语句被执行，否则{}中的语句不执行，直接跳到}后面。if 和控制表达式改变了程序的控制流程（Control Flow），不再是从前到后顺序执行，而是根据不同的条件执行不同的语句，这种控制流程称为分支（Branch）。上例中的!=号表示“不等于”，像这样的运算符如表 4.1 所示。

表 4.1 关系运算符和相等性运算符

运算符	含义
==	等于
!=	不等于
>	大于
<	小于
>=	大于或等于
<=	小于或等于

注意以下几点：

1．这里的==表示数学中的相等关系，相当于数学中的=号，初学者常犯的错误是在控制表达式中把==写成=，在 C 语言中=号是赋值运算符，两者的含义完全不同。

2．如果表达式所表示的比较关系成立则值为真（True），否则为假（False），在 C 语言中分别用 int 型的 1 和 0 表示。如果变量 x 的值是-1，那么 x>0 这个表达式的值为 0，x>-2 这个表达式的值为 1。

3．在数学中 a<b<c 表示 b 既大于 a 又小于 c，但作为 C 语言表达式却不是这样。以上几种运算符都是左结合的，请读者想一下这个表达式应如何求值。

4．这些运算符的两个操作数应该是相同类型的，两边都是整型或者都是浮点型可以做比较，但两个字符串不能做比较，在第 24.1.5 节我们会介绍比较字符串的方法。

5．==和!=称为相等性运算符（Equality Operator），其余四个称为关系运算符（Relational Operator），相等性运算符的优先级低于关系运算符。

总结一下，if (x != 0) { ... }这个语句的计算顺序是：首先求 x != 0 这个表达式的值，如果值为 0，就跳过{}中的语句直接执行后面的语句，如果值为 1，就先执行{}中的语句，然后再执行后面的语句。事实上控制表达式取任何非 0 值都表示真值，例如 if (x) { ... }和 if (x != 0) { ... }是等价的，如果 x 的值是 2，则 x != 0 的值是 1，但对于 if 来说不管是 2 还是 1 都表示真值。

和 if 语句相关的语法规则如下：

语句 → if (控制表达式) 语句

语句 → { 语句列表 }

语句 → ;

在 C 语言中，任何允许出现语句的地方既可以是由;号结尾的一条语句，也可以是由{}括起来的若干条语句或声明组成的语句块（Statement Block），语句块和上一章介绍的函数体的语法相同。注意语句块的}后面不需要加;号。如果}后面加了;号，则这个;号本身又是一条新的语句了，在 C 语言中一个单独的;号表示一条空语句（Null Statement）。上例的语句块中只有一条语句，其实没必要写成语句块，可以简单地写成：

```
if (x != 0)
        printf("x is nonzero.\n");
```

语句块中也可以定义局部变量，例如：

```
void foo(void)
{
    int i = 0;
    {
            int i = 1;
            int j = 2;
            printf("i=%d, j=%d\n", i, j);
    }
    printf("i=%d\n", i); /* cannot access j here */
}
```

和函数的局部变量同样道理，每次进入语句块时为变量 j 分配存储空间，每次退出语句块时释放变量 j 的存储空间。语句块也构成一个作用域，和例 3.6 的分析

类似，如果整个源文件是一张大纸，foo 函数是盖在上面的一张小纸，则函数中的语句块是盖在小纸上面的一张更小的纸。语句块中的变量 i 和函数的局部变量 i 是两个不同的变量，因此两次打印的 i 值是不同的；语句块中的变量 j 在退出语句块之后就没有了，因此最后一行的 printf 不能打印变量 j，否则编译器会报错。语句块可以用在任何允许出现语句的地方，不一定非得用在 if 语句中，单独使用语句块通常是为了定义一些比函数的局部变量更“局部”的变量。

习题

1．以下程序段编译能通过，执行也不出错，但是执行结果不正确（根据第 1.3 节的定义，这是一个语义错误），请分析一下哪里错了。还有，既然错了为什么编译能通过呢？

```
int x = -1;
if (x > 0);
        printf("x is positive.\n");
```

4.2 if/else 语句

if 语句还可以带一个 else 子句（Clause），例如：

```
if (x % 2 == 0)
        printf("x is even.\n");
else
        printf("x is odd.\n");
```

这里的%是取模（Modulo）运算符，x%2 表示 x 除以 2 所得的余数（Remainder），C 语言规定%运算符的两个操作数必须是整型的。两个正数相除取余数很好理解，如果操作数中有负数，结果应该是正是负呢？C99 规定，如果 a 和 b 是整型，b 不等于 0，则表达式(a/b)*b+a%b 的值总是等于 a，再结合第 2.5 节讲过的整数除法运算要 Truncate towards Zero，可以得到一个结论：**%运算符的结果总是与被除数同号**（想一想为什么）。其他编程语言对取模运算的规定各不相同，也有规定结果和除数同号的，也有不做明确规定的。

取模运算在程序中是非常有用的，例如上面的例子判断 x 的奇偶性（Parity），看 x 除以 2 的余数是不是 0，如果是 0 则打印 x is even.，如果不是 0 则打印 x is odd.，读者应该能看出 else 在这里的作用了，如果在上面的例子中去掉 else，则不管 x 是奇是偶，printf("x is odd.\n");总是执行。为了让这条语句更有用，可以把它封装成一个函数：

```
void print_parity(int x)
{
        if (x % 2 == 0)
                printf("x is even.\n");
        else
                printf("x is odd.\n");
}
```

把语句封装成函数的基本步骤是：**把语句放到函数体中，把变量改成函数的参数。**这样，以后要检查一个数的奇偶性只需调用这个函数而不必重复写这条语句了，例如：

```
print_parity(17);
print_parity(18);
```

if/else 语句的语法规则如下：

语句 → if (控制表达式) 语句 else 语句

右边的“语句”既可以是一条语句，也可以是由{}括起来的语句块。一条 if 语句中包含一条子语句，一条 if/else 语句中包含两条子语句，子语句可以是任何语句或语句块，当然也可以是另外一条 if 或 if/else 语句。根据组合规则，if 或 if/else 可以嵌套使用。例如可以这样：

```
if (x > 0)
        printf("x is positive.\n");
else if (x < 0)
      printf("x is negative.\n");
else
      printf("x is zero.\n");
```

也可以这样：

```
if (x > 0) {
       printf("x is positive.\n");
} else {
       if (x < 0)
              printf("x is negative.\n");
       else
              printf("x is zero.\n");
}
```

现在有一个问题，类似 if (A) if (B) C; else D;形式的语句怎么理解呢？可以理解成：

```
if (A)
        if (B)
                C;
else
        D;
```

也可以理解成：

```
if (A)
        if (B)
                C;
        else
                D;
```

在第 2.1 节讲过，C 代码的缩进只是为了程序员看起来方便，实际上对编译器不起任何作用，你的代码不管写成上面哪一种缩进格式，在编译器看起来都是一样的。那么编译器到底按哪种方式理解呢？也就是说，else 到底是和 if (A)配对还是

和 if (B)配对？很多编程语言的语法都有这个问题，称为 Dangling-else 问题。C 语言规定，**else 总是和它上面最近的一个 if 配对**，因此应该理解成 else 和 if (B) 配对，也就是按第二种方式理解。如果你写成上面第一种缩进的格式就很危险了：你看到的是这样，而编译器理解的却是那样。如果你希望编译器按第一种方式理解，应该明确加上{}：

```
if (A) {
        if (B)
                C;
} else
                D;
```

顺便提一下，浮点型的精度有限，不适合用==运算符做精确比较。以下代码可以说明问题：

```
double i = 20.0;
double j = i / 7.0;
if (j * 7.0 == i)
        printf("Equal.\n");
else
        printf("Unequal.\n");
```

不同平台的浮点数实现有很多不同之处，在我的平台上运行这段程序结果为 Unequal，即使在你的平台上运行结果为 Equal，你再把 i 改成其他值试试，总有些值会使得结果为 Unequal。等学习了第 13.4 节你就知道为什么浮点型不能做精确比较了。

习题

1. 写两个表达式，分别取整型变量 x 的个位和十位。

2. 写一个函数，参数是整型变量 x，功能是打印 x 的个位和十位。

4.3 布尔代数

在第 4.1 节讲过，a<b<c 不表示 b 既大于 a 又小于 c，那么如果想表示这个含义该怎么写呢？可以这样：

```
if (a < b) {
        if (b < c) {
                printf("b is between a and c.\n");
        }
}
```

我们也可以用逻辑与（Logical AND）运算符表示这两个条件同时成立。逻辑与运算符在 C 语言中写成两个&（Ampersand），上面的语句可以改写为：

```
if (a < b && b < c) {
        printf("b is between a and c.\n");
}
```

对于 a < b && b < c 这个控制表达式，要求“a < b 的值非 0”和“b < c 的值非 0”这两个条件同时成立整个表达式的值才为 1，否则整个表达式的值为 0。也就是只有两个条件都为真，它们做逻辑与运算的结果才为真，有一个条件为假，则逻辑与运算的结果为假，如表 4.2 所示。

表 4.2 AND 的真值表

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

这种表称为真值表（Truth Table）。注意逻辑与运算的操作数以非 0 表示真以 0 表示假，而运算结果以 1 表示真以 0 表示假（类型是 int），我们忽略这些细微的差别，在表中全部以 1 表示真以 0 表示假。C 语言还提供了逻辑或（Logical OR）运算符，写成两个|线（Pipe Sign），逻辑非（Logical NOT）运算符，写成一个!号（Exclamation Mark），它们的真值表如表 4.3 及表 4.4 所示。

表 4.3 OR 的真值表

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

表 4.4 NOT 的真值表

A	NOT A
0	1
1	0

逻辑或表示两个条件只要有一个为真，它们做逻辑或运算的结果就为真，只有两个条件都为假，逻辑或运算的结果才为假。逻辑非的作用是对原来的逻辑值取反，原来是真的就是假，原来是假的就是真，例如!(x > 1)，如果表达式 x > 1 的值非零，则!(x > 1)的值为 0。逻辑非运算符只有一个操作数，称为单目运算符（Unary Operator），以前讲过的加减乘除、赋值、相等性、关系、逻辑与、逻辑或运算符都有两个操作数，称为双目运算符（Binary Operator）。

关于逻辑运算的数学体系称为布尔代数（Boolean Algebra），以它的创始人布尔命名。在编程语言中表示真和假的数据类型叫做布尔类型，在 C 语言中通常用 int 型来表示，非 0 表示真，0 表示假[①]。布尔逻辑是写程序的基本功之一，程序中的很多错误都可以归因于逻辑错误。以下是一些布尔代数的基本定理，为了简洁易

① C99 也定义了专门的布尔类型_Bool，但目前没有被广泛使用。

读，真和假用1和0表示，AND用*号表示，OR用+号表示（从真值表可以看出AND和OR运算确实有点像乘法和加法运算），NOT用¬表示，变量x、y、z的值可能是0也可能是1。

```
¬¬x=x

x*0=0
x+1=1

x*1=x
x+0=x

x*x=x
x+x=x

x*¬x=0
x+¬x=1

x*y=y*x
x+y=y+x

x*(y*z)=(x*y)*z
x+(y+z)=(x+y)+z

x*(y+z)=x*y+x*z
x+y*z=(x+y)*(x+z)

x+x*y=x
x*(x+y)=x

x*y+x*¬y=x
(x+y)*(x+¬y)=x

¬(x*y)=¬x+¬y
¬(x+y)=¬x*¬y

x+¬x*y=x+y
x*(¬x+y)=x*y

x*y+¬x*z+y*z=x*y+¬x*z
(x+y)*(¬x+z)*(y+z)=(x+y)*(¬x+z)
```

除了第1行之外，这些公式都是每两行一组的，每组的两个公式就像对联一样：把其中一个公式中的*换成+、+换成*、0换成1、1换成0，就变成了与它对称的另一个公式。这些定理都可以通过真值表证明，更多细节可参考有关数字逻辑的教材，例如参考文献[9]。我们将在本节的练习题中强化训练对这些定理的理解。

目前为止介绍的这些运算符的优先级顺序是：!高于* / %，高于+ -，高于>><>= <=，高于== !=，高于&&，高于||，高于=。写一个控制表达式很可能同时用到这些运算符中的多个，如果记不清楚运算符的优先级一定要多套括号。我们将在第15.4节总结C语言所有运算符的优先级和结合性。

习题

1. 把代码段：

```
if (x > 0 && x < 10);
```

```
    else
            printf("x is out of range.\n");
```

改写成下面这种形式：

```
    if (____ || ____)
            printf("x is out of range.\n");
```

____应该怎么填？

2. 把代码段：

```
    if (x > 0)
            printf("Test OK!\n");
    else if (x <= 0 && y > 0)
            printf("Test OK!\n");
    else
            printf("Test failed!\n");
```

改写成下面这种形式：

```
    if (____ && ____)
            printf("Test failed!\n");
    else
            printf("Test OK!\n");
```

____应该怎么填？

3. 有这样一段代码：

```
    if (x > 1 && y != 1) {
            ...
    } else if (x < 1 && y != 1) {
            ...
    } else {
            ...
    }
```

要进入最后一个 else，x 和 y 需要满足条件____ || ____。这里应该怎么填？

4. 以下哪一个 if 判断条件是多余的可以去掉？这里所谓的“多余”是指，某种情况下如果本来应该打印 Test OK!，去掉这个多余条件后仍然打印 Test OK!，如果本来应该打印 Test failed!，去掉这个多余条件后仍然打印 Test failed!。

```
    if (x<3 && y>3)
            printf("Test OK!\n");
    else if (x>=3 && y>=3)
            printf("Test OK!\n");
    else if (z>3 && x>=3)
            printf("Test OK!\n");
    else if (z<=3 && y>=3)
            printf("Test OK!\n");
    else
            printf("Test failed!\n");
```

5. 以下两段代码是否等价？

```
if (A && B)
        statement1;
else
        statement2;
if (A) {
    if (B)
            statement1;
} else
        statement2;
```

4.4 switch 语句

switch 语句可以产生具有多个分支的控制流程。它的格式是：

```
switch (控制表达式) {
case 常量表达式： 零或多条语句
case 常量表达式： 零或多条语句
...
default： 零或多条语句
}
```

例如以下程序根据传入的参数 1～7 分别打印 Monday～Sunday：

例 4.1 switch 语句

```
#include <stdio.h>

void print_day(int day)
{
    switch (day) {
    case 1:
        printf("Monday\n");
        break;
    case 2:
        printf("Tuesday\n");
        break;
    case 3:
        printf("Wednesday\n");
        break;
    case 4:
        printf("Thursday\n");
        break;
    case 5:
        printf("Friday\n");
        break;
    case 6:
        printf("Saturday\n");
        break;
    case 7:
        printf("Sunday\n");
        break;
    default:
        printf("Illegal day number!\n");
        break;
    }
}

int main(void)
{
    print_day(2);
    return 0;
}
```

如果传入的参数是2，则从case 2分支开始执行，先是打印相应的信息，然后遇到break语句，它的作用是跳出整个switch语句块。C语言规定各case分支的常量表达式必须互不相同，如果控制表达式不等于任何一个常量表达式，则从default分支开始执行，通常把default分支写在最后，但不是必需的。使用switch语句要注意以下几点：

1. case后面跟的表达式必须是常量表达式，这个值和全局变量的初始值一样必须在编译时计算出来。

2. 第4.2节讲过浮点型不适合做精确比较，所以C语言规定case后面跟的必须是整型常量表达式。

3. 进入case后如果没有遇到break语句就会一直往下执行（这称为Fall Through），后面其他case或default分支的语句也会被执行到，直到遇到break，或者执行到整个switch语句块的末尾。通常每个case后面都要加上break语句，但有时会故意不加break，例如：

例4.2 缺break的switch语句

```
#include <stdio.h>

void print_day(int day)
{
    switch (day) {
    case 1:
    case 2:
    case 3:
    case 4:
    case 5:
        printf("Weekday\n");
        break;
    case 6:
    case 7:
        printf("Weekend\n");
        break;
    default:
        printf("Illegal day number!\n");
        break;
    }
}

int main(void)
{
    print_day(2);
    return 0;
}
```

switch语句不是必不可缺的，显然可以用一组if ... else if ... else if ... else ...代替，但是一方面用switch语句会使代码更清晰，另一方面，有时候编译器会对switch语句做整体优化，使它比等价的if/else语句所生成的指令效率更高。

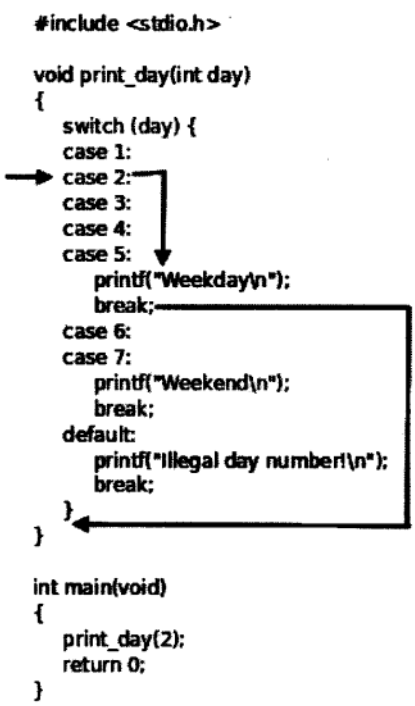
如果传入的参数是2，则从case 2分支开始执行，先是打印相应的信息，然后遇到break语句，它的作用是跳出整个switch语句块。C语言规定各case分支的常量表达式必须互不相同，如果控制表达式不等于任何一个常量表达式，则从default分支开始执行，通常把default分支写在最后，但不是必需的。使用switch语句要注意以下几点：

1. case后面跟的表达式必须是常量表达式，这个值和全局变量的初始值一样必须在编译时计算出来。

2. 第4.2节讲过浮点型不适合做精确比较，所以C语言规定case后面跟的必须是整型常量表达式。

3. 进入case后如果没有遇到break语句就会一直往下执行（这称为Fall Through），后面其他case或default分支的语句也会被执行到，直到遇到break，或者执行到整个switch语句块的末尾。通常每个case后面都要加上break语句，但有时会故意不加break，例如：

例4.2 缺break的switch语句



```
#include <stdio.h>

void print_day(int day)
{
    switch (day) {
    case 1:
    case 2:
    case 3:
    case 4:
    case 5:
        printf("Weekday\n");
        break;
    case 6:
    case 7:
        printf("Weekend\n");
        break;
    default:
        printf("Illegal day number!\n");
        break;
    }
}

int main(void)
{
    print_day(2);
    return 0;
}
```

switch语句不是必不可缺的，显然可以用一组if ... else if ... else if ... else ...代替，但是一方面用switch语句会使代码更清晰，另一方面，有时候编译器会对switch语句做整体优化，使它比等价的if/else语句所生成的指令效率更高。

先级，在这里不起任何作用。我们可以这样调用这个函数：

```
int i = 19;
if (is_even(i)) {
        /* do something */
} else {
        /* do some other thing */
}
```

返回布尔值的函数是一类非常有用的函数，在程序中通常充当控制表达式，函数名通常带有 is 或 if 等表示判断的词，这类函数也叫做谓词（Predicate）。is_even 这个函数写得有点啰嗦，x % 2 这个表达式本来就有 0 值或非 0 值，可以直接把它看作布尔值：

```
int is_even(int x)
{
        return !(x % 2);
}
```

函数的返回值应该这样理解：**函数返回一个值相当于定义一个和返回值类型相同的临时变量并用 return 后面的表达式来初始化。**例如上面的函数调用相当于这样的过程：

```
int 临时变量 = !(x % 2);
函数退出，局部变量 x 的存储空间释放;
if (临时变量) { /* 临时变量用完就释放 */
    /* do something */
} else {
    /* do some other thing */
}
```

当 if 语句对函数的返回值做判断时，函数已经退出，局部变量 x 已经释放，所以不可能在这时候才计算表达式!(x % 2)的值，表达式的值必然是事先计算好了保存在一个没有名字的临时变量里的，然后函数退出，局部变量释放，if 语句对这个临时变量的值做判断。注意，虽然函数的返回值可以看作是一个临时变量，但我们只是读一下它的值，读完值就释放它，而不能往它里面存新的值，换句话说，**函数的返回值不是左值，或者说函数调用表达式不能做左值**，因此下面的赋值语句是非法的：

```
is_even(20) = 1;
```

在第 3.3 节中讲过，C 语言的传参规则是 Call by Value，按值传递，现在我们知道返回值也是按值传递的，即便返回语句写成 return x;，返回的也是变量 x 的值，而非变量 x 本身，因为变量 x 马上就要被释放了。

在写带有 return 语句的函数时要小心检查所有的代码路径（Code Path）。有些代码路径在任何条件下都执行不到，这称为 Dead Code，例如把&&和||运算符记混了（据我了解初学者犯这个低级错误的不在少数），写出如下代码：

```
void foo(int x, int y)
```

```
{
        if (x >= 0 || y >= 0) {
                printf("both x and y are positive.\n");
                return;
        } else if (x < 0 || y < 0) {
                printf("both x and y are negative.\n");
                return;
        }
        printf("x has a different sign from y.\n");
}
```

最后一行printf永远都没机会被执行到，是一行Dead Code。有Dead Code就一定有Bug，你写的每一行代码都是想让程序在某种情况下去执行的，你不可能故意写出一行永远不会被执行的代码，如果程序在任何情况下都不会去执行它，说明跟你预想的不一样，要么是你对所有可能的情况分析得不正确，要么就是像上例这样的笔误，这些属于逻辑错误和语义错误。还有一些时候，对程序中所有可能的情况分析得不够全面将导致漏掉一些代码路径，例如：

```
int absolute_value(int x)
{
        if (x < 0) {
                return -x;
        } else if (x > 0) {
                return x;
        }
}
```

这个函数被定义为返回int，就应该在任何情况下都返回int，但是上面这个程序在x==0时安静地退出函数，什么也不返回，C标准对于这种情况会返回什么结果是未定义的，通常返回不确定的值，等学到第18.1节你就知道为什么了。另外注意这个例子中把-号当负号用而不是当减号用，事实上+号也可以这么用。正负号是单目运算符，而加减号是双目运算符，正负号的优先级和逻辑非运算符相同，比加减的优先级要高。

以上两段代码都不会产生编译错误，编译器只做语法检查和最简单的语义检查，而不检查程序的逻辑[①]。虽然到现在为止你见到了各种各样的编译器错误提示，也许你已经十分讨厌编译器报错了，但很快你就会认识到，如果程序中有错误编译器还不报错，那一定比报错更糟糕。比如上面的绝对值函数，在你测试的时候运行得很好，也许是你没有测到x==0的情况，也许刚好在你的环境中x==0时返回的不确定值就是0，然后你放心地把它集成到一个数万行的程序之中。然后你把这个程序交给用户，起初的几天里相安无事，之后每过几个星期就有用户报告说程序出错，但每次出错的现象都不一样，而且这个错误很难复现，你想让它出现时它就不出现，在你毫无防备时它又突然冒出来了。然后你花了大量的时间在数万行的程序中排查哪里错了，几天之后终于幸运地找到了这个函数的Bug，这时候你就会想，如果当初编译器能报个错多好啊！所以，如果编译器报错了，不

① 有的代码路径没有返回值的问题编译器是可以检查出来的，如果编译时加-Wall选项会报警告。

要责怪编译器太过于挑剔，它帮你节省了大量的调试时间。另外，在 math.h 中有一个 fabs 函数就是求绝对值的，我们通常不必自己写绝对值函数。

习题

1．编写一个布尔函数 int is_leap_year(int year)，判断参数 year 是不是闰年。如果某年份能被 4 整除，但不能被 100 整除，那么这一年就是闰年，此外，能被 400 整除的年份也是闰年。

2．编写一个函数 double myround(double x)，输入一个小数，将它四舍五入。例如 myround(-3.51)的值是-4.0，myround(4.49)的值是 4.0。可以调用 math.h 中的库函数 ceil 和 floor 实现这个函数，代码要尽可能简洁高效。

5.2 增量式开发

目前为止你看到了很多示例代码，也在它们的基础上做了很多改动并在这个过程中巩固所学的知识。但是如果从头开始编写一个程序解决某个问题，应该按什么步骤来写呢？本节提出一种增量式（Incremental）开发的思路，很适合初学者。

现在问题来了：我们要编一个程序求圆的面积，圆的半径以两个端点的坐标(x_1,y_1)和(x_2,y_2)给出。首先分析和分解问题，把大问题分解成小问题，再对小问题分别求解。这个问题可分为两步：

1．由两个端点坐标求半径的长度，我们知道平面上两点间距离的公式是：

$$\text{distance} = \sqrt{((x_2 - x_1)^2 + (y_2 - y_1)^2)}$$

括号里的部分都可以用我们学过的 C 语言表达式来表示，求平方根可以用 math.h 中的 sqrt 函数，因此这个小问题全部都可以用我们学过的知识解决。这个公式可以实现成一个函数，参数是两点的坐标，返回值是 distance。

2．上一步算出的距离是圆的半径，已知圆的半径之后求面积的公式是：

$$\text{area} = \pi \cdot \text{radius}^2$$

也可以用我们学过的 C 语言表达式来解决，这个公式也可以实现成一个函数，参数是 radius，返回值是 area。

首先编写 distance 这个函数，我们已经明确了它的参数是两点的坐标，返回值是两点间距离，可以先写一个简单的函数定义：

```
double distance(double x1, double y1, double x2, double y2)
{
        return 0.0;
}
```

初学者写到这里就已经不太自信了：这个函数定义写得对吗？虽然我是按我理解的语法规则写的，但书上没有和这个一模一样的例子，万一不小心遗漏了什么呢？既然不自信就不要再往下写了，没有一个平稳的心态来写程序很可能会引入Bug。所以在函数定义中插一个return 0.0;立刻结束掉它，然后立刻测试这个函数定义得有没有错：

```
int main(void)
{
        printf("distance is %f\n", distance(1.0, 2.0, 4.0, 6.0));
        return 0;
}
```

编译，运行，一切正常。这时你就会建立起信心了：既然没问题，就不用管它了，继续往下写。在测试时给这个函数的参数是(1.0, 2.0)和(4.0, 6.0)，两点的X坐标距离是3.0，Y坐标距离是4.0，因此两点间距离应该是5.0，你必须事先知道正确答案是5.0，这样你才能测试程序计算的结果对不对。当然，现在函数还没实现，计算结果肯定是不对的。现在我们再往函数里添一点代码：

```
double distance(double x1, double y1, double x2, double y2)
{
        double dx = x2 - x1;
        double dy = y2 - y1;
        printf("dx is %f\ndy is %f\n", dx, dy);

        return 0.0;
}
```

如果你不确定dx和dy这样初始化行不行，那么就此打住，在函数里插一条打印语句把dx和dy的值打出来看看。把它和上面的main函数一起编译运行，由于我们事先知道结果应该是3.0和4.0，因此能够验证程序算得对不对。一旦验证无误，函数里的这句打印就可以撤掉了，像这种打印语句，以及我们用来测试的main函数，都起到了类似脚手架（Scaffold）的作用：在盖房子时很有用，但它不是房子的一部分，房子盖好之后就可以拆掉了。房子盖好之后可能还需要维修、加盖、翻新，又要再加上脚手架，这很麻烦，要是当初不用拆就好了，可是不拆不行，不拆多难看啊。写代码却可以有一个更高明的解决办法：把Scaffolding的代码注释掉。

```
double distance(double x1, double y1, double x2, double y2)
{
        double dx = x2 - x1;
        double dy = y2 - y1;
        /* printf("dx is %f\ndy is %f\n", dx, dy); */
        return 0.0;
}
```

这样如果以后出了新的Bug又需要跟踪调试时，还可以把这句重新加进代码中使用。两点的X坐标距离和Y坐标距离都没问题了，下面求它们的平方和：

```
double distance(double x1, double y1, double x2, double y2)
{
```

```
        double dx = x2 - x1;
        double dy = y2 - y1;
        double dsquared = dx * dx + dy * dy;
        printf("dsquared is %f\n", dsquared);

        return 0.0;
}
```

然后再编译、运行，看看是不是得25.0。这样的增量式开发非常适合初学者，每写一行代码都编译运行，确保没问题了再写下一行，一方面在写代码时更有信心，另一方面也方便了调试：总是有一个先前的正确版本做参照，改动之后如果出了问题，几乎可以肯定就是刚才改的那行代码出的问题，这样就避免了必须从很多行代码中查找分析到底是哪一行出的问题。在这个过程中 printf 功不可没，你怀疑哪一行代码有问题，就插一个 printf 进去看看中间的计算结果，任何错误都可以通过这个办法找出来。以后我们会介绍程序调试工具 gdb，它提供了更强大的调试功能帮你分析更隐蔽的错误。但即使有了 gdb，printf 这个最原始的办法仍然是最直接、最有效的。最后一步，我们完成这个函数：

例 5.1 distance 函数

```
#include <math.h>
#include <stdio.h>

double distance(double x1, double y1, double x2, double y2)
{
        double dx = x2 - x1;
        double dy = y2 - y1;
        double dsquared = dx * dx + dy * dy;
        double result = sqrt(dsquared);

        return result;
}

int main(void)
{
        printf("distance is %f\n", distance(1.0, 2.0, 4.0, 6.0));
        return 0;
}
```

然后编译运行，看看是不是得5.0。随着编程经验越来越丰富，你可能每次写若干行代码再一起测试，而不是像现在这样每写一行就测试一次，但不管怎么样，增量式开发的思路是很有用的，它可以帮你节省大量的调试时间，不管你有多强，都不应该一口气写完整个程序再编译运行，那几乎是一定会有 Bug 的，到那时候再找 Bug 就难了。

这个程序中引入了很多临时变量：dx、dy、dsquared、result，如果你有信心把整个表达式一次性写好，也可以不用临时变量：

```
double distance(double x1, double y1, double x2, double y2)
{
        return sqrt((x2-x1) * (x2-x1) + (y2-y1) * (y2-y1));
}
```

这样写简洁得多了。但如果写错了呢？只知道是这一长串表达式有错，根本不知道错在哪，而且整个函数就一个语句，插 printf 都没地方插。所以用临时变量有它的好处，使程序更清晰，调试更方便，而且有时候可以避免不必要的计算，例如上面这一行表达式要把(x2-x1)计算两遍，如果算完(x2-x1)把结果存在一个临时变量 dx 里，就不需要再算第二遍了。

接下来编写 area 这个函数：

```
double area(double radius)
{
        return 3.1416 * radius * radius;
}
```

给出两点的坐标求距离，给出半径求圆的面积，这两个子问题都解决了，如何把它们组合起来解决整个问题呢？给出半径的两端点坐标(1.0, 2.0)和(4.0, 6.0)求圆的面积，先用 distance 函数求出半径的长度，再把这个长度传给 area 函数：

```
double radius = distance(1.0, 2.0, 4.0, 6.0);
double result = area(radius);
```

也可以这样：

```
double result = area(distance(1.0, 2.0, 4.0, 6.0));
```

我们一直把“给出半径的两端点坐标求圆的面积”这个问题当做整个问题来看，如果它也是一个更大的程序当中的子问题呢？我们可以把先前的两个函数组合起来做成一个新的函数以便日后使用：

```
double area_point(double x1, double y1, double x2, double y2)
{
        return area(distance(x1, y1, x2, y2));
}
```

还有另一种组合的思路，不是把 distance 和 area 两个函数调用组合起来，而是把那两个函数中的语句组合到一起：

```
double area_point(double x1, double y1, double x2, double y2)
{
        double dx = x2 - x1;
        double dy = y2 - y1;
        double radius = sqrt(dx * dx + dy * dy);

        return 3.1416 * radius * radius;
}
```

这样组合是不理想的。这样组合了之后，原来写的 distance 和 area 两个函数还要不要了呢？如果不要了删掉，那么如果有些情况只需要求两点间的距离，或者只需要给定半径长度求圆的面积呢？area_point 把所有语句都写在一起，太不灵活了，满足不了这样的需要。如果保留 distance 和 area 同时也保留这个 area_point 怎么样呢？area_point 和 distance 有相同的代码，一旦在 distance 函数中发现了

Bug，或者要升级 distance 这个函数采用更高的计算精度，那么不仅要修改 distance，还要记着修改 area_point，同理，要修改 area 也要记着修改 area_point，维护重复的代码是非常容易出错的，在任何时候都要尽量避免。因此，**尽可能复用（Reuse）以前写的代码，避免写重复的代码**。封装就是为了复用，把解决各种小问题的代码封装成函数，在解决第一个大问题时可以用这些函数，在解决第二个大问题时可以复用这些函数。

解决问题的过程是把大问题分解成小问题，小问题再分解成更小的问题，这个过程在代码中体现为函数的分层设计（Stratify）。distance 和 area 是两个底层函数，解决一些很小的问题，而 area_point 是一个上层函数，上层函数通过调用底层函数来解决更大的问题，底层和上层函数都可以被更上一层的函数调用，最终所有的函数都直接或间接地被 main 函数调用，如图 5.1 所示。

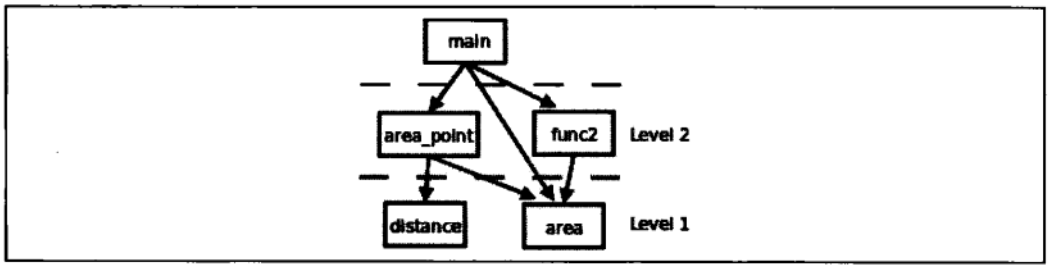


图 5.1 函数的分层设计

5.3 递归

如果定义一个概念需要用到这个概念本身，我们称它的定义是递归的(Recursive)。例如：

```
frabjuous
an adjective used to describe something that is frabjuous.
```

这只是一个玩笑，如果你在字典上看到这么一个词条肯定要怒了。然而数学上确实有很多概念是用它自己来定义的，比如 n 的阶乘（Factorial）是这样定义的：n 的阶乘等于 n 乘以 n-1 的阶乘。如果这样就算定义完了，恐怕跟上面那个词条有异曲同工之妙了：n-1 的阶乘是什么？是 n-1 乘以 n-2 的阶乘。那 n-2 的阶乘又是什么？这样下去永远也没完。因此需要定义一个最关键的基础条件（Base Case）：0 的阶乘等于 1。

```
0! = 1
n! = n · (n-1)!
```

因此，3!=3×2!，2!=2×1!，1!=1×0!=1×1=1，正因为有了 Base Case，才不会永远没完地数下去，知道了 1!=1 我们再反过来算回去，2!=2×1!=2×1=2，3!=3×2!=3×2=6。下面用程序来完成这一计算过程，我们要写一个计算阶乘的函数 factorial，先把 Base Case 这种最简单的情况写进去：

```
int factorial(int n)
```

```
{
        if (n == 0)
                return 1;
}
```

如果参数 n 不是 0 应该 return 什么呢？根据定义，应该 return n*factorial(n-1);，为了下面的分析方便，我们引入几个临时变量把这个语句拆分一下：

```
int factorial(int n)
{
        if (n == 0)
                return 1;
        else {
                int recurse = factorial(n-1);
                int result = n * recurse;
                return result;
        }
}
```

factorial 这个函数居然可以自己调用自己？是的。自己直接或间接调用自己的函数称为递归函数。这里的 factorial 是直接调用自己，有些时候函数 A 调用函数 B，函数 B 又调用函数 A，也就是函数 A 间接调用自己，这也是一种递归调用。如果你觉得迷惑，可以把 factorial(n-1)这一步看成是在调用另一个函数——另一个有着相同函数名和相同代码的函数，调用它就是跳到它的代码里执行，然后再返回 factorial(n-1)这个调用的下一步继续执行。我们以 factorial(3)为例分析整个调用过程，如图 5.2 所示。

图中用实线箭头表示调用，用虚线箭头表示返回，右侧的框表示在调用和返回过程中各层函数调用的存储空间变化情况。

1．main()有一个局部变量 result，用一个框表示。

2．调用 factorial(3)时要分配参数和局部变量的存储空间，于是在 main()的下面又多了一个框表示 factorial(3)的参数和局部变量，其中 n 已初始化为 3。

3．factorial(3)又调用 factorial(2)，又要分配 factorial(2)的参数和局部变量，于是在 main()和 factorial(3)下面又多了一个框。第 3.4 节讲过，每次调用函数时分配参数和局部变量的存储空间，退出函数时释放它们的存储空间。factorial(3)和 factorial(2)是两次不同的调用，factorial(3)的参数 n 和 factorial(2)的参数 n 各有各的存储单元，虽然我们写代码时只写了一次参数 n，但运行时却是两个不同的参数 n。并且由于调用 factorial(2)时 factorial(3)还没退出，所以两个函数调用的参数 n 同时存在，所以在原来的基础上多画一个框。

4．依此类推，请读者对照着图 5.2 自己分析整个调用过程。读者会发现这个过程和前面我们用数学公式计算 3!的过程是一样的，都是先一步步展开然后再一步步收回去。

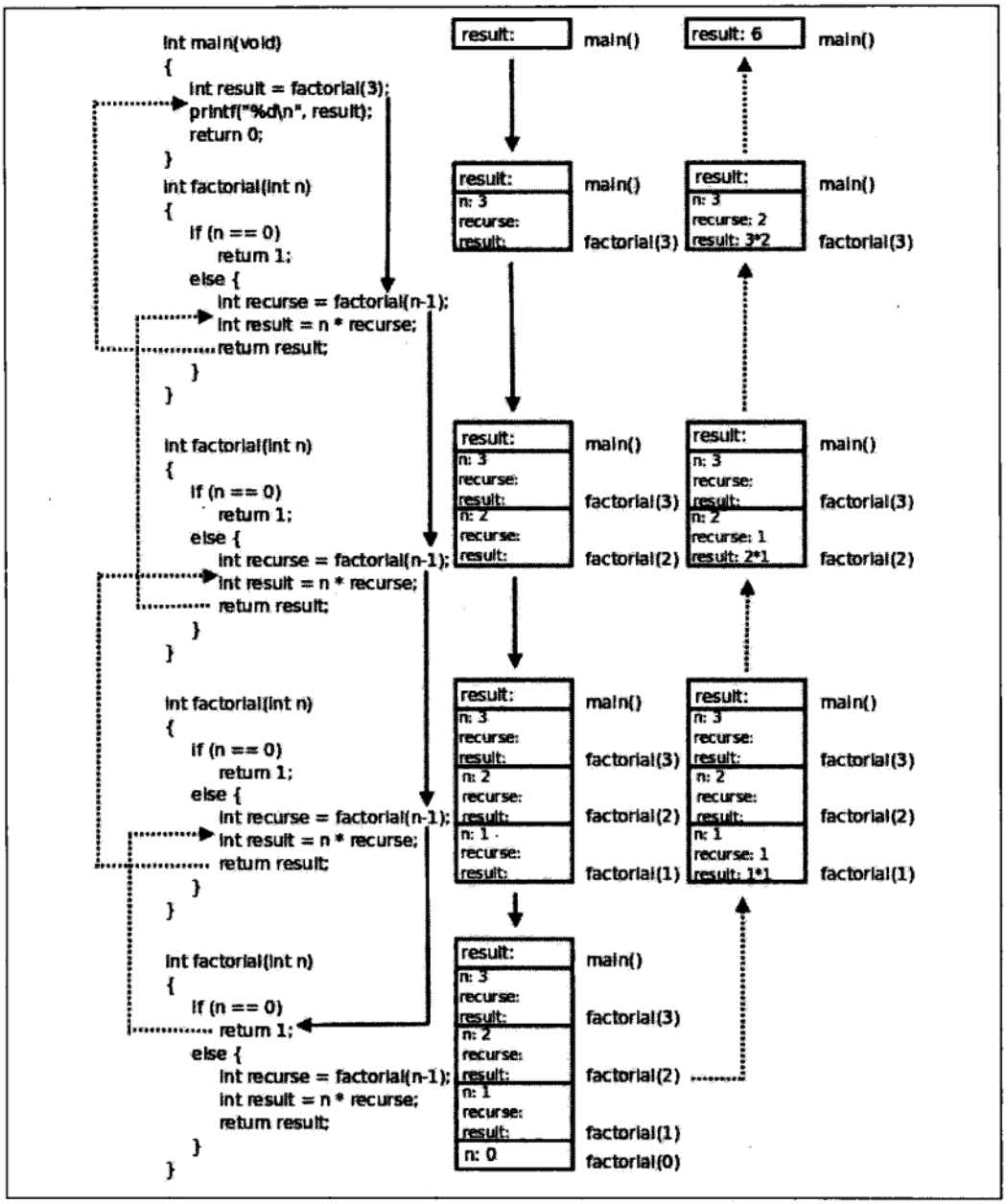


图 5.2　factorial(3)的调用过程

我们看图 5.2 右侧存储空间的变化过程，随着函数调用的层层深入，存储空间的一端逐渐增长，然后随着函数调用的层层返回，存储空间的这一端又逐渐缩短，并且每次访问参数和局部变量时只能访问这一端的存储单元，而不能访问内部的存储单元，比如当 factorial(2)的存储空间位于末端时，只能访问它的参数和局部变量，而不能访问 factorial(3)和 main()的参数和局部变量。具有这种性质的数据结构称为堆栈或栈（Stack），随着函数调用和返回而不断变化的这一端称为栈顶，每个函数调用的参数和局部变量的存储空间（图 5.2 中的每个小方框）称为一个栈帧（Stack Frame）。操作系统为程序的运行预留了一块栈空间，函数调用时就在这个栈空间里分配栈帧，函数返回时就释放栈帧。

在写一个递归函数时，你如何证明它是正确的？像上面那样跟踪函数的调用和返回过程算是一种办法，但只是factorial(3)就已经这么麻烦了，如果是factorial(100)呢？虽然我们已经证明了factorial(3)是正确的，因为它跟我们用数学公式计算的过程一样，结果也一样，但这不能代替factorial(100)的证明，你怎么办？别的函数你可以跟踪它的调用过程去证明它的正确性，因为每个函数只调用一次就返回了，但是对于递归函数，这么跟下去只会跟得你头都大了。事实上并不是每个函数调用都需要钻进去看的。我们在调用printf时没有钻进去看它是怎么打印的，我们只是**相信**它能打印，能正确完成它的工作，然后就继续写下面的代码了。在上一节中，我们写了distance和area函数，然后立刻测试证明了这两个函数是正确的，然后我们写area_point时调用了这两个函数：

```
    return area(distance(x1, y1, x2, y2));
```

在写这一句的时候，我们需要钻进distance和area函数中去走一趟才知道我们调用得是否正确吗？不需要，因为我们已经**相信**这两个函数能正确工作了，也就是相信把坐标传给distance它就能返回正确的距离，把半径传给area它就能返回正确的面积，因此调用它们去完成另外一件工作也应该是正确的。这种“相信”称为Leap of Faith，首先相信一些结论，然后再用它们去证明另外一些结论。

在写factorial(n)的代码时写到这个地方：

```
    ...
    int recurse = factorial(n-1);
    int result = n * recurse;
    ...
```

这时，如果我们相信factorial(n-1)是正确的，也就是相信传给它n-1它就能返回(n-1)!，那么recurse就是(n-1)!，那么result就是n*(n-1)!，也就是n!，这正是我们要返回的factorial(n)的结果。当然这有点奇怪：我们还没写完factorial这个函数，凭什么要相信factorial(n-1)是正确的？可Leap of Faith本身就是Leap（跳跃）的，不是吗？**如果你相信你正在写的递归函数是正确的，并调用它，然后在此基础上写完这个递归函数，那么它就会是正确的，从而值得你相信它正确。**

这么说好像有点儿玄，我们从数学上严格证明一下factorial函数的正确性。刚才说了，factorial(n)的正确性依赖于factorial(n-1)的正确性，只要后者正确，在后者的结果上乘个n返回这一步显然也没有疑问，那么我们的函数实现就是正确的。因此要证明factorial(n)的正确性就是要证明factorial(n-1)的正确性，同理，要证明factorial(n-1)的正确性就是要证明factorial(n-2)的正确性，依此类推下去，最后是：要证明factorial(1)的正确性就是要证明factorial(0)的正确性。而factorial(0)的正确性不依赖于别的函数调用，它就是程序中的一个小的分支return 1;，这个1是我们根据阶乘的定义写的，肯定是正确的，因此factorial(1)的实现是正确的，因此factorial(2)也正确，依此类推，最后factorial(n)也是正确的。其实这就是在中学时学的数学归纳法（Mathematical Induction），用数学归纳法来证明只需要证明两点：Base Case正确，递推关系正确。**写递归函数时一定要记得写Base Case**，否则即使递推关系正确，整个函数也不正确。如果factorial函数漏掉了Base Case：

```
int factorial(int n)
{
        int recurse = factorial(n-1);
        int result = n * recurse;
        return result;
}
```

那么这个函数就会永远调用下去，直到操作系统为程序预留的栈空间耗尽程序崩溃（段错误）为止，这称为无穷递归（Infinite recursion）。

到目前为止我们只学习了全部C语法的一个小的子集，但是现在应该告诉你：这个子集是完备的，它本身就可以作为一门编程语言了，以后还要学习很多C语言特性，但全部都可以用已经学过的这些特性来代替。也就是说，以后要学的C语言特性会使代码写起来更加方便，但不是必不可少的，现在学的这些已经完全覆盖了第1.1节讲的五种基本指令了。有的读者会说循环还没讲到呢，是的，循环在下一章才讲，但有一个重要的结论就是**递归和循环是等价的**，用循环能做的事用递归都能做，反之亦然，事实上有的编程语言（比如某些LISP实现）只有递归而没有循环。计算机指令能做的所有事情就是数据存取、运算、测试和分支、循环（或递归），在计算机上运行高级语言写的程序最终也要翻译成指令，指令做不到的事情高级语言写的程序肯定也做不到，虽然高级语言有丰富的语法特性，但也只是比指令写起来更方便而已，能做的事情是一样多的。那么，为什么计算机要设计成这样？在设计时怎么想到计算机应该具备这几样功能，而不是更多或更少的功能？这些要归功于早期的计算机科学家，例如Alan Turing，他们在计算机还没有诞生的年代就从数学理论上为计算机的设计指明了方向。有兴趣的读者可以参考有关计算理论的教材，例如参考文献[10]。

递归绝不只是为解决一些奇技淫巧的数学题而想出来的招，它是计算机的精髓所在，也是编程语言的精髓所在。我们在学习C的语法时已经看到很多递归定义了，例如在第3.1节讲过的语法规则中，“表达式”就是递归定义的：

表达式 → **表达式**(参数列表)
参数列表 → **表达式**, **表达式**, ...

再比如在第4.1节讲过的语法规则中，“语句”也是递归定义的：

语句 → **if** (控制表达式) **语句**

可见编译器在解析我们写的程序时一定也用了大量的递归，有关编译器的实现原理可参考文献[11]。

习题

1. 编写递归函数求两个正整数a和b的最大公约数（GCD，Greatest Common Divisor），使用Euclid算法：

② 例如很多编程书都会举例的汉诺塔问题，本书不打算再重复这个题目了。

- 如果 a 除以 b 能整除，则最大公约数是 b。
- 否则，最大公约数等于 b 和 a%b 的最大公约数。

Euclid 算法是很容易证明的，请读者自己证明一下为什么这么算就能算出最大公约数。最后，修改你的程序使之适用于所有整数，而不仅仅是正整数。

2. 编写递归函数求 Fibonacci 数列的第 n 项，这个数列是这样定义的：

```
fib(0)=1
fib(1)=1
fib(n)=fib(n-1)+fib(n-2)
```

上面两个看似毫不相干的问题之间却有一个有意思的联系：

Lamé定理

如果 Euclid 算法需要 k 步来计算两个数的 GCD，那么这两个数之中较小的一个必然大于等于 Fibonacci 数列的第 k 项。

感兴趣的读者可以查看参考文献[12]第 1.2 节的简略证明。

第 6 章

循环语句

6.1 while 语句

在第 5.3 节中，我们介绍了用递归求 n!的方法，其实每次递归调用都在重复做同样一件事，就是把 n 乘到(n-1)!上然后把结果返回。虽说是重复，但每次做都稍微有一点区别（n 的值不一样），这种每次都有一点区别的重复工作称为迭代（Iteration）。我们使用计算机的主要目的之一就是让它做重复迭代的工作，因为把一件工作重复做成千上万次而不出错正是计算机最擅长的，也是人类最不擅长的。虽然迭代用递归来做就够了，但 C 语言提供了循环语句使迭代程序写起来更方便。例如 factorial 用 while 语句可以写成：

```
int factorial(int n)
{
        int result = 1;
        while (n > 0) {
                result = result * n;
                n = n - 1;
        }
        return result;
}
```

和 if 语句类似，while 语句由一个控制表达式和一个子语句组成，子语句可以是由若干条语句和声明组成的语句块。

```
语句 → while (控制表达式) 语句
```

如果控制表达式的值为真，子语句就被执行，然后再次测试控制表达式的值，如果还是真，就把子语句再执行一遍，再测试控制表达式的值……这种控制流程称为循环（Loop），子语句称为循环体。如果某次测试控制表达式的值为假，就跳出循环执行后面的 return 语句，如果第一次测试控制表达式的值就是假，那么直接跳到 return 语句，循环体一次都不执行。

变量 result 在这个循环中的作用是累加器（Accumulator），把每次循环的中间结果累积起来，循环结束后得到的累积值就是最终结果，由于这个例子是用乘法来累积的，所以 result 的初值是 1，如果用加法累积则 result 的初值应该是 0。变量 n 是循环变量（Loop Variable），每次循环都要改变它的值，在控制表达式中要测试它的值，这两点合起来起到控制循环次数的作用，在这个例子中 n 的值是递

减的，也有些循环采用递增的循环变量。这个例子具有一定的典型性，累加器和循环变量这两种模式在循环中都很常见。

可见，递归能解决的问题用循环也能解决，但解决问题的思路不一样。用递归解决这个问题靠的是递推关系 n!=n·(n-1)!，用循环解决这个问题则更像是把这个公式展开了：n!=n·(n-1)·(n-2)·…·3×2×1。把公式展开了理解会更直观一些，所以有些时候循环程序比递归程序更容易理解。但也有一些公式要展开是非常复杂的甚至是不可能的，反倒是递推关系更直观一些，这种情况下递归程序比循环程序更容易理解。此外还有一点不同：看图 5.2，在整个递归调用过程中，虽然分配和释放了很多变量，但所有变量都只在初始化时赋值，没有任何变量的值发生过改变，而上面的循环程序则通过对 n 和 result 这两个变量多次赋值来达到同样的目的。前一种思路称为函数式编程（Functional Programming），而后一种思路称为命令式编程（Imperative Programming），这个区别类似于第 1.1 节讲的 Declarative 和 Imperative 的区别。函数式编程的“函数”类似于数学函数的概念，回顾一下第 3.1 节所讲的，数学函数是没有 Side Effect 的，而 C 语言的函数可以有 Side Effect，比如在一个函数中修改某个全局变量的值就是一种 Side Effect。第 3.4 节指出，全局变量被多次赋值会给调试带来麻烦，如果一个函数体很长，控制流程很复杂，那么局部变量被多次赋值也会有同样的问题。因此，不要以为“变量可以多次赋值”是天经地义的，有很多编程语言可以完全采用函数式编程的方式，避免 Side Effect，例如 LISP、Haskell、Erlang 等。用 C 语言编程主要还是采用 Imperative 的方式，但要记住，**给变量多次赋值时要格外小心，在代码中多次读写同一变量应该以一种一致的方式进行**。所谓“一致的方式”是说应该有一套统一的规则，规定在一段代码中哪里会对某个全局变量赋值、哪里会读取它的值，比如在第 24.2.4 节会讲到访问 errno 的规则。

递归函数如果写得不小心就会变成无穷递归，同样道理，循环如果写得不小心就会变成无限循环（Infinite Loop）或者叫死循环。如果 while 语句的控制表达式永远为真就成了一个死循环，例如 while (1) {...}。在写循环时要小心检查你写的控制表达式有没有可能取值为假，除非你故意写死循环（有的时候这是必要的）。在上面的例子中，不管 n 一开始是几，每次循环都会把 n 减掉 1，n 越来越小最后必然等于 0，所以控制表达式最后必然取值为假，但如果把 n = n - 1;这句漏掉就成了死循环。有的时候是不是死循环并不是那么一目了然：

```
while (n != 1) {
        if (n % 2 == 0) {
                n = n / 2;
        } else {
                n = n * 3 + 1;
        }
}
```

如果 n 为正整数，这个循环能跳出来吗？循环体所做的事情是：如果 n 是偶数，就把 n 除以 2，如果 n 是奇数，就把 n 乘 3 加 1。一般来说循环变量要么递增要么递减，可是这个例子中的 n 一会儿变大一会儿变小，最终会不会变成 1 呢？可以找个数试试，例如一开始 n 等于 7，每次循环后 n 的值依次是：7、22、11、34、

17、52、26、13、40、20、10、5、16、8、4、2、1。最后 n 确实等于 1 了。读者可以再试几个数都是如此，但无论试多少个数也不能代替证明，这个循环有没有可能对某些正整数 n 是死循环呢？其实这个例子只是给读者提提兴趣，同时提醒读者写循环时要有意识地检查控制表达式。至于这个循环有没有可能是死循环，这是著名的 3x+1 问题，目前世界上还无人能证明。许多世界难题都是这样的：问题的描述无比简单，连小学生都能看懂，但证明却无比困难。

习题

1．用循环解决第 5.3 节中的所有习题，体会递归和循环这两种不同的思路。

2．编写程序数一下 1 到 100 的所有整数中出现多少次数字 9。在写程序之前先把这些问题考虑清楚：

- 这个问题中的循环变量是什么？
- 这个问题中的累加器是什么？用加法还是用乘法累积？
- 在第 4.2 节的习题 1 写过取一个整数的个位和十位的表达式，这两个表达式怎样用到程序中？

3．如果一个循环体中有这样的语句：

```
while (...) {
        int i = 0;
        printf("%d\n", i);
        i = i + 1;
}
```

每次循环打印的 i 值会增加 1 吗？

6.2 do/while 语句

do/while 语句的语法是：

```
语句 → do 语句 while (控制表达式);
```

while 语句先测试控制表达式的值再执行循环体，而 do/while 语句先执行循环体再测试控制表达式的值。如果控制表达式的值一开始就是假，while 语句的循环体一次都不执行，而 do/while 语句的循环体仍然要执行一次再跳出循环。其实只要有 while 循环就足够了，do/while 循环和后面要讲的 for 循环都可以改写成 while 循环，只不过有些情况下用 do/while 或 for 循环写起来更简便，代码更易读。上面的 factorial 也可以改用 do/while 循环来写：

```
int factorial(int n)
{
        int result = 1;
        int i = 1;
        do {
```

```
            result = result * i;
            i = i + 1;
        } while (i <= n);

        return result;
    }
```

写循环一定要注意循环即将结束时控制表达式的临界条件是否准确，上面的循环控制条件如果写成i<n就错了，当i==n时跳出循环，最后的结果中就少乘了一个n。虽然变量名应该尽可能起得有意义一些，不过用i、j、k给循环变量起名是很常见的。

6.3 for语句

前两节我们在while和do/while循环中使用循环变量，其实使用循环变量最常见的是for循环这种形式。for语句的语法是：

语句 → for (控制表达式1; 控制表达式2; 控制表达式3) 语句

如果不考虑循环体中包含continue语句的情况（稍后介绍continue语句），这个for循环等价于下面的while循环：

```
控制表达式1;
while (控制表达式2) {
        语句
        控制表达式3;
}
```

从这种等价形式来看，控制表达式1和3都可以为空，但控制表达式2是必不可少的，例如for (;1;) {...}等价于while (1) {...}死循环。C语言规定，如果控制表达式2为空，则认为控制表达式2的值为真，因此死循环也可以写成for (;;) {...}。

上一节do/while循环的例子可以改写成for循环：

```
int factorial(int n)
{
    int result = 1;
    int i;
    for(i = 1; i <= n; ++i)
        result = result * i;
    return result;
}
```

其中++i这个表达式相当于i = i + 1[①]，++称为前缀自增运算符（Prefix Increment Operator），类似地，--称为前缀自减运算符（Prefix Decrement Operator）[②]，--i

① 这两种写法在语义上稍有区别，详见第15.2.1节。

② increment和decrement这两个词很有意思，大多数字典都说它们是名词，但经常被当成动词用，在计算机术语中，它们当动词用应该理解为increase by one和decrease by one。现代英语中很多原本是名词的都被当成动词用，字典都跟不上时代了，再比如transition也是如此。

相当于 i = i - 1。如果把++i 这个表达式看作一个函数调用，除了传入一个参数返回一个值（等于参数值加 1）之外，还产生一个 Side Effect，就是把变量 i 的值增加了 1。

++和--运算符也可以用在变量后面，例如 i++和 i--，为了和前缀运算符区别，这两个运算符称为后缀自增运算符（Postfix Increment Operator）和后缀自减运算符（Postfix Decrement Operator）。如果把 i++这个表达式看作一个函数调用，传入一个参数返回一个值，返回值就等于参数值（而不是参数值加 1），此外也产生一个 Side Effect，就是把变量 i 的值增加了 1，它和++i 的区别就在于返回值不同。同理，--i 返回减 1 之后的值，而 i--返回减 1 之前的值，但这两个表达式都产生同样的 Side Effect，就是把变量 i 的值减了 1。

使用++、--运算符会使程序更加简洁，但也会影响程序的可读性，参考文献[3]中的示例代码大量运用++、--和其他表达式的组合使得代码非常简洁。为了让初学者循序渐进，在接下来的几章中++、--运算符总是单独组成一个表达式而不跟其他表达式组合，从第 11 章开始将采用参考文献[3]的简洁风格。

我们看一个有意思的问题：a+++++b 这个表达式如何理解？应该理解成 a++ ++ +b 还是 a++ + ++b，还是 a + ++ ++b 呢？应该按第一种方式理解。编译的过程分为词法解析和语法解析两个阶段，在词法解析阶段，编译器总是从前到后找最长的合法 Token。把这个表达式从前到后解析，变量名 a 是一个 Token，a 后面有两个以上的+号，在 C 语言中一个+号是合法的 Token（可以是加法运算符或正号），两个+号也是合法的 Token（可以是自增运算符），根据最长匹配原则，编译器绝不会止步于一个+号，而一定会把两个+号当做一个 Token。再往后解析仍然有两个以上的+号，所以又是一个++运算符。再往后解析只剩一个+号了，是加法运算符。再往后解析是变量名 b。词法解析之后进入下一阶段语法解析，a 是一个表达式，表达式++还是表达式，表达式再++还是表达式，表达式再+b 还是表达式，语法上没有问题。最后编译器会做一些基本的语义分析，这时就有问题了，++运算符要求操作数能做左值，a 能做左值所以 a++没问题，但表达式 a++的值只能做右值，不能再++了，所以最终编译器会报错。

C99 规定了一种新的 for 循环语法（其实是从 C++借鉴的），在控制表达式 1 的位置可以有变量定义。例如上例的循环变量 i 可以只在 for 循环中定义：

```
int factorial(int n)
{
    int result = 1;
    for(int i = 1; i <= n; i++)
        result = result * i;
    return result;
}
```

如果这样定义，那么变量 i 只是 for 循环中的局部变量而不是整个函数的局部变量，相当于第 4.1 节讲过的语句块中的局部变量，在循环结束后就不能再使用 i 这个变量了，注意这个程序用 gcc 编译时必须加上选项-std=c99。

6.4 break和continue语句

在第4.4节中我们见到了break语句的一种用法，用来跳出switch语句块，这个语句也可以用来跳出循环体。continue语句也会终止当前循环，和break语句不同的是，continue语句终止当前循环后又回到循环体的开头准备执行下一次循环。对于while循环和do/while循环，执行continue语句之后测试控制表达式，如果值为真则继续执行下一次循环；对于for循环，执行continue语句之后首先计算控制表达式3，然后测试控制表达式2，如果值为真则继续执行下一次循环。例如下面的代码打印1到100之间的素数：

例6.1 求1～100的素数

```
#include <stdio.h>

int is_prime(int n)
{
        int i;
        for (i = 2; i < n; i++)
                if (n % i == 0)
                        break;
        if (i == n)
                return 1;
        else
                return 0;
}

int main(void)
{
        int i;
        for (i = 1; i <= 100; i++) {
                if (!is_prime(i))
                        continue;
                printf("%d\n", i);
        }
        return 0;
}
```

is_prime函数从2到n-1依次检查有没有能被n整除的数，如果有就说明n不是素数，立刻跳出循环而不执行i++。因此，如果n不是素数，则循环结束后i一定小于n，如果n是素数，则循环结束后i一定等于n。注意检查临界条件：2应该是素数，如果n是2，则循环体一次也不执行，但i的初值就是2，也等于n，在程序中也判定为素数。其实没有必要从2一直检查到n-1，只要从2检查到⌊sqrt(n)⌋，如果全都不能整除就足以证明n是素数了，请读者想一想为什么。

在主程序中，从1到100依次检查每个数是不是素数，如果不是素数，并不直接跳出循环，而是i++后继续执行下一次循环，因此用continue语句。注意主程序的局部变量i和is_prime中的局部变量i是不同的两个变量，其实在调用is_prime函数时主程序的局部变量i和参数n的值相等。

习题

1．求素数这个程序只是为了说明 break 和 continue 的用法才这么写的，其实完全可以不用 break 和 continue，请读者修改一下控制流程，去掉 break 和 continue 而保持功能不变。

2．上一节讲过怎样把 for 循环改写成等价的 while 循环，但也提到如果循环体中有 continue 语句这两种形式就不等价了，想一想为什么不等价了？

6.5 嵌套循环

上一节求素数的例子在循环中调用一个函数，而那个函数里面又有一个循环，这其实是一种嵌套循环。如果把那个函数的代码拿出来写就更清楚了：

例 6.2 用嵌套循环求 1～100 的素数

```
#include <stdio.h>

int main(void)
{
        int i, j;
        for (i = 1; i <= 100; i++) {
                for (j = 2; j < i; j++)
                        if (i % j == 0)
                                break;
                if (j == i)
                        printf("%d\n", i);
        }
        return 0;
}
```

现在内循环的循环变量就不能再用 i 了，而是改用 j，原来程序中 is_prime 函数的参数 n 现在直接用 i 代替。在有多层循环或 switch 嵌套的情况下，break 只能跳出最内层的循环或 switch，continue 也只能终止最内层循环并回到该循环的开头。

用循环也可以打印表格式的数据，比如打印小九九乘法表。

例 6.3 打印小九九乘法表

```
#include <stdio.h>

int main(void)
{
        int i, j;
        for (i=1; i<=9; i++) {
                for (j=1; j<=9; j++)
                        printf("%d  ", i*j);
                printf("\n");
        }
        return 0;
}
```

内循环每次打印一个数，数与数之间用两个空格隔开，外循环每次打印一行。结果如下：

```
1  2  3  4  5  6  7  8  9
2  4  6  8  10  12  14  16  18
3  6  9  12  15  18  21  24  27
4  8  12  16  20  24  28  32  36
5  10  15  20  25  30  35  40  45
6  12  18  24  30  36  42  48  54
7  14  21  28  35  42  49  56  63
8  16  24  32  40  48  56  64  72
9  18  27  36  45  54  63  72  81
```

结果有一位数的有两位数的，这个表格很不整齐，如果把打印语句改为printf("%d\t", i*j);就整齐了，所以Tab字符称为制表符。

习题

1．上面打印的小九九有一半数据是重复的，因为8×9和9×8的结果一样。请修改程序打印这样的小九九：

```
1
2   4
3   6   9
4   8   12  16
5   10  15  20  25
6   12  18  24  30  36
7   14  21  28  35  42  49
8   16  24  32  40  48  56  64
9   18  27  36  45  54  63  72  81
```

2．编写函数diamond打印一个菱形。如果调用diamond(3, '*')则打印：

```
    *
*   *   *
    *
```

如果调用diamond(5, '+')则打印：

```
        +
    +   +   +
+   +   +   +   +
    +   +   +
        +
```

如果用偶数做参数则打印错误提示。

6.6 goto语句和标号

分支、循环都讲完了，现在只剩下最后一种影响控制流程的语句了，就是goto语句，实现无条件跳转。我们知道break只能跳出最内层的循环，如果在一个嵌套循环中遇到某个错误条件需要立即跳出最外层循环做出错处理，就可以用goto语

句，例如：

```
for (...)
        for (...) {
                ...
                if (出现错误条件)
                        goto error;
        }
error:
        出错处理;
```

这里的 error:叫做标号（Label），任何语句前面都可以加若干个标号，每个标号的命名也要遵循标识符的命名规则。

goto 语句过于强大了，从程序中的任何地方都可以无条件跳转到任何其他地方，只要在那个地方定义一个标号就行，唯一的限制是 goto 只能跳转到同一个函数中的某个标号处，而不能跳到别的函数中[③]。**滥用 goto 语句会使程序的控制流程非常复杂，可读性很差**。著名的计算机科学家 Edsger W. Dijkstra 最早指出编程语言中 goto 语句的危害，提倡取消 goto 语句。goto 语句不是必须存在的，显然可以用别的办法替代，比如上面的代码段可以改写为：

```
int cond = 0; /* bool variable indicating error condition */
for (...) {
        for (...) {
                ...
                if (出现错误条件) {
                        cond = 1;
                        break;
                }
        }
        if (cond)
                break;
}
if (cond)
        出错处理;
```

通常 goto 语句只用于这种场合，一个函数中任何地方出现了错误条件都可以立即跳转到函数末尾做出错处理（例如释放先前分配的资源、恢复先前改动过的全局变量等），处理完之后函数返回。比较用 goto 和不用 goto 的两种写法，用 goto 语句还是方便很多。但是除此之外，在任何其他场合都不要轻易考虑使用 goto 语句。有些编程语言（如 C++）中有异常（Exception）处理的语法，可以代替 goto 和 setjmp/longjmp 的这种用法。

③ C 标准库函数 setjmp 和 longjmp 配合起来可以实现函数间的跳转，但只能从被调用的函数跳回到它的直接或间接调用者（同时从栈空间弹出一个或多个栈帧），而不能从一个函数跳转到另一个和它毫不相干的函数中。setjmp/longjmp 函数主要也是用于出错处理，比如函数 A 调用函数 B，函数 B 调用函数 C，如果在 C 中出现某个错误条件，使得函数 B 和 C 继续执行下去都没有意义了，可以利用 setjmp/longjmp 机制快速返回到函数 A 做出错处理，本书不详细介绍这种机制，有兴趣的读者可查阅参考文献[31]的 7.10 节和 10.15 节。

回想一下，我们在第4.4节学过case和default后面也要跟冒号（:号，Colon），事实上它们是两种特殊的标号。和标号有关的语法规则如下：

```
语句 → 标识符: 语句
语句 → case 常量表达式: 语句
语句 → default: 语句
```

反复应用这些语法规则进行组合可以在一条语句前面添加多个标号，例如在例4.2的代码中，有些语句前面有多个case标号。现在我们再看switch语句的格式：

```
switch (控制表达式) {
case 常量表达式:  零或多条语句
case 常量表达式:  零或多条语句
...
default:  零或多条语句
}
```

{}里面是一组语句列表，其中每个分支的第一条语句带有case或default标号，从语法上来说，switch的语句块和其他分支、循环结构的语句块没有本质区别，因此前面的语法规则可以改写为：

```
语句 → switch (控制表达式) 语句
语句 → { 语句列表 }
```

改写后的语法规则更为准确，我们知道{}中的语句列表不仅可以包含语句，还可以包含声明，而前面的语法规则并没有体现出这一点。但要注意，只有语句前面才能带标号，声明前面不能带标号，例如这样写是错的：

```
case 1: int i = 10;
```

但这样写是对的：

```
case 1: { int i = 10; }
```

这样写也是对的：

```
case 1: ; int i = 10;
```

再比如这样的switch语句：

```
switch (n) {
        int i = 10;
case 1:
        printf("%d\n", i * 1);
        break;
case 2:
        printf("%d\n", i * 2);
        break;
}
```

这段代码从语法上看是对的，从语义上看却有一个陷阱。变量i在switch语句块中定义，但并不会初始化成10，因为不管n的值是几，进入switch语句块都会跳

过给 i 赋初值的指令，从某一个 case 标号开始执行。

习题

1. 以下代码编译没有问题，但运行结果却和预期不符（不能打印出 other number），请分析原因。

```
int n = 3;
switch (n) {
case 1:
        printf("1\n");
        break;
case 2:
        printf("2\n");
        break;
default:
        printf("other number\n");
}
```

2. 请在网上查找有关 Duff's Device 的资料，Duff's Device 是一段很有意思的代码，正是利用“switch 的语句块和循环结构的语句块没有本质区别”这一点实现了一个巧妙的代码优化。

第 7 章

结构体

7.1 复合类型与结构体

在编程语言中，最基本的、不可再分的数据类型称为基本类型（Primitive Type），例如整型、浮点型；根据语法规则由基本类型组合而成的类型称为复合类型（Compound Type），例如字符串是由很多字符组成的。有些场合下要把复合类型当做一个整体来用，而另外一些场合下需要分解组成这个复合类型的各种基本类型，复合类型的这种两面性为数据抽象（Data Abstraction）奠定了基础。参考文献[12]的 1.1 节指出，在学习一门编程语言时要特别注意以下三个方面：

1．这门语言提供了哪些 Primitive，比如基本类型，比如基本运算符、表达式和语句。

2．这门语言提供了哪些组合规则，比如基本类型如何组成复合类型，比如简单的表达式和语句如何组成复杂的表达式和语句。

3．这门语言提供了哪些抽象机制，包括数据抽象和过程抽象（Procedure Abstraction）。

本章以结构体为例讲解数据类型的组合和数据抽象。至于过程抽象，我们在第 4.2 节已经见过最简单的形式，就是把一组语句用一个函数名封装起来，当做一个整体使用，本章将介绍更复杂的过程抽象。

现在我们用 C 语言表示一个复数。从直角坐标系来看，复数由实部和虚部组成，从极坐标系来看，复数由模和辐角组成，两种坐标系可以相互转换，如图 7.1 所示。

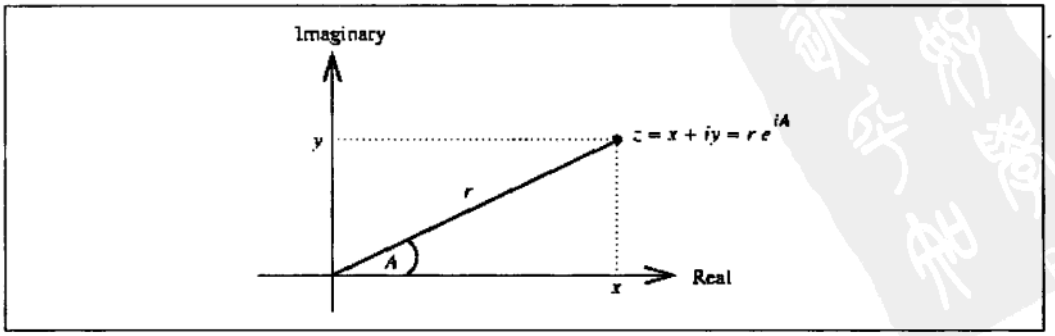


图 7.1 复数

如果用实部和虚部表示一个复数，我们可以写成由两个 double 型组成的结构体：

```
struct complex_struct {
        double x, y;
};
```

这一句定义了标识符 complex_struct（同样遵循标识符的命名规则），这种标识符在 C 语言中称为 Tag，struct complex_struct { double x, y; }整个可以看作一个类型名[①]，就像 int 或 double 一样，只不过它是一个复合类型，如果用这个类型名来定义变量，可以这样写：

```
struct complex_struct {
        double x, y;
} z1, z2;
```

这样 z1 和 z2 就是两个变量名，变量定义后面带个;号是我们早就习惯的。但即使像先前的例子那样只定义了 complex_struct 这个 Tag 而不定义变量，}后面的;号也不能少。这点一定要注意，类型定义也是一种声明，声明都要以;号结尾，结构体类型定义的}后面少;号是初学者常犯的错误。不管是用上面两种形式的哪一种定义了 complex_struct 这个 Tag，以后都可以直接用 struct complex_struct 来代替类型名了。例如可以这样定义另外两个复数变量：

```
struct complex_struct z3, z4;
```

如果在定义结构体类型的同时定义了变量，也可以不必写 Tag，例如：

```
struct {
        double x, y;
} z1, z2;
```

但这样就没办法再次引用这个结构体类型了，因为它没有名字。每个复数变量都有两个成员（Member）x 和 y，可以用.后缀运算符（.号，Period）来访问，这两个成员的存储空间是相邻的[②]，合在一起组成复数变量的存储空间。看下面的例子：

例 7.1　定义和访问结构体

```
#include <stdio.h>

int main(void)
{
        struct complex_struct { double x, y; } z;
        double x = 3.0;
        z.x = x;
        z.y = 4.0;
```

① 其实 C99 已经定义了复数类型_Complex。如果包含 C 标准库的头文件 complex.h，也可以用 complex 做类型名。当然，只要不包含头文件 complex.h 就可以自己定义标识符 complex，但为了尽量减少混淆，本章的示例代码都用 complex_struct 做标识符而不用 complex。

② 我们在第 18.4 节会看到，结构体成员之间也可能有若干个填充字节。

```
        if (z.y < 0)
                printf("z=%f%fi\n", z.x, z.y);
        else
                printf("z=%f+%fi\n", z.x, z.y);

        return 0;
}
```

注意上例中变量 x 和变量 z 的成员 x 的名字并不冲突，因为变量 z 的成员 x 只能通过表达式 z.x 来访问，编译器可以从语法上区分哪个 x 是变量 x，哪个 x 是变量 z 的成员 x，第 18.3 节会讲到这两个标识符 x 属于不同的命名空间。结构体 Tag 也可以定义在全局作用域中，这样定义的 Tag 在其定义之后的各函数中都可以使用。例如：

```
struct complex_struct { double x, y; };

int main(void)
{
        struct complex_struct z;
        ...
}
```

结构体变量也可以在定义时初始化，例如：

```
struct complex_struct z = { 3.0, 4.0 };
```

Initializer 中的数据依次赋给结构体的各成员。如果 Initializer 中的数据比结构体的成员多，编译器会报错，但如果只是末尾多个逗号则不算错。如果 Initializer 中的数据比结构体的成员少，未指定的成员将用 0 来初始化，就像未初始化的全局变量一样。例如以下几种形式的初始化都是合法的：

```
double x = 3.0;
struct complex_struct z1 = { x, 4.0, }; /* z1.x=3.0, z1.y=4.0 */
struct complex_struct z2 = { 3.0, }; /* z2.x=3.0, z2.y=0.0 */
struct complex_struct z3 = { 0 }; /* z3.x=0.0, z3.y=0.0 */
```

注意，z1 必须是局部变量才能用另一个变量 x 的值来初始化它的成员，如果是全局变量就只能用常量表达式来初始化。这也是 C99 的新特性，C89 只允许在{}中使用常量表达式来初始化，无论是初始化全局变量还是局部变量。

{}这种语法不能用于结构体的赋值，例如这样是错误的：

```
struct complex_struct z1;
z1 = { 3.0, 4.0 };
```

以前我们初始化基本类型的变量所使用的 Initializer 都是表达式，表达式当然也可以用来赋值，但现在这种由{}括起来的 Initializer 并不是表达式，所以不能用来赋值[③]。Initializer 的语法总结如下：

③ C99 引入一种新的表达式语法 Compound Literal 可以用来赋值，例如 z1 = (struct complex_struct){ 3.0, 4.0 };，本书不使用这种新语法。

```
Initializer → 表达式
Initializer → { 初始化列表 }
初始化列表 → Designated-Initializer, Designated-Initializer, ...
（最后一个 Designated-Initializer 末尾可以有一个多余的,号）
Designated-Initializer → Initializer
Designated-Initializer → .标识符 = Initializer
Designated-Initializer → [常量表达式] = Initializer
```

Designated Initializer 是 C99 引入的新特性，用于初始化稀疏（Sparse）结构体和稀疏数组很方便。有些时候结构体或数组中只有某一个或某几个成员需要初始化，其他成员都用 0 初始化即可，用 Designated Initializer 语法可以很方便地针对每个成员做初始化（Memberwise Initialization）例如：

```
struct complex_struct z1 = { .y = 4.0 }; /* z1.x=0.0, z1.y=4.0 */
```

数组的 Memberwise Initialization 语法将在下一章介绍。

结构体类型用在表达式中有很多限制，不像基本类型那么自由，比如+ - * /等算术运算符和&& || !等逻辑运算符都不能作用于结构体类型，if 语句、while 语句中的控制表达式的值也不能是结构体类型。严格来说，可以做算术运算的类型称为算术类型（Arithmetic Type），算术类型包括整型和浮点型。可以表示零和非零，可以参与逻辑与、或、非运算或者做控制表达式的类型称为标量类型（Scalar Type），标量类型包括算术类型和以后要讲的指针类型，详见图 22.5。

结构体变量之间使用赋值运算符是允许的，用一个结构体变量初始化另一个结构体变量也是允许的，例如：

```
struct complex_struct z1 = { 3.0, 4.0 };
struct complex_struct z2 = z1;
z1 = z2;
```

同样地，z2 必须是局部变量才能用变量 z1 的值来初始化。既然结构体变量之间可以相互赋值和初始化，也就可以当做函数的参数和返回值来传递：

```
struct complex_struct add_complex(struct complex_struct z1, struct
complex_struct z2)
{
        z1.x = z1.x + z2.x;
        z1.y = z1.y + z2.y;
        return z1;
}
```

这个函数实现了两个复数相加，如果在 main 函数中这样调用：

```
struct complex_struct z = { 3.0, 4.0 };
z = add_complex(z, z);
```

那么调用传参的过程如图 7.2 所示。

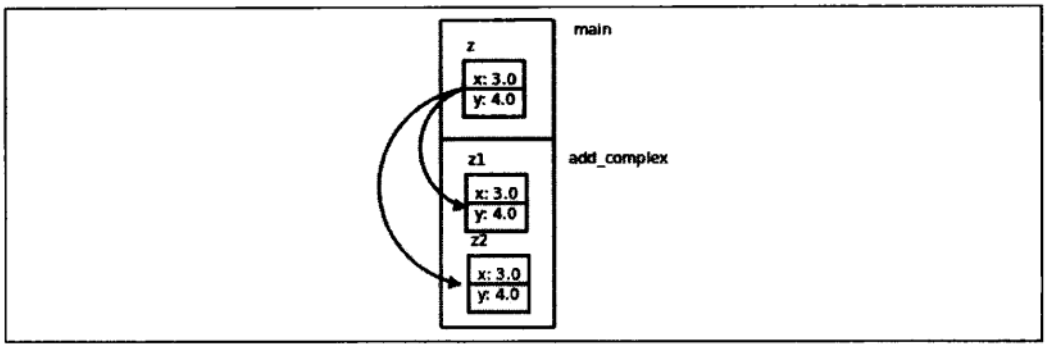


图 7.2 结构体传参

变量 z 在 main 函数的栈帧上，参数 z1 和 z2 在 add_complex 函数的栈帧上，z 的值分别赋给 z1 和 z2。在这个函数里，z2 的实部和虚部被累加到 z1 中，然后 return z1;可以看成是：

1. 用 z1 初始化一个临时变量。

2. 函数返回并释放栈帧。

3. 把临时变量的值赋给变量 z，释放临时变量。

由.后缀运算符组成的表达式能不能做左值取决于.后缀运算符左边的操作数能不能做左值。在上面的例子中，z 是一个变量，可以做左值，因此表达式 z.x 也可以做左值，但表达式 add_complex(z, z).x 只能做右值而不能做左值，因为 add_complex(z, z)不能做左值。

7.2 数据抽象

现在我们来实现一个完整的复数运算程序。在上一节我们已经定义了复数的结构体类型，现在需要围绕它定义一些函数。复数可以用直角坐标或极坐标表示，直角坐标做加减法比较方便，极坐标做乘除法比较方便。如果我们定义的复数结构体是直角坐标的，那么应该提供极坐标的转换函数，以便在需要的时候可以方便地取它的模和辐角：

```
#include <math.h>

struct complex_struct {
        double x, y;
};

double real_part(struct complex_struct z)
{
        return z.x;
}

double img_part(struct complex_struct z)
{
        return z.y;
```

```
}

double magnitude(struct complex_struct z)
{
        return sqrt(z.x * z.x + z.y * z.y);
}

double angle(struct complex_struct z)
{
        return atan2(z.y, z.x);
}
```

此外，我们还提供两个函数用来构造复数变量，参数既可以是直角坐标也可以是极坐标，在函数中自动做相应的转换然后返回构造的复数变量：

```
struct complex_struct make_from_real_img(double x, double y)
{
        struct complex_struct z;
        z.x = x;
        z.y = y;
        return z;
}

struct complex_struct make_from_mag_ang(double r, double A)
{
        struct complex_struct z;
        z.x = r * cos(A);
        z.y = r * sin(A);
        return z;
}
```

在此基础上就可以实现复数的加减乘除运算了：

```
struct complex_struct add_complex(struct complex_struct z1, struct
complex_struct z2)
{
        return make_from_real_img(real_part(z1) + real_part(z2),
                                  img_part(z1) + img_part(z2));
}

struct complex_struct sub_complex(struct complex_struct z1, struct
complex_struct z2)
{
        return make_from_real_img(real_part(z1) - real_part(z2),
                                  img_part(z1) - img_part(z2));
}

struct complex_struct mul_complex(struct complex_struct z1, struct
complex_struct z2)
{
        return make_from_mag_ang(magnitude(z1) * magnitude(z2),
                                 angle(z1) + angle(z2));
}

struct complex_struct div_complex(struct complex_struct z1, struct
complex_struct z2)
{
        return make_from_mag_ang(magnitude(z1) / magnitude(z2),
```

```
                                        angle(z1) - angle(z2));
}
```

可以看出，复数加减乘除运算的实现并没有直接访问结构体 complex_struct 的成员 x 和 y，而是把它看成一个整体，通过调用相关函数来取它的直角坐标和极坐标。这样就可以非常方便地替换掉结构体 complex_struct 的存储表示，例如改用极坐标来存储：

```
#include <math.h>

struct complex_struct {
        double r, A;
};

double real_part(struct complex_struct z)
{
        return z.r * cos(z.A);
}

double img_part(struct complex_struct z)
{
        return z.r * sin(z.A);
}

double magnitude(struct complex_struct z)
{
        return z.r;
}

double angle(struct complex_struct z)
{
        return z.A;
}

struct complex_struct make_from_real_img(double x, double y)
{
        struct complex_struct z;
        z.A = atan2(y, x);
        z.r = sqrt(x * x + y * y);
        return z;
}

struct complex_struct make_from_mag_ang(double r, double A)
{
        struct complex_struct z;
        z.r = r;
        z.A = A;
        return z;
}
```

虽然结构体 complex_struct 的存储表示做了这样的改动，add_complex、sub_complex、mul_complex、div_complex 这几个复数运算的函数却不需要做任何改动，仍然可以用，原因在于这几个函数只把结构体 complex_struct 当做一个整体来使用，而没有直接访问它的成员，因此也不依赖于它有哪些成员。我们结合图 7.3 具体分析一下。

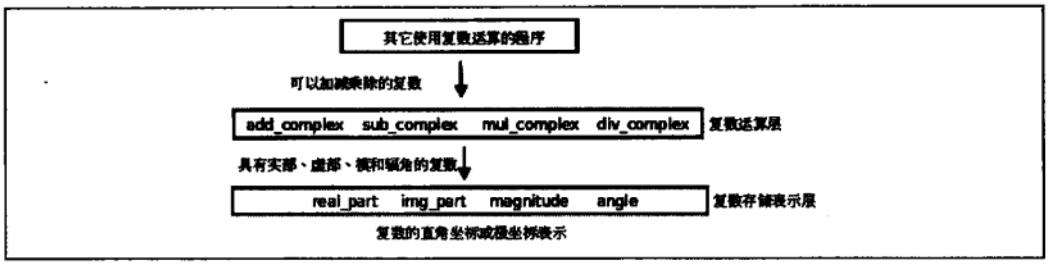


图 7.3　数据抽象

这是一种抽象的思想。其实“抽象”这个概念并没有那么抽象，简单地说就是“提取公因式”：ab+ac=a(b+c)。如果 a 变了，ab 和 ac 这两项都需要改，但如果写成 a(b+c)的形式就只需要改其中一个因子。

在我们的复数运算程序中，复数有可能用直角坐标或极坐标来表示，我们把这个有可能变动的因素提取出来组成复数存储表示层：real_part、img_part、magnitude、angle、make_from_real_img、make_from_mag_ang。这一层看到的数据是结构体的两个成员 x 和 y，或者 r 和 A，如果改变了结构体的实现就要改变这一层函数的实现，但函数接口不改变，因此调用这一层函数接口的复数运算层不需要改变。复数运算层看到的数据只是一个抽象的“复数”的概念，知道它有直角坐标和极坐标，可以调用复数存储表示层的函数得到这些坐标。再往上看，其他使用复数运算的程序看到的数据是一个更为抽象的“复数”的概念，只知道它是一个数，像整数、小数一样可以加减乘除，甚至连它有直角坐标和极坐标也不需要知道。

这里的复数存储表示层和复数运算层称为抽象层（Abstraction Layer），从底层往上层来看，复数越来越抽象了，把所有这些层组合在一起就是一个完整的系统。**组合使得系统可以任意复杂，而抽象使得系统的复杂性是可以控制的，任何改动都只局限在某一层，而不会波及整个系统。**著名的计算机科学家 Butler Lampson 说过：“All problems in computer science can be solved by another level of indirection.”这里的 indirection 其实就是 abstraction 的意思。

习题

1．在本节的基础上实现一个打印复数的函数，打印的格式是 x+yi，如果实部或虚部为 0 则省略，例如：1.0、-2.0i、-1.0+2.0i、1.0-2.0i。最后编写一个 main 函数测试本节的所有代码。想一想这个打印函数应该属于图 7.3 中的哪一层？

2．实现一个用分子分母的格式来表示有理数的结构体 rational 以及相关的函数，rational 结构体之间可以做加减乘除运算，运算的结果仍然是 rational。测试代码如下：

```
int main(void)
{
    struct rational a = make_rational(1, 8); /* a=1/8 */
    struct rational b = make_rational(-1, 8); /* b=-1/8 */
    print_rational(add_rational(a, b));
```

```
        print_rational(sub_rational(a, b));
        print_rational(mul_rational(a, b));
        print_rational(div_rational(a, b));

        return 0;
    }
```

注意要约分为最简分数，例如 1/8 和-1/8 相减的打印结果应该是 1/4 而不是 2/8，可以利用第 5.3 节习题中的 Euclid 算法来约分。在动手编程之前先思考一下这个问题实现了什么样的数据抽象，抽象层应该由哪些函数组成。

7.3 数据类型标志

在上一节中，我们通过一个复数存储表示抽象层把 complex_struct 结构体的存储格式和上层的复数运算函数隔开，complex_struct 结构体既可以采用直角坐标也可以采用极坐标存储。但有时候需要同时支持两种存储格式，比如先前已经采集了一些数据存在计算机中，有些数据是以极坐标存储的，有些数据是以直角坐标存储的，如果要把这些数据都存到 complex_struct 结构体中怎么办？一种办法是规定 complex_struct 结构体采用直角坐标格式，直角坐标的数据可以直接存入 complex_struct 结构体，而极坐标的数据先转成直角坐标再存，但由于浮点数的精度有限，转换总是会损失精度的。这里介绍另一种办法，complex_struct 结构体由一个数据类型标志和两个浮点数组成，如果数据类型标志为 0，那么两个浮点数就表示直角坐标，如果数据类型标志为 1，那么两个浮点数就表示极坐标。这样，直角坐标和极坐标的数据都可以适配（Adapt）到 complex_struct 结构体中，无须转换和损失精度：

```
enum coordinate_type { RECTANGULAR, POLAR };
struct complex_struct {
        enum coordinate_type t;
        double a, b;
};
```

enum 关键字的作用和 struct 关键字类似，把 coordinate_type 这个标识符定义为一个 Tag，struct complex_struct 表示一个结构体类型，而 enum coordinate_type 表示一个枚举（Enumeration）类型。枚举类型的成员是常量，它们的值由编译器自动分配，例如定义了上面的枚举类型之后，RECTANGULAR 就表示常量 0，POLAR 表示常量 1。如果不希望从 0 开始分配，可以这样定义：

```
enum coordinate_type { RECTANGULAR = 1, POLAR };
```

这样，RECTANGULAR 就表示常量 1，而 POLAR 表示常量 2。枚举常量也是一种整型，其值在编译时确定，因此也可以出现在常量表达式中，可以用于初始化全局变量或者作为 case 分支的判断条件。

有一点需要注意，虽然结构体的成员名和变量名不在同一命名空间中，但枚举的成员名却和变量名在同一命名空间中，所以会出现命名冲突。例如这样是不合法的：

```
int main(void)
{
        enum coordinate_type { RECTANGULAR = 1, POLAR };
        int RECTANGULAR;
        printf("%d %d\n", RECTANGULAR, POLAR);
        return 0;
}
```

complex_struct 结构体的格式变了，就需要修改复数存储表示层的函数，但只要保持函数接口不变就不会影响到上层函数。例如：

```
struct complex_struct make_from_real_img(double x, double y)
{
        struct complex_struct z;
        z.t = RECTANGULAR;
        z.a = x;
        z.b = y;
        return z;
}

struct complex_struct make_from_mag_ang(double r, double A)
{
        struct complex_struct z;
        z.t = POLAR;
        z.a = r;
        z.b = A;
        return z;
}
```

习题

1. 本节只给出了 make_from_real_img 和 make_from_mag_ang 函数的实现，请读者自己实现 real_part、img_part、magnitude、angle 这些函数。

2. 编译运行下面这段程序：

```
#include <stdio.h>

enum coordinate_type { RECTANGULAR = 1, POLAR };

int main(void)
{
        int RECTANGULAR;
        printf("%d %d\n", RECTANGULAR, POLAR);
        return 0;
}
```

结果是什么？并解释一下为什么是这样的结果。

7.4 嵌套结构体

结构体也是一种递归定义：结构体的成员具有某种数据类型，而结构体本身也是一种数据类型。换句话说，结构体的成员可以是另一个结构体，即结构体可以嵌

套定义。例如我们在复数的基础上定义复平面上的线段：

```
struct segment {
    struct complex_struct start;
    struct complex_struct end;
};
```

从第 7.1 节讲的 Initializer 的语法可以看出，Initializer 也可以嵌套，因此嵌套结构体可以嵌套地初始化，例如：

```
struct segment s = {{ 1.0, 2.0 }, { 4.0, 6.0 }};
```

也可以平坦（Flat）地初始化。例如：

```
struct segment s = { 1.0, 2.0, 4.0, 6.0 };
```

甚至可以把两种方式混合使用（这样可读性很差，应该避免）：

```
struct segment s = {{ 1.0, 2.0 }, 4.0, 6.0 };
```

利用 C99 的新特性也可以做 Memberwise Initialization，例如[④]：

```
struct segment s = { .start.x = 1.0, .end.x = 2.0 };
```

访问嵌套结构体的成员要用到多个.后缀运算符，例如：

```
s.start.t = RECTANGULAR;
s.start.a = 1.0;
s.start.b = 2.0;
```


④ 为了便于理解，第 7.1 节讲的 Initializer 语法并没有描述这种复杂的用法。

第 8 章
数组

8.1 数组的基本概念

数组（Array）也是一种复合数据类型，它由一系列相同类型的元素（Element）组成。例如定义一个由 4 个 int 型元素组成的数组 count：

```
int count[4];
```

和结构体成员类似，数组 count 的 4 个元素的存储空间也是相邻的。结构体成员可以是基本数据类型，也可以是复合数据类型，数组中的元素也是如此。根据组合规则，我们可以定义一个由 4 个结构体元素组成的数组：

```
struct complex_struct {
        double x, y;
} a[4];
```

也可以定义一个包含数组成员的结构体：

```
struct {
        double x, y;
        int count[4];
} s;
```

数组类型的长度应该用一个整数常量表达式来指定[①]。数组中的元素通过下标（或者叫索引，Index）来访问。例如前面定义的由 4 个 int 型元素组成的数组 count 图示如图 8.1 所示。

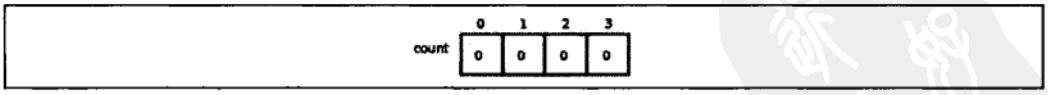


图 8.1 数组 count

整个数组占了 4 个 int 型的存储单元，存储单元用小方框表示，里面的数字是存储在这个单元中的数据（假设都是 0），而框外面的数字是下标，这 4 个单元分

① C99 的新特性允许在数组长度表达式中使用变量，称为变长数组（Variable Length Array，VLA），VLA 只能定义为局部变量而不能是全局变量，与 VLA 有关的语法规则比较复杂，本书不做详细介绍。

别用count[0]、count[1]、count[2]、count[3]来访问。注意，在定义数组int count[4];时，方括号（Bracket）中的数字4表示数组的长度，而在访问数组时，方括号中的数字表示访问数组的第几个元素。和我们平常数数不同，数组元素是从“第0个”开始数的，大多数编程语言都是这么规定的，所以计算机术语中有Zeroth这个词。这样规定使得访问数组元素非常方便，比如count数组中的每个元素占4个字节，则count[0]位于数组开头，而count[i]表示从数组开头跳过4×i个字节之后的那个存储单元。这种数组下标的表达式不仅可以表示存储单元中的值，也可以表示存储单元本身，也就是说可以做左值，因此以下语句都是正确的：

```
count[0] = 7;
count[1] = count[0] * 2;
++count[2];
```

到目前为止我们学习了五种后缀运算符：后缀++、后缀--、结构体取成员.、数组取下标[]、函数调用()。还学习了五种单目运算符（或者叫前缀运算符）：前缀++、前缀--、正号+、负号-、逻辑非!。在C语言中后缀运算符的优先级最高，单目运算符的优先级仅次于后缀运算符，比其他运算符的优先级都高，所以上面举例的++count[2]应该看作对count[2]做前缀++运算。

数组下标也可以是表达式，但表达式的值必须是整型的。例如：

```
int i = 10;
count[i] = count[i+1];
```

使用数组下标不能超出数组的长度范围，这一点在使用变量做数组下标时尤其要注意。C编译器并不检查count[-1]或是count[100]这样的访问越界错误，编译时能顺利通过，所以属于运行时错误[②]。但有时候这种错误很隐蔽，发生访问越界时程序可能并不会立即崩溃，而执行到后面某个正确的语句时却有可能突然崩溃（在第10.4节我们会看到这样的例子）。所以从一开始写代码时就要小心避免出问题，事后依靠调试来解决问题的成本是很高的。

数组也可以像结构体一样初始化，未赋初值的元素也是用0来初始化，例如：

```
int count[4] = { 3, 2, };
```

则count[0]等于3， count[1]等于2，后面两个元素等于0。如果定义数组的同时初始化它，也可以不指定数组的长度，例如：

```
int count[] = { 3, 2, 1, };
```

② 你可能会想为什么编译器对这么明显的错误都视而不见？理由一，这种错误并不总是显而易见的，在第22章会讲到通过指针而不是数组名来访问数组的情况，指针指向数组中的什么位置只有运行时才知道，编译时无法检查是否越界，而运行时每次访问数组元素都检查越界会严重影响性能，所以干脆不检查了；理由二，参考文献[6]的第0章指出，C语言的设计精神是：相信每个C程序员都是高手，不要阻止程序员去干他们需要干的事，高手们使用count[-1]这种技巧其实并不少见，不应该当做错误。

编译器会根据 Initializer 有三个元素确定数组的长度为 3。利用 C99 的新特性也可以做 Memberwise Initialization：

```
int count[4] = { [2] = 3 };
```

下面举一个完整的例子：

例 8.1 定义和访问数组

```
#include <stdio.h>

int main(void)
{
    int count[4] = { 3, 2, }, i;

    for (i = 0; i < 4; i++)
        printf("count[%d]=%d\n", i, count[i]);
    return 0;
}
```

这个例子通过循环把数组中的每个元素依次访问一遍，在计算机术语中称为遍历（Traversal）。注意控制表达式 i < 4，如果写成 i <= 4 就错了，因为 count[4]是访问越界。

数组和结构体虽然有很多相似之处，但也有一个显著的不同：**数组不能相互赋值或初始化**。比如：

```
int a[5] = { 4, 3, 2, 1 };
int b[5] = a;
```

用数组 a 来初始化数组 b 是错的。再比如：

```
a = b;
```

用数组 b 给数组 a 赋值也是错的。

既然不能相互赋值，也就**不能用数组类型作为函数的参数或返回值**。如果写出这样的函数定义：

```
void foo(int a[5])
{
        ...
}
```

然后这样调用：

```
int array[5] = {0};
foo(array);
```

编译器也不会报错，但这样写并不是传一个数组类型参数的意思。对于数组类型有一条特殊的类型转换规则：**数组类型做右值使用时，自动转换成指向数组**

首元素的指针。对于函数声明也有一条特殊规则：**在函数原型中，如果参数写成数组的形式，则该参数实际上是指针类型**。所以上面的函数调用其实是传一个指针类型的参数，而不是数组类型的参数。接下来的几章里有的函数需要访问数组，我们就把数组定义为全局变量给函数访问，等到第22章讲了指针再使用传参的办法。

数组类型不能相互赋值或初始化也是因为这条规则，例如上面提到的a = b这个表达式，a和b都是数组类型的变量，但是b做右值使用，自动转换成指针类型，而左边仍然是数组类型，所以编译器报的错是error: incompatible types in assignment。

习题

1. 编写一个程序，定义两个类型和长度都相同的数组，将其中一个数组的所有元素拷贝给另一个。既然数组不能直接赋值，想想应该怎么实现。

8.2 数组应用实例：统计随机数

本节通过一个实例介绍使用数组的一些基本模式。问题是这样的：首先生成一列0～9的随机数保存在数组中，然后统计其中每个数字出现的次数并打印，检查这些数字的随机性如何。随机数在某些场合（例如游戏程序）是非常有用的，但是用计算机生成完全随机的数却不是那么容易。计算机执行每一条指令的结果都是确定的，没有一条指令产生的是随机数，调用C标准库函数得到的随机数其实是伪随机（Pseudorandom）数，是用数学公式算出来的确定的数，只不过这些数看起来很随机，并且从统计意义上也很接近于均匀分布（Uniform Distribution）的随机数。

C标准库中生成伪随机数的是rand函数，使用这个函数需要包含头文件stdlib.h，它没有参数，返回值是一个介于0和RAND_MAX之间的接近均匀分布的整数。RAND_MAX是该头文件中定义的一个常量，在不同的平台上有不同的取值，但可以肯定它是一个非常大的整数。通常我们用到的随机数是限定在某个范围之中的，例如0～9，而不是0～RAND_MAX，我们可以用%运算符将rand函数的返回值处理一下：

```
int x = rand() % 10;
```

完整的程序如下：

例8.2 生成并打印随机数

```
#include <stdio.h>
#include <stdlib.h>
#define N 20

int a[N];

void gen_random(int upper_bound)
```

```
{
        int i;
        for (i = 0; i < N; i++)
                a[i] = rand() % upper_bound;
}

void print_random()
{
        int i;
        for (i = 0; i < N; i++)
                printf("%d ", a[i]);
        printf("\n");
}

int main(void)
{
        gen_random(10);
        print_random();
        return 0;
}
```

这里介绍一种新的语法：用#define 定义一个常量。实际上编译器的工作分为两个阶段，先是预处理（Preprocess）阶段，然后才是编译阶段，用 gcc 的-E 选项可以看到预处理之后、编译之前的程序，例如：

```
$ gcc -E main.c
...（这里省略了很多行 stdio.h 和 stdlib.h 的代码）
int a[20];

void gen_random(int upper_bound)
{
 int i;
 for (i = 0; i < 20; i++)
  a[i] = rand() % upper_bound;
}

void print_random()
{
 int i;
 for (i = 0; i < 20; i++)
  printf("%d ", a[i]);
 printf("\n");
}

int main(void)
{
 gen_random(10);
 print_random();
 return 0;
}
```

可见在这里预处理器做了两件事情，一是把头文件 stdio.h 和 stdlib.h 在代码中展开，二是把#define 定义的标识符 N 替换成它的定义 20（在代码中做了三处替换，分别位于数组的定义中和两个函数中）。像#include 和#define 这种以#号开头的行称为预处理指示（Preprocessing Directive），我们将在第 20 章学习其他预处理指示。此外，用 cpp main.c 命令也可以达到同样的效果，只做预处理而不编译，cpp

表示 C preprocessor。

那么用#define 定义的常量和第 7.3 节讲的枚举常量有什么区别呢？首先，define 不仅用于定义常量，也可以定义更复杂的语法结构，称为宏（Macro）定义。其次，define 定义是在预处理阶段处理的，而枚举是在编译阶段处理的。试试看把第 7.3 节习题 2 的程序改成下面这样是什么结果。

```
#include <stdio.h>
#define RECTANGULAR 1
#define POLAR 2

int main(void)
{
    int RECTANGULAR;
    printf("%d %d\n", RECTANGULAR, POLAR);
    return 0;
}
```

注意，虽然 include 和 define 在预处理指示中有特殊含义，但它们并不是 C 语言的关键字，换句话说，它们也可以用作标识符，例如声明 int include;或者 void define(int);。在预处理阶段，如果一行以#号开头，后面跟 include 或 define，预处理器就认为这是一条预处理指示，除此之外出现在其他地方的 include 或 define 预处理器并不关心，只当成普通的标识符交给编译阶段去处理。

回到随机数这个程序继续讨论，一开始为了便于分析和调试，我们取小一点的数组长度，只生成 20 个随机数，这个程序的运行结果为：

```
3 6 7 5 3 5 6 2 9 1 2 7 0 9 3 6 0 6 2 6
```

看起来很随机了。但随机性如何呢？分布得均匀吗？所谓均匀分布，应该每个数出现的概率是一样的。在上面的 20 个结果中，6 出现了 5 次，而 4 和 8 一次也没出现过。但这说明不了什么问题，毕竟我们的样本太少了，才 20 个数，如果样本足够多，比如说 100000 个数，统计一下其中每个数字出现的次数也许能说明问题。但总不能把 100000 个数都打印出来然后挨个去数吧？我们需要写一个函数统计每个数字出现的次数。完整的程序如下：

例 8.3 统计随机数的分布

```
#include <stdio.h>
#include <stdlib.h>
#define N 100000

int a[N];

void gen_random(int upper_bound)
{
        int i;
        for (i = 0; i < N; i++)
                a[i] = rand() % upper_bound;
}

int howmany(int value)
```

```
{
        int count = 0, i;
        for (i = 0; i < N; i++)
                if (a[i] == value)
                        ++count;
        return count;
}

int main(void)
{
        int i;

        gen_random(10);
        printf("value\thow many\n");
        for (i = 0; i < 10; i++)
                printf("%d\t%d\n", i, howmany(i));

        return 0;
}
```

我们只要把#define N 的值改为 100000，就相当于把整个程序中所有用到 N 的地方都改为 100000 了。如果我们不这么写，而是在定义数组时直接写成 int a[20];，在每个循环中也直接使用 20 这个值，这称为硬编码（Hard coding）。如果原来的代码是硬编码的，那么一旦需要把 20 改成 100000 就非常麻烦，你需要找遍整个代码，判断哪些 20 表示这个数组的长度就改为 100000，哪些 20 表示别的数量则不做改动，如果代码很长，这是很容易出错的。所以，**写代码时应尽可能避免硬编码**，这其实也是一个“提取公因式”的过程，和第 7.2 节讲的抽象具有相同的作用，就是避免一个地方的改动波及较大的范围。这个程序的运行结果如下：

```
$ ./a.out
value   how many
0   10130
1   10072
2   9990
3   9842
4   10174
5   9930
6   10059
7   9954
8   9891
9   9958
```

各数字出现的次数都在 10000 次左右，可见是比较均匀的。

习题

1. 用 rand 函数生成[10, 20]之间的随机整数，表达式应该怎么写？

8.3 数组应用实例：直方图

继续上面的例子。我们统计一列 0～9 的随机数，打印每个数字出现的次数，像这样的统计结果称为直方图（Histogram）。有时候我们并不只是想打印，更想把统

计结果保存下来以便做后续处理。我们可以把程序改成这样：

```
int main(void)
{
        int howmanyones = howmany(1);
        int howmanytwos = howmany(2);
        ...
}
```

这显然太繁琐了。要是这样的随机数有 100 个呢？显然这里用数组最合适不过了：

```
int main(void)
{
        int i, histogram[10];

        gen_random(10);
        for (i = 0; i < 10; i++)
                histogram[i] = howmany(i);
        ...
}
```

有意思的是，这里的循环变量 i 有两个作用，一是作为参数传给 howmany 函数，统计数字 i 出现的次数，二是做 histogram 的下标，也就是“把数字 i 出现的次数保存在数组 histogram 的第 i 个位置”。

尽管上面的方法可以准确地得到统计结果，但是效率很低，这 100000 个随机数需要从头到尾检查十遍，每一遍检查只统计一种数字的出现次数。其实可以把 histogram 中的元素当做累加器来用，这些随机数只需要从头到尾检查一遍（Single Pass）就可以得出结果：

```
int main(void)
{
        int i, histogram[10] = {0};

        gen_random(10);
        for (i = 0; i < N; i++)
                histogram[a[i]]++;
        ...
}
```

首先把 histogram 的所有元素初始化为 0，注意使用局部变量的值之前一定要初始化，否则值是不确定的。接下来的代码很有意思，在每次循环中，a[i]就是出现的随机数，而这个随机数同时也是 histogram 的下标，这个随机数每出现一次就把 histogram 中相应的元素加 1。

把上面的程序运行几遍，你就会发现每次产生的随机数都是一样的，不仅如此，在别的计算机上运行该程序产生的随机数很可能也是这样的。这正说明了这些数是伪随机数，是用一套确定的公式基于某个初值算出来的，只要初值相同，随后的整个数列就都相同。实际应用中不可能使用每次都一样的随机数，例如开发一个麻将游戏，每次运行这个游戏摸到的牌不应该是一样的。因此，C 标准库允许我们自己指定一个初值，然后在此基础上生成伪随机数，这个初值称为 Seed，可以用 srand 函数指定 Seed。通常我们通过别的途径得到一个不确定的数作为 Seed，

例如调用 time 函数得到当前系统时间距 1970 年 1 月 1 日 00:00:00[③]的秒数，然后传给 srand：

```
srand(time(NULL));
```

然后再调用 rand，得到的随机数就和刚才完全不同了。调用 time 函数需要包含头文件 time.h，这里的 NULL 表示空指针，到第 22.1 节再详细解释。

习题

1. 补完本节直方图程序的 main 函数，以可视化的形式打印直方图。例如上一节统计 20 个随机数的结果是：

```
0  1  2  3  4  5  6  7  8  9

*  *  *  *     *  *  *     *
*     *  *     *  *  *     *
      *  *        *
                  *
                  *
```

2. 定义一个数组，编程打印它的全排列。比如定义：

```
#define N 3
int a[N] = { 1, 2, 3 };
```

则运行结果是：

```
$ ./a.out
1 2 3
1 3 2
2 1 3
2 3 1
3 2 1
3 1 2
```

程序的主要思路是：

1. 把第 1 个数换到最前面来（本来就在最前面），准备打印 1xx，再对后两个数 2 和 3 做全排列。
2. 把第 2 个数换到最前面来，准备打印 2xx，再对后两个数 1 和 3 做全排列。
3. 把第 3 个数换到最前面来，准备打印 3xx，再对后两个数 1 和 2 做全排列。

可见这是一个递归的过程，把对整个序列做全排列的问题归结为对它的子序列做全排列的问题，注意我没有描述 Base Case 怎么处理，你需要自己想。你的程序要具有通用性，如果改变了 N 和数组 a 的定义（比如改成 4 个数的数组），其他代码不需要修改就可以做 4 个数的全排列（共 24 种排列）。

③ 各种派生自 UNIX 的系统都把这个时刻称为 Epoch，因为 UNIX 系统最早发明于 1969 年。

完成了上述要求之后再考虑第二个问题：如果再定义一个常量 M 表示从 N 个数中取几个数做排列（N == M 时表示全排列），原来的程序应该怎么改？

最后再考虑第三个问题：如果要求从 N 个数中取 M 个数做组合而不是做排列，就不能用原来的递归过程了，想想组合的递归过程应该怎么描述，编程实现它。

8.4 字符串

之前我一直对字符串避而不谈，不做详细解释，现在已经具备了必要的基础知识，可以深入讨论一下字符串了。字符串字面值和数组类型相似，它的每个元素是字符型的，例如字符串"Hello, world.\n"图示如图 8.2 所示。

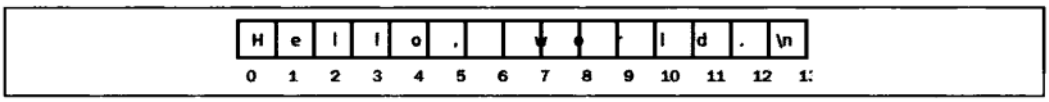


图 8.2 字符串

注意每个字符串末尾都有一个字符'\0'做结束符，这里的\0 是 ASCII 码的八进制表示，也就是 ASCII 码为 0 的 Null 字符，所以字符串也称为“以 Null 结尾的字符串”（Null-terminated String）。

数组元素可以通过数组名加下标的方式访问，而字符串字面值也可以像数组名一样使用，可以加下标访问其中的字符，例如：

```
char c = "Hello, world.\n"[14];
```

把"Hello, world.\n"这个字符串看作一个数组，从图 8.2 可以看出，下标 14 的位置是字符'\0'，所以这个语句把'\0'赋给变量 c。注意，通过下标可以读取字符串字面值中的字符，却不允许修改其中的字符：

```
"Hello, world.\n"[0] = 'A';
```

这行代码会产生编译错误“error: assignment of read-only location”，即字符串字面值所代表的存储空间是只读的，不允许修改。字符串字面值还有一点和数组类型相似，做右值使用时自动转换成指向首元素的指针，在第 3.3 节我们看到 printf 原型的第一个参数是指针类型，而 printf("hello world")其实就是传一个指针参数给 printf，关于字符串字面值和指针的关系将在第 22.4 节详细解释。

字符串字面值有一种特殊用法，前面讲过数组可以像结构体一样初始化，如果是字符数组，也可以用一个字符串字面值来初始化[④]：

④ 这个语法的特殊之处在于，在这里字符串字面值和数组的用法并不相似，我们不能用一个数组给另一个数组初始化，却可以用一个字符串字面值给一个数组初始化。另一方面，我们不能把一个数组赋值给另一个数组，同样也不能把一个字符串字面值赋值给一个数组。这些特殊规定没什么道理可讲，一切都可归结于历史原因。

```
char str[10] = "Hello";
```

相当于：

```
char str[10] = { 'H', 'e', 'l', 'l', 'o', '\0' };
```

str 的后 4 个元素没有指定，自动初始化为'\0'，即 Null 字符。注意，虽然字符串字面值"Hello"是只读的，但用它初始化的数组 str 却是可读可写的。数组 str 中保存了一串字符，以 Null 字符结尾，也可以叫字符串。**在本书中只要是以 Null 结尾的一串字符都叫字符串，不管是像 str 这样的字符数组，还是像"Hello"这样的字符串字面值。**

如果用于初始化的字符串字面值比数组还长，比如：

```
char str[10] = "Hello, world.\n";
```

则数组 str 只包含字符串的前 10 个字符，不包含 Null 字符，这种情况编译器会给出警告。如果要用一个字符串字面值准确地初始化一个字符数组，最好的办法是不指定数组的长度，让编译器自己计算：

```
char str[] = "Hello, world.\n";
```

字符串字面值的长度包括 Null 字符在内一共 15 个字符，编译器会确定数组 str 的长度为 15。

有一种情况需要特别注意，如果用于初始化的字符串字面值比数组刚好长出一个 Null 字符的长度，比如：

```
char str[14] = "Hello, world.\n";
```

则数组 str 不包含 Null 字符，并且编译器不会给出警告，参考文献[6]的 6.7.8 节说这样规定是为程序员方便，以前的很多编译器都是这样实现的，不管它有理没理，C 标准既然这么规定了我们也没办法，只能自己小心了。

补充一点，printf 函数的格式化字符串中可以用%s 表示字符串的占位符。在学习字符数组以前，我们用%s 没什么意义，因为：

```
printf("string: %s\n", "Hello");
```

还不如写成：

```
printf("string: Hello\n");
```

但现在字符串可以保存在一个数组里面，用%s 来打印就很有必要了：

```
printf("string: %s\n", str);
```

printf会从数组str的开头一直打印到Null字符为止，Null字符本身是Non-printable字符，不打印。这其实是一个危险的信号：如果数组 str 中没有 Null 字符，那么

printf 函数就会访问数组越界，后果可能会很诡异，有时候打印出乱码，有时候看起来没错误，有时候引起程序崩溃。

8.5 多维数组

就像结构体可以嵌套一样，数组也可以嵌套，一个数组的元素可以是另外一个数组，这样就构成了多维数组（Multi-dimensional Array）。例如定义并初始化一个二维数组：

```
int a[3][2] = { 1, 2, 3, 4, 5 };
```

数组 a 有 3 个元素，a[0]、a[1]、a[2]。每个元素也是一个数组，例如 a[0]是一个数组，它有两个元素 a[0][0]、a[0][1]，这两个元素的类型是 int，值分别是 1、2，同理，数组 a[1]的两个元素是 3、4，数组 a[2]的两个元素是 5、0，如图 8.3 所示。

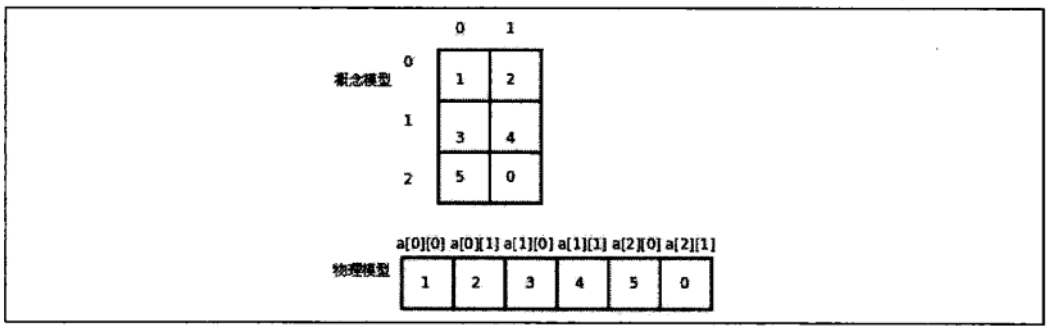


图 8.3 多维数组

从概念模型上看，这个二维数组是三行两列的表格，元素的两个下标分别是行号和列号。从物理模型上看，这六个元素在存储器中仍然是连续存储的，就像一维数组一样，相当于把概念模型的表格一行一行接起来拼成一串，C 语言的这种存储方式称为 Row-major 方式，而有些编程语言（例如 FORTRAN）是把概念模型的表格一列一列接起来拼成一串存储的，称为 Column-major 方式。

多维数组也可以像嵌套结构体一样用嵌套 Initializer 初始化，例如上面的二维数组也可以这样初始化：

```
int a[3][2] = { { 1, 2 }, { 3, 4 }, { 5, } };
```

利用 C99 的新特性也可以做 Memberwise Initialization，例如：

```
int a[3][2] = { [0][1] = 9, [2][1] = 8 };
```

结构体和数组嵌套的情况也可以做 Memberwise Initialization，例如：

```
struct complex_struct {
        double x, y;
} a[4] = { [0].x = 8.0 };

struct {
```

```
        double x, y;
        int count[4];
} s = { .count[2] = 9 };
```

如果是多维字符数组，也可以嵌套使用字符串字面值做 Initializer，例如：

例 8.4 多维字符数组

```
#include <stdio.h>

void print_day(int day)
{
        char days[8][10] = { "", "Monday", "Tuesday",
                             "Wednesday", "Thursday", "Friday",
                             "Saturday", "Sunday" };

        if (day >= 1 && day <= 7)
                printf("%s\n", days[day]);
        else
                printf("Illegal day number!\n");
}

int main(void)
{
        rint_day(2);
        eturn 0;
}
```

这个程序中定义了一个多维字符数组 char days[8][10];，如图 8.4 所示。为了使 1～7 刚好映射到 days[1]～days[7]，我们把 days[0]空出来不用，所以第一维的长度是 8，为了使最长的字符串"Wednesday"能够保存到一行，末尾还能多出一个 Null 字符的位置，所以第二维的长度是 10。

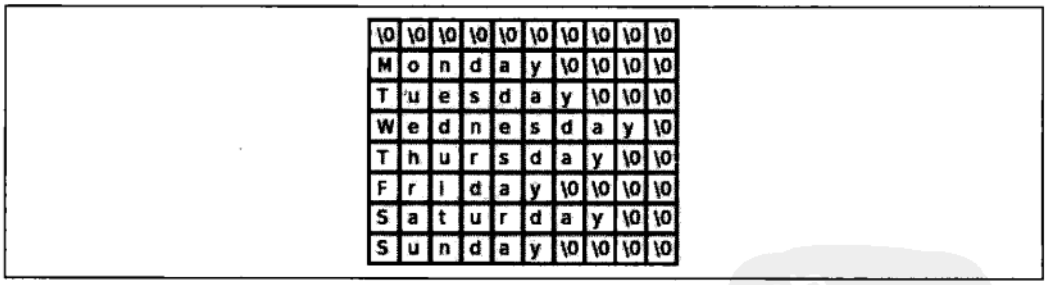


图 8.4 多维字符数组

这个程序和例 4.1 的功能其实是一样的，但是代码简洁多了。简洁的代码不仅可读性强，而且维护成本也低，像例 4.1 那样一堆 case、printf 和 break，如果漏写一个 break 就要出 Bug。这个程序之所以简洁，是因为用数据代替了代码。具体来说，通过下标访问字符串组成的数组可以代替一堆 case 分支判断，这样就可以把每个 case 里重复的代码（printf 调用）提取出来，从而又一次达到了“提取公因式”的效果。这种方法称为数据驱动的编程（Data-driven Programming），写代码最重要的是选择正确的数据结构来组织信息，设计控制流程和算法尚在其次，只要数据结构选择得正确，其他代码自然而然就变得容易理解和维护了，就像这

里的 printf 自然而然就被提取出来了。参考文献[13]的第 9 章说：“Show me your flowcharts and conceal your tables, and I shall continue to be mystified. Show me your tables, and I won't usually need your flowcharts; they'll be obvious.”

最后，综合本章的知识，我们来写一个最简单的小游戏——剪刀石头布：

例 8.5 剪刀石头布

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(void)
{
        char gesture[3][10] = { "scissor", "stone", "cloth" };
        int man, computer, result, ret;

        srand(time(NULL));
        while (1) {
                computer = rand() % 3;
                printf("\nInput your gesture (0-scissor 1-stone "
                      "2-cloth):\n");
                ret = scanf("%d", &man);
                if (ret != 1 || man < 0 || man > 2) {
                        printf("Invalid input!\n");
                        return 1;
                }
                printf("You: %s\tComputer: %s\n",
                      gesture[man], gesture[computer]);

                result = (man - computer + 4) % 3 - 1;
                if (result > 0)
                        printf("You win!\n");
                else if (result == 0)
                        printf("Draw!\n");
                else
                        printf("You lose!\n");
        }
        return 0;
}
```

0、1、2 三个整数分别是剪刀、石头、布在程序中的内部表示，用户也要求输入 0、1 或 2，然后和计算机随机生成的 0、1 或 2 比胜负。这个程序的主体是一个死循环，需要按 Ctrl+C 组合键退出程序。以往我们写的程序都只有打印输出，在这个程序中我们第一次碰到处理用户输入的情况。我们简单介绍一下 scanf 函数的用法，到第 24.2.9 节再详细解释。scanf("%d", &man)这个调用的功能是等待用户输入一个整数并回车，这个整数会被 scanf 函数保存在 man 这个整型变量里。如果用户输入合法（输入的确实是数字而不是别的字符），则 scanf 函数返回 1，表示成功读入一个数据。但即使用户输入的是整数，我们还需要进一步检查是不是在 0～2 的范围内，写程序时对用户输入要格外小心，用户有可能输入任何数据，他才不管游戏规则是什么。

和 printf 类似，scanf 也可以用%c、%f、%s 等转换说明。如果在传给 scanf 的第

一个参数中用%d、%f 或%c 表示读入一个整数、浮点数或字符，则第二个参数的形式应该是&运算符加相应类型的变量名，表示读进来的数保存到这个变量中，&运算符的作用是得到一个指针类型，到第 22.1 节再详细解释；如果在第一个参数中用%s 读入一个字符串，则第二个参数应该是数组名，数组名前面不加&，因为数组类型做右值时自动转换成指针类型，在第 10.2 节有 scanf 读入字符串的例子。

留给读者思考的问题是：(man - computer + 4) % 3 - 1 这个神奇的表达式是如何比较出 0、1、2 这三个数字在“剪刀石头布”意义上的大小的？

第 9 章

编码风格

代码风格好不好就像字写得好不好看一样，如果一个公司招聘秘书，肯定不要字写得难看的。同理，代码风格糟糕的程序员肯定也是不称职的。虽然编译器不会挑剔难看的代码，照样能编译通过，但是和你一个 Team 的其他程序员肯定受不了，你自己也受不了，写完代码几天之后再来看，自己都不知道自己写的是什么。参考文献[12]的第一版前言里有句话说得好："Thus, programs must be written for people to read, and only incidentally for machines to execute."代码主要是为了写给人看的，而不是写给机器看的，只是顺便也能用机器执行而已，如果是为了写给机器看那直接写机器指令就好了，没必要用高级语言了。代码和语言文字一样是为了表达思想、记载信息，所以一定要写得清楚整洁才能有效地表达。正因为如此，在一个软件项目中，代码风格一般都用文档规定死了，所有参与项目的人不管他自己原来是什么风格，都要遵守统一的风格，例如 Linux 内核的参考文献[14]就是这样一个文档。本章我们以内核的代码风格为基础来讲解好的编码风格都有哪些规定，这些规定的 Rationale 是什么。我只是以 Linux 内核为例来讲解编码风格的概念，并没有说内核编码风格就一定是最好的编码风格，但 Linux 内核项目如此成功，就足以说明它的编码风格是最好的 C 语言编码风格之一了。

9.1 缩进和空白

我们知道 C 语言的语法对缩进和空白没有要求，空格、Tab、换行都可以随意写，实现同样功能的代码可以写得很好看，也可以写得很难看。例如上一章例 8.5 的代码如果写成这样就很难看了：

例 9.1 缺少缩进和空白的代码

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main(void)
{
char gesture[3][10]={"scissor","stone","cloth"};
int man,computer,result, ret;
srand(time(NULL));
while(1){
computer=rand()%3;
printf("\nInput your gesture (0-scissor 1-stone 2-cloth):\n");
ret=scanf("%d",&man);
```

```
    if(ret!=1||man<0||man>2){
    printf("Invalid input!\n");
    return 1;
    }
    printf("You: %s\tComputer: %s\n",gesture[man],gesture[computer]);
    result=(man-computer+4)%3-1;
    if(result>0)printf("You win!\n");
    else if(result==0)printf("Draw!\n");
    else printf("You lose!\n");
    }
    return 0;
    }
```

一是缺少空白字符，代码密度太大，看着很费劲。二是没有缩进，看不出来哪个{和哪个}配对，像这么短的代码还能凑合着看，如果代码超过一屏就完全没法看了。参考文献[14]中关于空白字符并没有特别规定，因为基本上所有的 C 代码风格对于空白字符的规定都差不多，主要有以下几条。

1．关键字 if、while、for 与其后的控制表达式的(括号之间插入一个空格分隔，但括号内的表达式应紧贴括号。例如：

```
    while␣(1);
```

2．双目运算符的两侧各插入一个空格分隔，单目运算符和操作数之间不加空格，例如 i␣=␣i␣+␣1、++i、!(i␣<␣1)、-x、&a[1]等。

3．后缀运算符和操作数之间也不加空格，例如取结构体成员 s.a、函数调用 foo(arg1)、取数组成员 a[i]。

4．,号和;号之后要加空格，这是英文的书写习惯，例如 for␣(i␣=␣1;␣i␣<␣10;␣i++)、foo(arg1,␣arg2)。

5．以上关于加空格的规则并没有严格要求，有时为了突出优先级也可以写得更紧凑一些，例如 for␣(i=1;␣i<10;␣i++)、distance␣=␣sqrt(x*x␣+␣y*y)等。但是省略的空格一定不要误导了读代码的人，例如 a||b␣&&␣c 很容易让人理解成错误的优先级。

6．由于 UNIX 系统标准的字符终端是 24 行 80 列的，接近或大于 80 个字符的较长语句或声明要折行写，折行后用空格和上面的表达式或参数对齐，例如：

```
    if␣(sqrt(x*x␣+␣y*y)␣>␣5.0
    &&␣x␣<␣0.0
    &&␣y␣>␣0.0)
```

再比如：

```
    foo(sqrt(x*x␣+␣y*y),
        a[i-1]␣+␣b[i-1]␣+␣c[i-1])
```

7．较长的字符串可以断成多个字符串然后分行书写，例如：

```
    printf("This is such a long sentence that "
```

```
        "it cannot be held within a line\n");
```

C编译器会自动把相邻的多个字符串接在一起，以上两个字符串相当于一个字符串"This is such a long sentence that it cannot be held within a line\n"。注意这个语法有时候会带来一点麻烦，比如下面这段代码在语法上没有问题，但在语义上有问题，你能看出是什么问题吗？

```
    char days[8][20] = { "", "Monday", "Tuesday"
                "Wednesday", "Thursday", "Friday"
                "Saturday", "Sunday" };
```

8．有的人喜欢在变量定义中使用Tab字符，使变量名对齐，这样看起来很美观。

```
        →int    →a, b;
        →double →c;
```

内核代码风格关于缩进的规则有以下几条。

1．要用缩进体现出语句块的层次关系，使用Tab字符缩进，不能用空格代替Tab。在标准的字符终端上一个Tab看起来是8个空格的宽度，如果你的文本编辑器可以设置Tab的显示宽度是几个空格，建议也设成8，这样大的缩进使代码看起来非常清晰。如果有的行用空格做缩进，有的行用Tab做缩进，甚至空格和Tab混用，那么一旦改变了文本编辑器的Tab显示宽度就会看起来非常混乱，所以内核代码风格规定只能用Tab做缩进，不能用空格代替Tab。

2．if/else、while、do/while、for、switch这些可以带语句块的语句，语句块的{或}应该和关键字写在同一行，用空格隔开，而不是单独占一行。这个规定和参考文献[3]的代码风格一致，好处是不必占太多行，使得一屏能显示更多代码。例如应该这样写：

```
    if␣(...)␣{
        →语句列表
    }␣else␣if␣(...)␣{
        →语句列表
    }
```

但很多人还是习惯这样写：

```
    if␣(...)
    {
        →语句列表
    }
    else␣if␣(...)
    {
        →语句列表
    }
```

这两种写法用得都很广泛，只要在同一个项目中能保持统一就可以了。

3．函数定义的{和}单独占一行，这一点和语句块的规定不同，例如：

```
int␣foo(int␣a,␣int␣b)
{
      →语句列表
}
```

4. switch 和语句块里的 case、default 对齐写，也就是说语句块里的 case、default 标号相对于 switch 不往里缩进，但标号下的语句要往里缩进。例如：

```
      →switch␣(c)␣{
      →case 'A':
      →      →语句列表
      →case 'B':
      →      →语句列表
      →default:
      →      →语句列表
      →}
```

用于 goto 语句的自定义标号应该顶头写不缩进，而不管标号下的语句缩进到第几层。

5. 代码中每个逻辑段落之间应该用一个空行分隔开。例如每个函数定义之间应该插入一个空行，头文件、全局变量定义和函数定义之间也应该插入空行，例如：

```
#include <stdio.h>
#include <stdlib.h>

int g;
double h;

int foo(void)
{
      →语句列表
}

int bar(int a)
{
      →语句列表
}

int main(void)
{
      →语句列表
}
```

6. 如果一个函数的语句列表很长，可以根据相关性分成若干组，用空行分隔，通常把变量定义组成一组，后面加空行，return 语句之前加空行，例如：

```
int main(void)
{
      →int    →a, b;
      →double →c;

      →语句组 1

      →语句组 2
```

```
        →return 0;
    }
```

9.2 注释

单行注释应采用/*␣comment␣*/的形式，用空格把界定符和文字分开。多行注释最常见的是这种形式：

```
    /*
    ␣*␣Multi-line
    ␣*␣comment
    ␣*/
```

也有更花哨的形式：

```
    /*************\
    * Multi-line  *
    * comment     *
    \*************/
```

使用注释的场合主要有以下几种。

1. 整个源文件的顶部注释。说明此模块的相关信息，例如文件名、作者和版本历史等，顶头写不缩进。例如内核源代码目录下的 kernel/sched.c 文件的开头：

```
    /*
     *  kernel/sched.c
     *
     *  Kernel scheduler and related syscalls
     *
     *  Copyright (C) 1991-2002  Linus Torvalds
     *
     *  1996-12-23  Modified by Dave Grothe to fix bugs in semaphores
and
     *              make semaphores SMP safe
     *  1998-11-19  Implemented schedule_timeout() and related stuff
     *              by Andrea Arcangeli
     *  2002-01-04  New ultra-scalable O(1) scheduler by Ingo Molnar:
     *              hybrid priority-list and round-robin design with
     *              an array-switch method of distributing timeslices
     *                    and per-CPU runqueues.  Cleanups and useful
suggestions
     *              by Davide Libenzi, preemptible kernel bits by Robert
Love.
     *  2003-09-03  Interactivity tuning by Con Kolivas.
     *  2004-04-02  Scheduler domains code by Nick Piggin
     */
```

2. 函数注释。说明此函数的功能、参数、返回值、错误码等，写在函数定义上侧，和此函数定义之间不留空行，顶头写不缩进。

3. 相对独立的语句组注释。对这一组语句做特别说明，写在语句组上侧，和此语

句组之间不留空行，与当前语句组的缩进一致。

4. 代码行右侧的简短注释。对当前代码行做特别说明，一般为单行注释，和代码之间至少用一个空格隔开，一个源文件中所有的右侧注释最好能上下对齐。尽管例 2.1 讲过注释可以穿插在一行代码中间，但不建议这么写。内核源代码目录下的 lib/radix-tree.c 文件中的一个函数包含了上述三种注释：

```
/**
 *      radix_tree_insert    -    insert into a radix tree
 *      @root:          radix tree root
 *      @index:         index key
 *      @item:          item to insert
 *
 *      Insert an item into the radix tree at position @index.
 */
int radix_tree_insert(struct radix_tree_root *root,
                      unsigned long index, void *item)
{
        struct radix_tree_node *node = NULL, *slot;
        unsigned int height, shift;
        int offset;
        int error;

        /* Make sure the tree is high enough.  */
        if ((!index && !root->rnode) ||
                        index > radix_tree_maxindex(root->height)) {
                error = radix_tree_extend(root, index);
                if (error)
                        return error;
        }

        slot = root->rnode;
        height = root->height;
        shift = (height-1) * RADIX_TREE_MAP_SHIFT;

        offset = 0;                     /* uninitialised var warning */
        do {
                if (slot == NULL) {
                        /* Have to add a child node.  */
                        if (!(slot = radix_tree_node_alloc(root)))
                                return -ENOMEM;
                        if (node) {
                                node->slots[offset] = slot;
                                node->count++;
                        } else
                                root->rnode = slot;
                }

                /* Go a level down */
                offset = (index >> shift) & RADIX_TREE_MAP_MASK;
                node = slot;
                slot = node->slots[offset];
                shift -= RADIX_TREE_MAP_SHIFT;
                height--;
        } while (height > 0);

        if (slot != NULL)
                return -EEXIST;
```

```
        BUG_ON(!node);
        node->count++;
        node->slots[offset] = item;
        BUG_ON(tag_get(node, 0, offset));
        BUG_ON(tag_get(node, 1, offset));

        return 0;
}
```

参考文献[14]中特别指出，函数内的注释要尽可能少用。写注释主要是为了说明你的代码“能做什么”（比如函数接口定义），而不是为了说明“怎样做”，只要代码写得足够清晰，“怎样做”是一目了然的，如果你需要用注释才能解释清楚，那就表示你的代码可读性很差，除非是特别需要提醒注意的地方才使用函数内注释。

5. 复杂的结构体定义比函数更需要注释。例如内核源代码目录下的kernel/sched.c文件中定义了这样一个结构体：

```
/*
 * This is the main, per-CPU runqueue data structure.
 *
 * Locking rule: those places that want to lock multiple runqueues
 * (such as the load balancing or the thread migration code), lock
 * acquire operations must be ordered by ascending &runqueue.
 */
struct runqueue {
        spinlock_t lock;

        /*
         * nr_running and cpu_load should be in the same cacheline
         because
         * remote CPUs use both these fields when doing load
         calculation.
         */
        unsigned long nr_running;
#ifdef CONFIG_SMP
        unsigned long cpu_load[3];
#endif
        unsigned long long nr_switches;

        /*
         * This is part of a global counter where only the total sum
         * over all CPUs matters. A task can increase this counter
         on
         * one CPU and if it got migrated afterwards it may decrease
         * it on another CPU. Always updated under the runqueue lock:
         */
        unsigned long nr_uninterruptible;

        unsigned long expired_timestamp;
        unsigned long long timestamp_last_tick;
        task_t *curr, *idle;
        struct mm_struct *prev_mm;
        prio_array_t *active, *expired, arrays[2];
        int best_expired_prio;
        atomic_t nr_iowait;
```

```
    #ifdef CONFIG_SMP
           struct sched_domain *sd;

           /* For active balancing */
           int active_balance;
           int push_cpu;

           task_t *migration_thread;
           struct list_head migration_queue;
           int cpu;
    #endif

    #ifdef CONFIG_SCHEDSTATS
           /* latency stats */
           struct sched_info rq_sched_info;

           /* sys_sched_yield() stats */
           unsigned long yld_exp_empty;
           unsigned long yld_act_empty;
           unsigned long yld_both_empty;
           unsigned long yld_cnt;

           /* schedule() stats */
           unsigned long sched_switch;
           unsigned long sched_cnt;
           unsigned long sched_goidle;

           /* try_to_wake_up() stats */
           unsigned long ttwu_cnt;
           unsigned long ttwu_local;
    #endif
    };
```

6. 复杂的宏定义和变量声明也需要注释。例如内核源代码目录下的 include/linux/jiffies.h 文件中的定义：

```
    /* TICK_USEC_TO_NSEC is the time between ticks in nsec assuming real
ACTHZ and  */
    /* a value TUSEC for TICK_USEC (can be set bij adjtimex)              */
    #define TICK_USEC_TO_NSEC(TUSEC) (SH_DIV (TUSEC * USER_HZ * 1000,
    ACTHZ, 8))

    /* some arch's have a small-data section that can be accessed
    register-relative
     * but that can only take up to, say, 4-byte variables. jiffies being
    part of
     * an 8-byte variable may not be correctly accessed unless we force
    the issue
     */
    #define __jiffy_data  __attribute__((section(".data")))

    /*
     * The 64-bit value is not volatile - you MUST NOT read it
     * without sampling the sequence number in xtime_lock.
     * get_jiffies_64() will do this for you as appropriate.
     */
    extern u64 __jiffy_data jiffies_64;
    extern unsigned long volatile __jiffy_data jiffies;
```

9.3 标识符命名

标识符命名应遵循以下原则：

1. 标识符命名要清晰明了，可以使用完整的单词和易于理解的缩写。短的单词可以通过去元音形成缩写，较长的单词可以取单词的头几个字母形成缩写。看别人的代码看多了就可以总结出一些缩写惯例，例如 count 写成 cnt，block 写成 blk，length 写成 len，window 写成 win，message 写成 msg，number 写成 nr，temporary 可以写成 temp，也可以进一步写成 tmp，最有意思的是 internationalization 写成 i18n，词根 trans 经常缩写成 x，例如 transmit 写成 xmt。我就不多举例了，请读者在看代码时自己注意总结和积累。

2. 内核编码风格规定变量、函数和类型采用全小写加下划线的方式命名，常量（比如宏定义和枚举常量）采用全大写加下划线的方式命名，比如上一节举例的函数名 radix_tree_insert、类型名 struct radix_tree_root、常量名 RADIX_TREE_MAP_SHIFT 等。

微软发明了一种变量命名法叫匈牙利命名法（Hungarian Notation），在变量名中用前缀表示类型，例如 iCnt（i 表示 int）、pMsg（p 表示 pointer）、lpszText（lpsz 表示 long pointer to a null-terminated string）等。Linus 在参考文献[14]中毫不客气地讽刺了这种写法："Encoding the type of a function into the name (so-called Hungarian notation) is brain damaged - the compiler knows the types anyway and can check those, and it only confuses the programmer. No wonder MicroSoft makes buggy programs."代码风格本来就是一个很有争议的问题，如果你接受本章介绍的内核编码风格（也是本书所有范例代码的风格），就不要使用大小写混合的变量命名方式[①]，更不要使用匈牙利命名法。

3. 全局变量和全局函数的命名一定要详细，不惜多用几个单词多写几个下划线，例如函数名 radix_tree_insert，因为它们在整个项目的许多源文件中都会用到，必须让使用者明确这个变量或函数是干什么用的。局部变量和只在一个源文件中调用的内部函数的命名可以简略一些，但不能太短。尽量不要使用单个字母做变量名，只有一个例外：用 i、j、k 做循环变量是可以的。

4. 针对中国程序员的一条特别规定：禁止用汉语拼音做标识符，可读性极差。

9.4 函数

每个函数都应该设计得尽可能简单，简单的函数才容易维护。设计函数应遵循以下原则：

1. 实现一个函数只是为了做好一件事情，不要把函数设计成用途广泛、面面俱到

① 大小写混合的命名方式是 Modern C++风格所提倡的，在 C++代码中很普遍，称为 CamelCase），大概是因为有高有低像骆峰一样。

的，这样的函数肯定会超长，而且往往不可重用，维护困难。

2．函数内部的缩进层次不宜过多，一般以少于4层为宜。如果缩进层次太多就说明设计得太复杂了，应考虑分割成更小的函数（Helper Function）来调用。

3．函数不要写得太长，建议在24行的标准终端上不超过两屏，太长会造成阅读困难，如果一个函数超过两屏就应该考虑分割函数了。参考文献[14]中特别说明，如果一个函数在概念上是简单的，只是长度很长，这倒没关系。例如函数由一个大的switch组成，其中有非常多的case，这是可以的，因为各case分支互不影响，整个函数的复杂度只等于其中一个case的复杂度，这种情况很常见，例如TCP的状态机实现。

4．执行函数就是执行一个动作，函数名通常应包含动词，例如get_current、radix_tree_insert。

5．比较重要的函数定义上方必须加注释，说明此函数的功能、参数、返回值、错误码等。

6．另一种度量函数复杂度的办法是看有多少个局部变量，5到10个局部变量已经很多了，再多就很难维护了，应该考虑分割成多个函数。

9.5 indent工具

indent工具可以把代码格式化成某种风格，例如把例9.1格式化成内核编码风格：

```
$ indent -kr -i8 main.c
$ cat main.c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main(void)
{
        char gesture[3][10] = { "scissor", "stone", "cloth" };
        int man, computer, result, ret;
        srand(time(NULL));
        while (1) {
                computer = rand() % 3;
                printf
                   ("\nInput your gesture (0-scissor 1-stone
                   2-cloth):\n");
                ret = scanf("%d", &man);
                if (ret != 1 || man < 0 || man > 2) {
                        printf("Invalid input!\n");
                        return 1;
                }
                printf("You: %s\tComputer: %s\n",
                      gesture[man], gesture[computer]);
                result = (man - computer + 4) % 3 - 1;
                if (result > 0)
                       printf("You win!\n");
                else if (result == 0)
                       printf("Draw!\n");
```

```
                else
                        printf("You lose!\n");
        }
        return 0;
}
```

-kr 选项表示 K&R 风格，-i8 表示缩进 8 个空格的长度。如果没有指定-nut 选项，则每 8 个缩进空格会自动用一个 Tab 代替。注意 indent 命令会直接修改原文件，而不是打印到屏幕上或者输出到另一个文件，这一点和很多 UNIX 命令不同。可以看出，-kr -i8 两个选项格式化出来的代码已经很符合本章介绍的代码风格了，添加了必要的缩进和空白，较长的代码行也会自动折行。美中不足的是没有添加适当的空行，因为 indent 工具也不知道哪几行代码在逻辑上是一组的，空行还是要自己动手添，当然原有的空行肯定不会被 indent 删去的。

如果你采纳本章介绍的内核编码风格，基本上-kr -i8 这两个参数就够用了。indent 工具也有支持其他编码风格的选项，具体请参考 Man Page。有时候 indent 工具的确非常有用，比如某个项目中途决定改变编码风格（这很少见），或者往某个项目中添加的几个代码文件来自另一个编码风格不同的项目，但绝不能因为有了 indent 工具就肆无忌惮，一开始把代码写得乱七八糟，最后再依靠 indent 去清理。

第 10 章
gdb

程序中除了一目了然的 Bug 之外都需要一定的调试手段来分析到底错在哪。到目前为止我们的调试手段只有一种：根据程序执行时的出错现象假设错误原因，然后在代码中适当的位置插入 printf，执行程序并分析打印结果，如果结果和预期的一样，基本上证明了自己假设的错误原因，就可以动手修正 Bug 了，如果结果和预期的不一样，就根据结果做进一步的假设和分析。本章我们介绍一种很强大的调试工具 gdb，可以完全操控程序的运行，使得程序就像你手里的玩具一样，叫它走就走，叫它停就停，并且随时可以查看程序中所有的内部状态，比如各变量的值、传给函数的参数、当前执行的代码行等。掌握了 gdb 的用法之后，调试手段就更加丰富了。但要注意，即使调试手段丰富了，调试的基本思想仍然是“分析现象→假设错误原因→产生新的现象去验证假设”这样一个循环，根据现象如何假设错误原因，以及如何设计新的现象去验证假设，这都需要非常严密的分析和思考，如果因为手里有了强大的工具就滥用而忽略了分析过程，往往会治标不治本地修正 Bug，导致一个错误现象消失了但 Bug 仍然存在，甚至是把程序越改越错。本章通过初学者易犯的几个错误实例来讲解如何使用 gdb 调试程序，在每个实例后面总结一部分常用的 gdb 命令。

10.1 单步执行和跟踪函数调用

看下面的程序：

例 10.1 函数调试实例

```
#include <stdio.h>

int add_range(int low, int high)
{
        int i, sum;
        for (i = low; i <= high; i++)
                sum = sum + i;
        return sum;
}

int main(void)
{
        int result[1000];
        result[0] = add_range(1, 10);
        result[1] = add_range(1, 100);
```

```
        printf("result[0]=%d\nresult[1]=%d\n", result[0],
        result[1]);
        return 0;
}
```

add_range 函数从 low 加到 high，在 main 函数中首先从 1 加到 10，把结果保存下来，然后从 1 加到 100，再把结果保存下来，最后打印的两个结果是：

```
result[0]=55
result[1]=5105
```

第一个结果正确，第二个结果显然不正确[①]，在小学我们就听说过高斯小时候的故事，从 1 加到 100 应该是 5050。一段代码，第一次运行结果是对的，第二次运行却不对，这是很常见的一类错误现象，这种情况一方面要怀疑代码，另一方面更要怀疑数据：第一次和第二次运行的都是同一段代码，如果代码是错的，那第一次的结果为什么能对呢？很可能是第二次运行时相关的状态数据错了，错误的数据导致了错误的结果。在动手调试之前，读者先试试只看代码能不能看出错误原因，只要前面几章学得扎实就应该能看出来。

在编译时要加上-g 选项，生成的可执行文件才能用 gdb 进行源码级调试：

```
$ gcc -g main.c -o main
$ gdb main
GNU gdb (GDB) 7.1-ubuntu
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/
gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show
copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/akaedu/main...done.
(gdb)
```

-g 选项的作用是在可执行文件中加入源代码的信息，比如可执行文件中第几条机器指令对应源代码的第几行，但并不是把整个源文件嵌入到可执行文件中，所以在调试时必须保证 gdb 能找到源文件。gdb 提供一个类似 Shell 的命令行环境，上面的(gdb)就是提示符，在这个提示符下输入 help 可以查看命令的类别：

```
(gdb) help
List of classes of commands:

aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
```

① 本章这些示例程序的错误现象都和编译器、操作系统、库函数的实现有关，也许在你的系统上跑不出这样的结果，那也没关系，重要的是学会本章介绍的思想方法。另外你也可以尝试修改程序，总有办法得到类似的结果，上例中定义了一个很大的数组 result[1000]，修改数组的大小就可能使运行结果有所不同。

```
    data -- Examining data
    files -- Specifying and examining files
    internals -- Maintenance commands
    obscure -- Obscure features
    running -- Running the program
    stack -- Examining the stack
    status -- Status inquiries
    support -- Support facilities
    tracepoints -- Tracing of program execution without stopping the
program
    user-defined -- User-defined commands

    Type "help" followed by a class name for a list of commands in that
class.
    Type "help all" for the list of all commands.
    Type "help" followed by command name for full documentation.
    Type "apropos word" to search for commands related to "word".
    Command name abbreviations are allowed if unambiguous.
```

也可以进一步查看某一类别中有哪些命令，例如查看files类别下有哪些命令可用：

```
    (gdb) help files
    Specifying and examining files.

    List of commands:

    add-symbol-file -- Load symbols from FILE
    add-symbol-file-from-memory -- Load the symbols out of memory from
a dynamically loaded object file
    cd -- Set working directory to DIR for debugger and program being
debugged
    core-file -- Use FILE as core dump for examining memory and registers
    directory -- Add directory DIR to beginning of search path for source
files
    edit -- Edit specified file or function
    exec-file -- Use FILE as program for getting contents of pure memory
    file -- Use FILE as program to be debugged
    forward-search -- Search for regular expression (see regex(3)) from
last line listed
    generate-core-file -- Save a core file with the current state of
the debugged process
    list -- List specified function or line
    ...
```

现在试试用 list 命令从第一行开始列出源代码：

```
    (gdb) list 1
    1       #include <stdio.h>
    2
    3       int add_range(int low, int high)
    4       {
    5               int i, sum;
    6               for (i = low; i <= high; i++)
    7                       sum = sum + i;
    8               return sum;
    9       }
    10
```

一次只列 10 行，如果要从第 11 行开始继续列源代码可以再输入一次

```
(gdb) list
```

也可以什么都不输直接敲回车，gdb 提供了一个很方便的功能，在提示符下直接敲回车表示重复上一条命令。

```
(gdb) （直接回车）
11      int main(void)
12      {
13              int result[1000];
14              result[0] = add_range(1, 10);
15              result[1] = add_range(1, 100);
16              printf("result[0]=%d\nresult[1]=%d\n", result[0],
                result[1]);
17              return 0;
18      }
```

gdb 的很多常用命令有简写形式，例如 list 命令可以写成 l，要列一个函数的源代码也可以用函数名做参数：

```
(gdb) l add_range
1       #include <stdio.h>
2
3       int add_range(int low, int high)
4   {
5               int i, sum;
6               for (i = low; i <= high; i++)
7                       sum = sum + i;
8               return sum;
9       }
10
```

现在退出 gdb 的环境：

```
(gdb) quit
```

我们做一个实验，把源代码改名或移到别处再用 gdb 调试，这样就列不出源代码了：

```
$ mv main.c mian.c
$ gdb main
...
(gdb) l
5       main.c: No such file or directory.
        in main.c
```

可见 gcc 的-g 选项并不是把源代码嵌入到可执行文件中，在调试时也需要源文件。现在把源代码恢复原样，我们继续调试。首先用 start 命令开始执行程序：

```
$ gdb main
...
(gdb) start
Temporary breakpoint 1 at 0x8048415: file main.c, line 14.
Starting program: /home/akaedu/main

Temporary breakpoint 1, main () at main.c:14
14              result[0] = add_range(1, 10);
```

```
(gdb)
```

gdb停在main函数中变量定义之后的第一条语句处等待我们发命令，gdb列出的这条语句是即将执行的下一条语句。我们可以用next命令（简写为n）控制这些语句一条一条地执行：

```
(gdb) n
15              result[1] = add_range(1, 100);
(gdb) （直接回车）
16              printf("result[0]=%d\nresult[1]=%d\n", result[0],
                result[1]);
(gdb) （直接回车）
result[0]=55
result[1]=5105
17              return 0;
```

用n命令依次执行两行赋值语句和一行打印语句，在执行打印语句时结果立刻打出来了，然后停在return语句之前等待我们发命令。虽然我们完全控制了程序的执行，但仍然看不出哪里错了，因为错误不在main函数中而在add_range函数中，现在用start命令重新来过，这次用step命令（简写为s）钻进add_range函数中去跟踪执行：

```
(gdb) start
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Temporary breakpoint 2 at 0x8048415: file main.c, line 14.
Starting program: /home/akaedu/main

Temporary breakpoint 2, main () at main.c:14
14              result[0] = add_range(1, 10);
(gdb) s
add_range (low=1, high=10) at main.c:6
6               for (i = low; i <= high; i++)
```

这次停在了add_range函数中变量定义之后的第一条语句处。在函数中有几种查看状态的办法，backtrace命令（简写为bt）可以查看函数调用的栈帧：

```
(gdb) bt
#0  add_range (low=1, high=10) at main.c:6
#1  0x08048429 in main () at main.c:14
```

可见当前的add_range函数是被main函数调用的，main传进来的参数是low=1, high=10。main函数的栈帧编号为1，add_range的栈帧编号为0。现在可以用info命令（简写为i）查看add_range函数局部变量的值：

```
(gdb) i locals
i = 0
sum = 0
```

如果想查看main函数当前局部变量的值也可以做到，先用frame命令（简写为f）选择1号栈帧然后再查看局部变量：

```
(gdb) f 1
#1  0x08048429 in main () at main.c:14
```

```
14      result[0] = add_range(1, 10);
(gdb) i locals
result = {0 <repeats 517 times>, 1180510, 0, 0, 0, 0, 0, 0, -1207961512,
-1073746824, 1228788, -1073746376,
...
```

注意到 result 数组中很多元素具有杂乱无章的值，我们知道未经初始化的局部变量具有不确定的值。到目前为止一切正常。用 s 或 n 往下走几步，然后用 print 命令（简写为 p）打印出变量 sum 的值：

```
(gdb) s
7                       sum = sum + i;
(gdb) （直接回车）
6               for (i = low; i <= high; i++)
(gdb) （直接回车）
7                       sum = sum + i;
(gdb) （直接回车）
6               for (i = low; i <= high; i++)
(gdb) p sum
$1 = 3
```

第一次循环 i 是 1，第二次循环 i 是 2，加起来是 3，没错。这里的$1 表示 gdb 保存着这些中间结果，$后面的编号会自动增长，在命令中可以用$1、$2、$3 等编号代替相应的值。由于我们本来就知道第一次调用的结果是正确的，再往下跟也没意义了，可以用 finish 命令让程序一直运行到从当前函数返回为止：

```
(gdb) finish
Run till exit from #0  add_range (low=1, high=10) at main.c:6
0x08048429 in main () at main.c:14
14              result[0] = add_range(1, 10);
Value returned is $2 = 55
```

返回值是 55，当前正准备执行赋值操作，用 s 命令赋值，然后查看 result 数组：

```
(gdb) s
15              result[1] = add_range(1, 100);
(gdb) p result
$3 = {55, 0 <repeats 516 times>, 1180510, 0, 0, 0, 0, 0, 0, -1207961512,
-1073746824, 1228788, -1073746376,
...
```

第一个值 55 确实赋给了 result 数组的第 0 个元素。下面用 s 命令进入第二次 add_range 调用，进入之后首先查看参数和局部变量：

```
(gdb) s
add_range (low=1, high=100) at main.c:6
6       for (i = low; i <= high; i++)
(gdb) bt
#0  add_range (low=1, high=100) at main.c:6
#1  0x08048441 in main () at main.c:15
(gdb) i locals
i = 11
sum = 55
```

由于局部变量 i 和 sum 没初始化，所以具有不确定的值，又由于两次调用是挨着

的，i 和 sum 正好取了上次调用时的值，原来这跟例 3.7 是一样的道理，只不过我这次举的例子设法让局部变量 sum 在第一次调用时初值为 0 而第二次调用时初值不为 0。i 的初值不确定没关系，在 for 循环中首先会把 i 赋值为 low，但 sum 如果初值不是 0，累加得到的结果就错了。好了，我们已经找到错误原因，可以退出 gdb 修改源代码了。如果我们不想浪费这次调试机会，可以在 gdb 中马上把 sum 的初值改为 0 继续运行，看看这一处改了之后还有没有别的 Bug：

```
(gdb) set var sum=0
(gdb) finish
Run till exit from #0  add_range (low=1, high=100) at main.c:6
0x08048441 in main () at main.c:15
15              result[1] = add_range(1, 100);
Value returned is $4 = 5050
(gdb) n
16              printf("result[0]=%d\nresult[1]=%d\n", result[0],
                result[1]);
(gdb) （直接回车）
result[0]=55
result[1]=5050
17              return 0;
```

这样结果就对了。修改变量的值除了用 set 命令之外也可以用 print 命令，因为 print 命令后面跟的是表达式，而我们知道赋值和函数调用也都是表达式，所以也可以用 print 命令修改变量的值或者调用函数：

```
(gdb) p result[2]=33
$5 = 33
(gdb) p printf("result[2]=%d\n", result[2])
result[2]=33
$6 = 13
```

我们讲过，printf 的返回值表示实际打印的字符数，所以$6 的结果是 13。总结一下本节用到的 gdb 命令，如表 10.1 所示。

表 10.1 gdb 基本命令 1

命令	描述
backtrace（或 bt）	查看各级函数调用及参数
finish	连续运行到当前函数返回为止，然后停下来等待命令
frame（或 f） 帧编号	选择栈帧
info（或 i） locals	查看当前栈帧局部变量的值
list（或 l）	列出源代码，接着上次的位置往下列，每次列 10 行
list 行号	列出从第几行开始的源代码
list 函数名	列出某个函数的源代码
next（或 n）	执行下一行语句
print（或 p）	打印表达式的值，通过表达式可以修改变量的值或者调用函数
quit（或 q）	退出 gdb 调试环境
set var	修改变量的值

续表

命令	描述
start	开始执行程序，停在 main 函数第一行语句前面等待命令
step（或 s）	执行下一行语句，如果有函数调用则进入到函数中

习题

1. 用 gdb 一步一步跟踪第 5.3 节讲的 factorial 函数，对照着图 5.2 查看各层栈帧的变化情况，练习本节所学的各种 gdb 命令。

10.2 断点

看以下程序：

例 10.2 断点调试实例

```
#include <stdio.h>

int main(void)
{
        int sum = 0, i = 0;
        char input[5];

        while (1) {
                scanf("%s", input);
                for (i = 0; input[i] != '\0'; i++)
                        sum = sum*10 + input[i] - '0';
                printf("input=%d\n", sum);
        }
        return 0;
}
```

这个程序的作用是：首先从键盘读入一串数字存到字符数组 input 中，再转换成整型存到 sum 中，然后打印出来，一直这样循环下去。scanf("%s", input);这个调用的功能是等待用户输入一个字符串并回车，scanf 把其中第一段非空白（非空格、Tab、换行）的字符串保存到 input 数组中，并自动在末尾添加'\0'。接下来的循环从左到右扫描字符串并把每个数字累加到结果中，例如输入"2345"，则循环累加的过程是(((0×10+2)×10+3)×10+4)×10+5=2345。注意字符型的'2'要减去'0'的 ASCII 码才能转换成整数值 2。下面编译运行程序看看有什么问题：

```
$ gcc main.c -g -o main
$ ./main
123
input=123
234
input=123234
（按 Ctrl+C 组合键退出程序）
$
```

又是这种现象，第一次是对的，第二次就不对。可是这个程序我们并没有忘了赋初值，不仅 sum 赋了初值，连不必赋初值的 i 都赋了初值。读者先试试只看代码能不能看出错误原因。下面来调试：

```
$ gdb main
...
(gdb) start
Temporary breakpoint 1 at 0x804844d: file main.c, line 5.
Starting program: /home/akaedu/main

Temporary breakpoint 1, main () at main.c:5
5               int sum = 0, i = 0;
```

有了上一次的经验，sum 被列为重点怀疑对象，我们可以用 display 命令使得每次停下来的时候都显示当前 sum 的值，然后继续往下走：

```
(gdb) display sum
1: sum = 2637812
(gdb) n
9                       scanf("%s", input);
1: sum = 0
(gdb) （直接回车）
123
10                      for (i = 0; input[i] != '\0'; i++)
1: sum = 0
```

undisplay 命令可以取消跟踪显示，变量 sum 的编号是 1，可以用 undisplay 1 命令取消它的跟踪显示。这个循环应该没有问题，因为上面第一次输入时打印的结果是正确的。如果不想一步一步走这个循环，可以用 break 命令（简写为 b）在第 9 行设一个断点（Breakpoint）：

```
(gdb) l
5               int sum = 0, i = 0;
6               char input[5];
7
8               while (1) {
9                       scanf("%s", input);
10                      for (i = 0; input[i] != '\0'; i++)
11                              sum = sum*10 + input[i] - '0';
12                      printf("input=%d\n", sum);
13              }
14              return 0;
(gdb) b 9
Breakpoint 2 at 0x804845d: file main.c, line 9.
```

break 命令的参数也可以是函数名，表示在某个函数开头设断点。现在用 continue 命令（简写为 c）连续运行而非单步运行，程序到达断点会自动停下来，这样就可以停在下一次循环的开头：

```
(gdb) c
Continuing.
input=123

Breakpoint 2, main () at main.c:9
9                       scanf("%s", input);
```

```
1: sum = 123
```

然后输入新的字符串准备转换：

```
(gdb) n
234
10                      for (i = 0; input[i] != '\0'; i++)
1: sum = 123
```

问题暴露出来了，新的转换应该再次从0开始累加，而sum现在已经是123了，原因在于新的循环没有把sum归零。可见断点有助于快速跳过没有问题的代码，然后在有问题的代码上慢慢走慢慢分析，“断点加单步”是使用调试器的基本方法。至于应该在哪里设置断点，怎么知道哪些代码可以跳过而哪些代码要慢慢走，也要通过对错误现象的分析和假设来确定，以前我们用printf打印中间结果时也要分析应该在哪里插入printf，打印哪些中间结果，调试的基本思路是一样的。一次调试可以设置多个断点，用info命令可以查看已经设置的断点：

```
(gdb) b 12
Breakpoint 3 at 0x80484b3: file main.c, line 12.
(gdb) i breakpoints
Num     Type           Disp Enb Address    What
2       breakpoint     keep y   0x0804845d in main at main.c:9
        breakpoint already hit 1 time
3       breakpoint     keep y   0x080484b3 in main at main.c:12
```

每个断点都有一个编号，可以用编号指定删除某个断点：

```
(gdb) delete breakpoints 2
(gdb) i breakpoints
Num     Type           Disp Enb Address    What
3       breakpoint     keep y   0x080484b3 in main at main.c:12
```

有时候一个断点暂时不用可以禁用掉而不必删除，这样以后想用的时候可以直接启用，而不必重新从代码里找应该在哪一行设断点：

```
(gdb) disable breakpoints 3
(gdb) i breakpoints
Num     Type           Disp Enb Address    What
3       breakpoint     keep n   0x080484b3 in main at main.c:12
(gdb) enable 3
(gdb) i breakpoints
Num     Type           Disp Enb Address    What
3       breakpoint     keep y   0x080484b3 in main at main.c:12
(gdb) delete breakpoints
Delete all breakpoints? (y or n) y
(gdb) i breakpoints
No breakpoints or watchpoints.
```

gdb的断点功能非常灵活，还可以设置断点在满足某个条件时才激活，例如我们仍然在循环开头设置断点，但是仅当sum不等于0时才中断，然后用run命令（简写为r）重新从程序开头连续运行：

```
(gdb) break 9 if sum != 0
Breakpoint 4 at 0x804845d: file main.c, line 9.
```

```
(gdb) i breakpoints
Num     Type           Disp Enb Address    What
4       breakpoint     keep y   0x0804845d in main at main.c:9
        stop only if sum != 0
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/akaedu/main
123
input=123

Breakpoint 4, main () at main.c:9
9               scanf("%s", input);
1: sum = 123
```

结果是第一次执行 scanf 之前没有中断，第二次却中断了。总结一下本节用到的 gdb 命令，如表 10.2 所示。

表 10.2 gdb 基本命令 2

命令	描述
break（或 b）行号	在某一行设置断点
break 函数名	在某个函数开头设置断点
break ... if ...	设置条件断点
continue（或 c）	从当前位置开始连续运行程序
delete breakpoints 断点号	删除断点
display 变量名	跟踪查看某个变量，每次停下来都显示它的值
disable breakpoints 断点号	禁用断点
enable 断点号	启用断点
info（或 i） breakpoints	查看当前设置了哪些断点
run（或 r）	从头开始连续运行程序
undisplay 跟踪显示号	取消跟踪显示

习题

1. 看下面的程序：

```
#include <stdio.h>

int main(void)
{
        int i;
        char str[6] = "hello";
        char reverse_str[6] = "";

        printf("%s\n", str);
        for (i = 0; i < 5; i++)
                reverse_str[5-i] = str[i];
        printf("%s\n", reverse_str);
        return 0;
}
```

首先用字符串"hello"初始化一个字符数组 str（算上'\0'共 6 个字符）。然后用空字符串""初始化一个同样长的字符数组 reverse_str，相当于所有元素用'\0'初始化。然后打印 str，把 str 倒序存入 reverse_str，再打印 reverse_str。然而结果并不正确：

```
$ ./main
hello
```

我们本来希望 reverse_str 打印出来是 olleh，结果打出来一个空行。重点怀疑对象肯定是循环，那么简单验算一下，i=0 时，reverse_str[5]=str[0]，也就是'h'，i=1 时，reverse_str[4]=str[1]，也就是'e'，依此类推，i=0,1,2,3,4，共 5 次循环，正好把 h,e,l,l,o 五个字母给倒过来了，哪里不对了？请用 gdb 跟踪循环，找出错误原因并改正。

10.3 观察点

接着上一节的步骤，经过调试我们知道，虽然 sum 已经赋了初值 0，但仍需要在 while (1)循环的开头加上 sum = 0;：

例 10.3 观察点调试实例

```
#include <stdio.h>

int main(void)
{
        int sum = 0, i = 0;
        char input[5];

        while (1) {
                sum = 0;
                scanf("%s", input);
                for (i = 0; input[i] != '\0'; i++)
                        sum = sum*10 + input[i] - '0';
                printf("input=%d\n", sum);
        }
        return 0;
}
```

使用 scanf 函数是非常凶险的，即使修正了这个 Bug 也还存在很多问题。如果输入的字符串超长了会怎么样？我们知道数组访问越界是不会检查的，所以 scanf 会写出界。现象是这样的：

```
$ ./main
123
input=123
67
input=67
12345
input=123407
```

下面用调试器看看最后这个诡异的结果是怎么出来的。

```
$ gdb main
...
```

```
(gdb) start
Temporary breakpoint 1 at 0x804844d: file main.c, line 5.
Starting program: /home/akaedu/main

Temporary breakpoint 1, main () at main.c:5
5               int sum = 0, i = 0;
(gdb) n
9                       sum = 0;
(gdb) （直接回车）
10                      scanf("%s", input);
(gdb) （直接回车）
12345
11                      for (i = 0; input[i] != '\0'; i++)
(gdb) p input
$1 = "12345"
```

input数组只有5个元素，写出界的是scanf自动添的'\0'，用x命令查看会更清楚一些：

```
(gdb) x/7bx input
0xbffff373: 0x31    0x32    0x33    0x34    0x35    0x00    0x00
```

x命令打印指定存储单元里保存的内容，后缀7bx是打印格式，7表示打印7组，b表示每个字节一组，x表示按十六进制格式打印[②]，x/7bx这条命令从input数组的第一个字节开始连续打印7个字节。前5个字节是input数组的存储单元，打印的正是十六进制ASCII码的'1'到'5'，第6个字节是写出界的'\0'。

根据运行结果，前4个字符转成数字都没错，第5个错了，也就是i从0到3的循环都没错，我们设一个条件断点从i等于4开始单步调试：

```
(gdb) l
6               char input[5];
7
8               while (1) {
9                       sum = 0;
10                      scanf("%s", input);
11                      for (i = 0; input[i] != '\0'; i++)
12                              sum = sum*10 + input[i] - '0';
13                      printf("input=%d\n", sum);
14              }
15              return 0;
(gdb) b 12 if i == 4
Breakpoint 2 at 0x8048484: file main.c, line 12.
(gdb) c
Continuing.

Breakpoint 2, main () at main.c:12
12                              sum = sum*10 + input[i] - '0';
(gdb) p sum
$2 = 1234
```

现在sum是1234没错，根据运行结果是123407，我们知道即将进行的这步计算肯定要出错，算出来应该是12340，那就是说input[4]肯定不是'5'了，事实证明这

② 打印结果最左边的一长串数字是内存地址，在第16.1节详细解释，目前可以无视。

个推理是不严谨的：

```
(gdb) x/7bx input
0xbffff373: 0x31    0x32    0x33    0x34    0x35    0x04    0x00
```

input[4]的确是0x35。再分析一下发现，产生123407这个结果还有另外一种可能，就是在下一次循环中123450不是加上而是减去一个数得到123407。可现在不是到字符串末尾了吗？怎么会有下一次循环呢？注意到循环控制条件是input[i] != '\0'，而本来应该是0x00的位置现在莫名其妙地变成了0x04，因此循环不会结束。继续单步调试：

```
(gdb) n
11                  for (i = 0; input[i] != '\0'; i++)
(gdb) p sum
$3 = 12345
(gdb) n
12                          sum = sum*10 + input[i] - '0';
(gdb) x/7bx input
0xbffff373: 0x31    0x32    0x33    0x34    0x35    0x05    0x00
```

进入下一次循环，原来的0x04又莫名其妙地变成了0x05，这是怎么回事？这个暂时解释不了，但123407这个结果可以解释了，是12345×10 + 0x05 - 0x30得到的，虽然多循环了一次，但下次一定会退出循环了，因为0x05的后面是'\0'。

input[4]后面那个字节到底是什么时候变的？可以用观察点（Watchpoint）来跟踪。我们知道断点是当程序执行到某一代码行时中断，而观察点是当程序访问某个存储单元时中断，如果我们不知道某个存储单元是在哪里被改动的，这时候观察点尤其有用。下面删除原来设的断点，从头执行程序，重复上次的输入，用watch命令设置观察点，跟踪input[4]后面那个字节（可以用input[5]表示，虽然这是访问越界）：

```
(gdb) delete breakpoints
Delete all breakpoints? (y or n) y
(gdb) start
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Temporary breakpoint 3 at 0x804844d: file main.c, line 5.
Starting program: /home/akaedu/main

Temporary breakpoint 3, main () at main.c:5
5               int sum = 0, i = 0;
(gdb) n
9                       sum = 0;
(gdb) （直接回车）
10                      scanf("%s", input);
(gdb) （直接回车）
12345
11                      for (i = 0; input[i] != '\0'; i++)
(gdb) watch input[5]
Hardware watchpoint 4: input[5]
(gdb) i watchpoints
Num     Type           Disp Enb Address    What
4       hw watchpoint  keep y              input[5]
```

```
(gdb) c
Continuing.
Hardware watchpoint 4: input[5]

Old value = 0 '\000'
New value = 1 '\001'
0x080484ae in main () at main.c:11
11                      for (i = 0; input[i] != '\0'; i++)
(gdb) c
Continuing.
Hardware watchpoint 4: input[5]

Old value = 1 '\001'
New value = 2 '\002'
0x080484ae in main () at main.c:11
11                      for (i = 0; input[i] != '\0'; i++)
(gdb) c
Continuing.
Hardware watchpoint 4: input[5]

Old value = 2 '\002'
New value = 3 '\003'
0x080484ae in main () at main.c:11
11                      for (i = 0; input[i] != '\0'; i++)
```

已经很明显了，每次都是回到 for 循环开头的时候改变了 input[5]的值，而且是每次加 1，而循环变量 i 正是在每次回到循环开头之前加 1，原来 input[5]就是变量 i 的存储单元，换句话说，i 的存储单元是紧跟在 input 数组后面的。

修正这个 Bug 对初学者来说有一定难度。如果你发现了这个 Bug 却没想到数组访问越界这一点，也许一时想不出原因，就会先去处理另外一个更容易修正的 Bug：如果输入的不是数字而是字母或别的符号也能算出结果来。这显然是不对的，可以在循环中加上判断条件检查非法字符：

```
while (1) {
    sum = 0;
    scanf("%s", input);
    for (i = 0; input[i] != '\0'; i++) {
            if (input[i] < '0' || input[i] > '9') {
                    printf("Invalid input!\n");
                    sum = -1;
                    break;
            }
            sum = sum*10 + input[i] - '0';
    }
    printf("input=%d\n", sum);
}
```

然后你会惊喜地发现，不仅输入字母会报错，输入超长也会报错：

```
$ ./main
123a
Invalid input!
input=-1
dead
Invalid input!
```

```
input=-1
1234578
Invalid input!
input=-1
1234567890abcdef
Invalid input!
input=-1
23
input=23
```

似乎是两个Bug一起解决掉了，但这是治标不治本的解决方法。看起来输入超长的错误是不会出现了，但只要没有找到根本原因就不可能真的解决掉，等到条件一变，它可能又冒出来了，在下一节你会看到它又以一种新的形式冒出来了。现在请思考一下为什么加上检查非法字符的代码之后输入超长也会报错。最后总结一下本节用到的gdb命令，如表10.3所示。

表10.3 gdb基本命令3

命令	描述
watch	设置观察点
info（或i） watchpoints	查看当前设置了哪些观察点
x	从某个位置开始打印存储单元的内容，全部当成字节来看，而不区分哪个字节属于哪个变量

10.4 段错误

如果程序运行时出现段错误，用gdb可以很容易定位到究竟是哪一行引发的段错误，例如这个小程序：

例10.4 段错误调试实例一

```
#include <stdio.h>

int main(void)
{
        int man = 0;
        scanf("%d", man);
        return 0;
}
```

调试过程如下：

```
$ gdb main
...
(gdb) r
Starting program: /home/akaedu/main
123

Program received signal SIGSEGV, Segmentation fault.
0x00175ed7 in _IO_vfscanf () from /lib/tls/i686/cmov/libc.so.6
(gdb) bt
#0  0x00175ed7 in _IO_vfscanf () from /lib/tls/i686/cmov/libc.so.6
```

```
#1  0x0017caa9 in __isoc99_scanf () from /lib/tls/i686/cmov/libc.so. 6
#2  0x0804842a in main () at main.c:6
```

在 gdb 中运行，遇到段错误会自动停下来，这时可以用命令查看当前执行到哪一行代码了。gdb 显示段错误出现在_IO_vfscanf 函数中，用 bt 命令可以看到这个函数是被 main.c 的第 6 行间接调用的，也就是 scanf 这行代码引发的段错误。仔细观察程序发现是 man 前面少了个&。

继续调试上一节的程序，上一节最后提出修正 Bug 的方法是在循环中加上判断条件，如果不是数字就报错退出，不仅输入字母可以报错退出，输入超长的字符串也会报错退出。表面上看这个程序无论怎么运行都不出错了，但假如我们把 while (1)循环去掉，每次执行程序只转换一个数：

例 10.5 段错误调试实例二

```
#include <stdio.h>

int main(void)
{
        int sum = 0, i = 0;
        char input[5];

        scanf("%s", input);
        for (i = 0; input[i] != '\0'; i++) {
                if (input[i] < '0' || input[i] > '9') {
                        printf("Invalid input!\n");
                        sum = -1;
                        break;
                }
                sum = sum*10 + input[i] - '0';
        }
        printf("input=%d\n", sum);
        return 0;
}
```

然后输入一个超长的字符串，看看会发生什么：

```
$ ./main
1234567890
Invalid input!
input=-1
```

看起来正常。再来一次，这次输个更长的：

```
$ ./main
1234567890abcdef1234567890abcdef
Invalid input!
input=-1
Segmentation fault
```

又出段错误了。我们按照同样的方法用 gdb 调试看看：

```
$ gdb main
...
(gdb) r
```

```
Starting program: /home/akaedu/main
1234567890abcdef1234567890abcdef
Invalid input!
input=-1

Program received signal SIGSEGV, Segmentation fault.
0x0804852e in main () at main.c:19
19      }
(gdb) l
14                      }
15                      sum = sum*10 + input[i] - '0';
16              }
17              printf("input=%d\n", sum);
18              return 0;
19      }
```

gdb 指出，段错误发生在第 19 行。可是这一行什么都没有啊，只有表示 main 函数结束的}括号。这可以算是一条规律，**如果某个函数的局部变量发生访问越界，有可能并不立即产生段错误，而是在函数返回时产生段错误。**

想要写出 Bug-free 的程序是非常不容易的，即使 scanf 读入字符串这么一个简单的函数调用都会隐藏着各种各样的错误，有些错误现象是我们暂时没法解释的：为什么变量 i 的存储单元紧跟在 input 数组后面？为什么同样是访问越界，有时出段错误有时不出段错误？为什么访问越界的段错误在函数返回时才出现？还有最基本的问题，为什么 scanf 输入整型变量就必须要加&，否则就出段错误，而输入字符串就不要加&？这些问题在后续章节中都会解释清楚。

其实现在讲 scanf 这个函数为时过早，读者还不具备充足的基础知识。但还是有必要讲的，学完这一阶段之后读者应该能写出有用的程序了，然而一个只有输出而没有输入的程序算不上是有用的程序，另一方面也让读者认识到，学 C 语言不可能不去了解底层计算机体系结构和操作系统的原理，不了解底层原理连一个 scanf 函数都没办法用好，更没有办法保证写出正确的程序。除了要理解底层工作原理之外，scanf 这个函数的用法也确实是相当复杂，要用得准确无误是挺难的，本书将在第 24.2.9 节详细解释这个函数。

第 11 章

排序与查找

11.1 算法的概念

算法（Algorithm）是将一组输入转化成一组输出的一系列计算步骤，其中每个步骤必须能在有限时间内完成。比如第 5.3 节习题 1 中的 Euclid 算法，输入是两个正整数，输出是它们的最大公约数，计算步骤是取模、比较等操作，这个算法一定能在有限的步骤和时间内完成（想一想为什么？）。再比如将一组数从小到大排序，输入是一组原始数据，输出是排序之后的数据，计算步骤包括比较、移动数据等操作。

算法是用来解决一类计算问题的，注意是一类问题，而不是一个特定的问题。例如，一个排序算法应该能对任意一组数据进行排序，而不是仅对 int a[] = { 1, 3, 4, 2, 6, 5 };这样一组特定的数据排序，如果只需要对这一组数据排序可以写这样一个函数来做：

```
void sort(void)
{
        a[0] = 1;
        a[1] = 2;
        a[2] = 3;
        a[3] = 4;
        a[4] = 5;
        a[5] = 6;
}
```

这显然不叫算法，因为不具有通用性。由于算法是用来解决一类问题的，它必须能够正确地解决这一类问题中的任何一个实例，这个算法才是正确的。对于排序算法，任意输入一组数据，它必须都能输出正确的排序结果，这个排序算法才是正确的。不正确的算法有两种可能，一是对于该问题的某些输入，该算法会无限计算下去，不会终止；二是对于该问题的某些输入，该算法终止时输出的是错误的结果。有时候不正确的算法也是有用的，如果对于某个问题寻求正确的算法很困难，而某个不正确的算法可以在有限时间内终止，并且能把误差控制在一定范围内，那么这样的算法也是有实际意义的。例如有时候寻找最优解的开销很大，往往会选择能给出次优解的算法。

本章介绍几种典型的排序和查找算法，并围绕这几种算法做时间复杂度分析。学

完本章之后如果想进一步学习，可以参考一些全面系统地介绍算法的书，例如参考文献[16]和参考文献[17]。

11.2 插入排序

插入排序算法类似于玩扑克时抓牌的过程，玩家每拿到一张牌都要插入到手中已有的牌里，使之从小到大排好序，如图 11.1 所示（该图出自参考文献[16]的 2.1 节）：

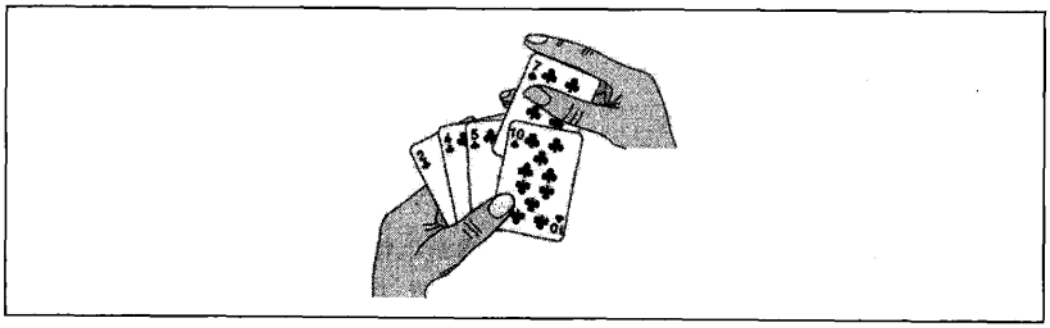


图 11.1 扑克牌的插入排序

也许你没有意识到，但其实你的思考过程是这样的：现在抓到一张 7，把它和手里的牌从右到左依次比较，7 比 10 小，应该再往左插，7 比 5 大，好，就插这里。为什么比较了 10 和 5 就可以确定 7 的位置？为什么不用再比较左边的 4 和 2 呢？因为这里有一个重要的前提：手里的牌已经是排好序的。现在我插了 7 之后，手里的牌仍然是排好序的，下次再抓到的牌还可以用这个方法插入。

编程对一个数组进行插入排序也是同样道理，但和插入扑克牌有一点不同，不可能在两个相邻的存储单元之间再插入一个单元，因此要将插入点之后的数据依次往后移动一个单元。排序算法如下：

例 11.1 插入排序

```
#include <stdio.h>

#define LEN 5
int a[LEN] = { 10, 5, 2, 4, 7 };

void insertion_sort(void)
{
        int i, j, key;
        for (j = 1; j < LEN; j++) {
                printf("%d, %d, %d, %d, %d\n",
                       a[0], a[1], a[2], a[3], a[4]);
                key = a[j];
                i = j - 1;
                while (i >= 0 && a[i] > key) {
                        a[i+1] = a[i];
                        i--;
                }
                a[i+1] = key;
```

```
        }
        printf("%d, %d, %d, %d, %d\n",
               a[0], a[1], a[2], a[3], a[4]);
}

int main(void)
{
        insertion_sort();
        return 0;
}
```

为了更清楚地观察排序过程，我们在每次循环开头插了打印语句，在排序结束后也插了打印语句。程序运行结果是：

```
$ ./a.out
10, 5, 2, 4, 7
5, 10, 2, 4, 7
2, 5, 10, 4, 7
2, 4, 5, 10, 7
2, 4, 5, 7, 10
```

如何严格证明这个算法是正确的？换句话说，只要反复执行该算法的 for 循环体，执行 LEN-1 次，就一定能把数组 a 排好序，而不管数组 a 的原始数据是什么，如何证明这一点呢？我们可以借助 Loop Invariant 的概念和数学归纳法来理解循环结构的算法，假如某个判断条件满足以下三条准则，它就称为 Loop Invariant：

1．第一次执行循环体之前该判断条件为真。

2．如果“第 N-1 次循环之后（或者说第 N 次循环之前）该判断条件为真”这个前提可以成立，那么就有办法证明第 N 次循环之后该判断条件仍为真。

3．如果在所有循环结束后该判断条件为真，那么就有办法证明该算法正确地解决了问题。

只要我们找到这个 Loop Invariant，就可以证明一个循环结构的算法是正确的。上述插入排序算法的 Loop Invariant 是这样的判断条件：**第 j 次循环之前，子序列 a[0..j-1]是排好序的**。在上面的打印结果中，我把子序列 a[0..j-1]加粗表示。下面我们验证一下 Loop Invariant 的三条准则：

1．第一次执行循环之前，j=1，子序列 a[0..j-1]只有一个元素 a[0]，只有一个元素的序列显然是排好序的。

2．第 j 次循环之前，如果“子序列 a[0..j-1]是排好序的”这个前提成立，现在要把 key=a[j]插进去，按照该算法的步骤，把 a[j-1]、a[j-2]、a[j-3]等比 key 大的元素都依次往后移一个，直到找到合适的位置将 key 插入，就能证明循环结束时子序列 a[0..j]是排好序的。就像插扑克牌一样，“手中已有的牌是排好序的”这个前提很重要，如果没有这个前提，就不能证明再插一张牌之后也是排好序的。

3．当循环结束时，j=LEN，如果“子序列 a[0..j-1]是排好序的”这个前提成立，那就是说 a[0..LEN-1]是排好序的，也就是说整个数组 a 的 LEN 个元素都排好序了。

可见，有了这三条，就可以用数学归纳法证明这个循环是正确的。这和第 5.3 节证明递归程序正确性的思路是一致的，这里的第一条就相当于递归的 Base Case，第二条就相当于递归的递推关系。这再次说明了递归和循环是等价的。

习题

1. 实现选择排序算法：第一次从数组 a[0..LEN-1]中找出最小元素交换到 a[0]的位置，第二次从数组 a[1..LEN-1]中找出最小元素交换到 a[1]的位置，依此类推。排序过程举例如下：

```
10, 5, 2, 4, 7
2, 5, 10, 4, 7
2, 4, 10, 5, 7
2, 4, 5, 10, 7
2, 4, 5, 7, 10
```

11.3 算法的时间复杂度分析

解决同一个问题可以有很多种算法，比较评价算法的好坏，一个重要的标准就是算法的时间复杂度。现在研究一下插入排序算法的执行时间，按照习惯，输入长度 LEN 以下用 n 表示。设循环中各条语句的执行时间分别是 $c1$、$c2$、$c3$、$c4$、$c5$ 这样五个常数[①]：

```
void insertion_sort(void)          执行时间
{
        int i, j, key;
        for (j = 1; j < LEN; j++) {
                key = a[j];                         c1
                i = j - 1;                          c2
                while (i >= 0 && a[i] > key) {
                        a[i+1] = a[i];              c3
                        i--;                        c4
                }
                a[i+1] = key;                       c5
        }
}
```

显然外层 for 循环的执行次数是 n-1 次，假设内层的 while 循环执行 m 次，则总的执行时间粗略估计是$(n-1)\times(c1+c2+c5+m\times(c3+c4))$。当然，for 和 while 后面()括号中的赋值和条件判断的执行也需要时间，而我没有设一个常数来表示，这不影响我们的粗略估计。

这里有一个问题，m 不是个常数，也不取决于输入长度 n，而是取决于具体的输入数据。在最好情况下，数组 a 的原始数据已经排好序了，while 循环一次也不执行，总的执行时间是$(c1+c2+c5)\times n-(c1+c2+c5)$，可以表示成 $an+b$ 的形式，是 n 的

① 受内存管理机制的影响，指令的执行时间不一定是常数，但执行时间的上界（Upper Bound）肯定是常数，我们这里假设语句的执行时间是常数只是一个粗略估计。

线性函数（Linear Function）。那么在最坏情况（Worst Case）下又如何呢？所谓最坏情况是指数组 *a* 的原始数据正好是从大到小排好序的，请读者想一想为什么这是最坏情况，然后把上式中的 *m* 替换掉算一下执行时间是多少。

数组 a 的原始数据属于最好和最坏情况的都比较少见，如果原始数据是随机的，可称为平均情况（Average Case）。如果原始数据是随机的，那么每次循环将已排序的子序列 a[1..j-1]与新插入的元素 key 相比较，子序列中平均都有一半的元素比 key 大而另一半比 key 小，请读者把上式中的 *m* 替换掉算一下执行时间是多少。最后的结论应该是：在最坏情况和平均情况下，总的执行时间都可以表示成 an^2+bn+c 的形式，是 *n* 的二次函数（Quadratic Function）。

在分析算法的时间复杂度时，我们更关心最坏情况而不是最好情况，理由如下：

1．最坏情况给出了算法执行时间的上界，我们可以确信，无论给什么输入，算法的执行时间都不会超过这个上界，这样为比较和分析提供了便利。

2．对于某些算法，最坏情况是最常发生的情况，例如在数据库中查找某个信息的算法，最坏情况就是数据库中根本不存在该信息，都找遍了也没有，而某些应用场合经常要查找一个信息在数据库中存在不存在。

比较两个多项式 a_1n+b_1 和 $a_2n^2+b_2n+c_2$ 的值（*n* 取正整数）可以得出结论：*n* 的最高次指数是最主要的决定因素，常数项、低次幂项和系数都是次要的。比如 $100n+1$ 和 n^2+1，虽然后者的系数小，当 *n* 较小时前者的值较大，但是当 $n>100$ 时，后者的值就远远大于前者了。如果同一个问题可以用两种算法解决，其中一种算法的时间复杂度为线性函数，另一种算法的时间复杂度为二次函数，当问题的输入长度 *n* 足够大时，前者明显优于后者。因此我们可以用一种更粗略的方式表示算法的时间复杂度，把系数和低次幂项都省去，线性函数记作 $\Theta(n)$，二次函数记作 $\Theta(n^2)$。

$\Theta(g(n))$表示和 $g(n)$同一量级的一类函数，例如所有的二次函数 $f(n)$都和 $g(n)=n^2$ 属于同一量级，都可以用 $\Theta(n^2)$来表示，甚至有些不是二次函数的也和 n^2 属于同一量级，例如 $2n^2+3\lg n$。“同一量级”这个概念可以用图 11.2 来说明（该图出自参考文献[16]的 3.1 节）：

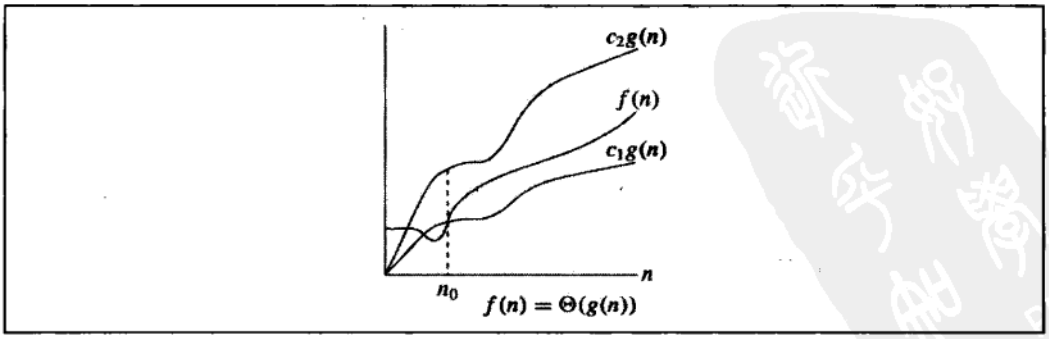


图 11.2　Θ-notation

如果可以找到两个正的常数 c_1 和 c_2，使得 *n* 足够大的时候（也就是 $n \geq n_0$ 的时候）

f(n)总是夹在c_1g(n)和c_2g(n)之间，就说f(n)和g(n)是同一量级的，f(n)就可以用Θ(g(n))来表示。

以二次函数为例，比如$1/2n^2-3n$，要证明它是属于Θ(n^2)这个集合的，我们必须确定c_1、c_2和n_0，这些常数不随n改变，并且当$n \geqslant n_0$以后，$c_1n^2 \leqslant 1/2n^2-3n \leqslant c_2n^2$总是成立的。为此我们从不等式的每一边都除以$n^2$，得到$c_1 \leqslant 1/2-3/n \leqslant c_2$。如图11.3所示：

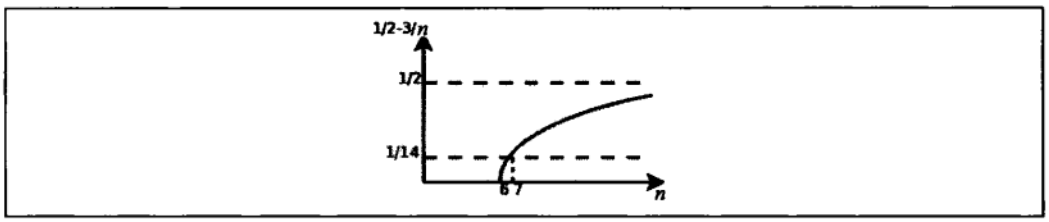


图11.3 1/2-3/n

这样就很容易看出来，无论n取多少，该函数一定小于1/2，因此c_2=1/2，当n=6时函数值为0，n>6时该函数都大于0，可以取n_0=7，c_1=1/14，这样当$n \geqslant n_0$时都有$1/2-3/n \geqslant c_1$。通过这个证明过程可以得出结论，当n足够大时任何an^2+bn+c都夹在c_1n^2和c_2n^2之间，相对于n^2项来说$bn+c$的影响可以忽略，a可以通过选取合适的c_1、c_2来补偿。

几种常见的时间复杂度函数按数量级从小到大的顺序依次是：Θ(lg$n$)，Θ(sqrt(n))，Θ(n)，Θ($n$lg$n$)，Θ(n^2)，Θ(n^3)，Θ(2^n)，Θ(n!)。其中，lg$n$通常表示以10为底n的对数，但是对于Θ-notation来说，Θ(lg$n$)和Θ($\log_2 n$)并无区别（想一想这是为什么），在算法分析中lg$n$通常表示以2为底n的对数。可是什么算法的时间复杂度里会出现lg$n$呢？回顾插入排序的时间复杂度分析，无非是循环体的执行时间乘以循环次数，只有加和乘运算，怎么会出来lg呢？下一节归并排序的时间复杂度里面就有lg，请读者留心lg运算是从哪出来的。

除了Θ-notation之外，表示算法的时间复杂度常用的还有一种Big-O notation。我们知道插入排序在最坏情况和平均情况下时间复杂度是Θ(n^2)，在最好情况下是Θ(n)，数量级比Θ(n^2)要小，那么总结起来在各种情况下插入排序的时间复杂度是O(n^2)。Θ的含义和“等于”类似，而大O的含义和“小于等于”类似。

11.4 归并排序

插入排序算法采取增量式（Incremental）的策略解决问题，每次添一个元素到已排序的子序列中，逐渐将整个数组排序完毕，它的时间复杂度是O(n^2)。下面介绍另一种典型的排序算法——归并排序，它采取分而治之（Divide-and-Conquer）的策略，时间复杂度是Θ($n$lg$n$)。归并排序的步骤如下：

1．Divide：把长度为n的输入序列分成两个长度为n/2的子序列。

2．Conquer：对这两个子序列分别采用归并排序。

3．Combine：将两个排序好的子序列合并成一个最终的排序序列。

在描述归并排序的步骤时又调用了归并排序本身，可见这是一个递归的过程。

例 11.2　归并排序

```
#include <stdio.h>

#define LEN 8
int a[LEN] = { 5, 2, 4, 7, 1, 3, 2, 6 };

void merge(int start, int mid, int end)
{
        int n1 = mid - start + 1;
        int n2 = end - mid;
        int left[n1], right[n2];
        int i, j, k;

        for (i = 0; i < n1; i++) /* left holds a[start..mid] */
                left[i] = a[start+i];
        for (j = 0; j < n2; j++) /* right holds a[mid+1..end] */
                right[j] = a[mid+1+j];

        i = j = 0;
        k = start;
        while (i < n1 && j < n2)
                if (left[i] < right[j])
                        a[k++] = left[i++];
                else
                        a[k++] = right[j++];

        while (i < n1) /* left[] is not exhausted */
                a[k++] = left[i++];
        while (j < n2) /* right[] is not exhausted */
                a[k++] = right[j++];
}

void sort(int start, int end)
{
        int mid;
        if (start < end) {
                mid = (start + end) / 2;
                printf("sort (%d-%d, %d-%d) %d %d %d %d %d %d %d
                %d\n",
                      start, mid, mid+1, end,
                      a[0], a[1], a[2], a[3], a[4], a[5], a[6],
                      a[7]);
                sort(start, mid);
                sort(mid+1, end);
                merge(start, mid, end);
                printf("merge (%d-%d, %d-%d) to %d %d %d %d %d %d
                %d %d\n",
                      start, mid, mid+1, end,
                      a[0], a[1], a[2], a[3], a[4], a[5], a[6],
                      a[7]);
        }
}

int main(void)
```

```
{
        sort(0, LEN-1);
        return 0;
}
```

执行结果是：

```
sort (0-3, 4-7) 5 2 4 7 1 3 2 6
sort (0-1, 2-3) 5 2 4 7 1 3 2 6
sort (0-0, 1-1) 5 2 4 7 1 3 2 6
merge (0-0, 1-1) to 2 5 4 7 1 3 2 6
sort (2-2, 3-3) 2 5 4 7 1 3 2 6
merge (2-2, 3-3) to 2 5 4 7 1 3 2 6
merge 0-1, 2-3) to 2 4 5 7 1 3 2 6
sort (4-5, 6-7) 2 4 5 7 1 3 2 6
sort (4-4, 5-5) 2 4 5 7 1 3 2 6
merge (4-4, 5-5) to 2 4 5 7 1 3 2 6
sort (6-6, 7-7) 2 4 5 7 1 3 2 6
merge (6-6, 7-7) to 2 4 5 7 1 3 2 6
merge (4-5, 6-7) to 2 4 5 7 1 2 3 6
merge (0-3, 4-7) to 1 2 2 3 4 5 6 7
```

sort 函数把 a[start..end]平均分成两个子序列，分别是 a[start..mid]和 a[mid+1..end]，对这两个子序列分别递归调用 sort 函数进行排序，然后调用 merge 函数将排好序的两个子序列合并起来，由于两个子序列都已经排好序了，合并的过程很简单，每次循环取两个子序列中最小的元素进行比较，将较小的元素取出放到最终的排序序列中，如果其中一个子序列的元素已取完，就把另一个子序列剩下的元素都放到最终的排序序列中。为了便于理解程序，我在 sort 函数开头和结尾插了打印语句，可以看出调用过程是这样的，如图 11.4 所示：

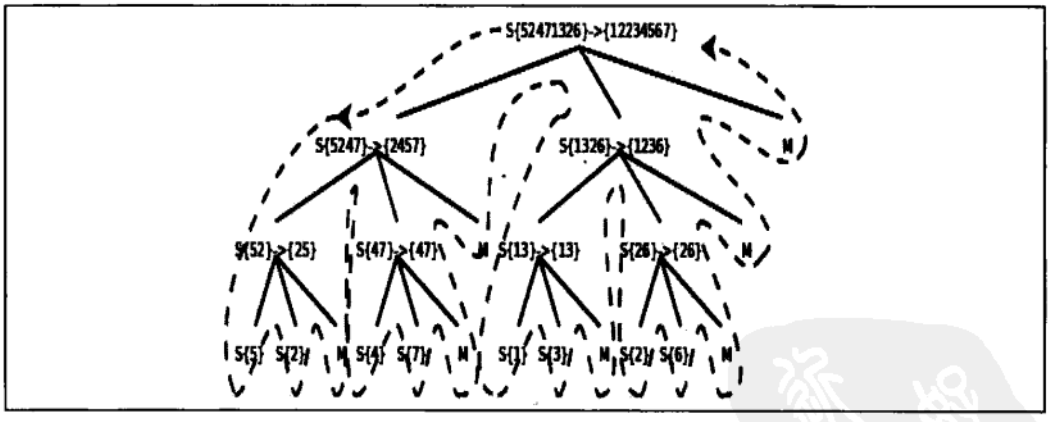


图 11.4 归并排序调用过程

图中 S 表示 sort 函数，M 表示 merge 函数，整个控制流程沿虚线所示的方向调用和返回。由于 sort 函数递归调用了自己两次，所以各函数之间的调用关系呈树状结构。画这个图只是为了清楚地展现归并排序的过程，读者在理解递归函数时一定不要全部展开来看，而是要抓住 Base Case 和递推关系来理解。我们分析一下归并排序的时间复杂度，以下分析出自参考文献[16]的 2.3 节。

首先分析 merge 函数的时间复杂度。在 merge 函数中演示了 C99 的新特性——可

变长数组，当然也可以避免使用这一特性，比如把 left 和 right 都按最大长度 LEN 分配。不管用哪种办法，定义数组并分配存储空间的执行时间都可以看作常数，与数组的长度无关，常数用 Θ-notation 记作 Θ(1)。设子序列 a[start..mid]的长度为 $n1$，子序列[mid+1..end]的长度为 $n2$，a[start..end]的总长度为 $n=n1+n2$，则前两个 for 循环的执行时间是 $\Theta(n1+n2)$，也就是 $\Theta(n)$，后面三个 for 循环合在一起看，每走一次循环就会在最终的排序序列中确定一个元素，最终的排序序列共有 n 个元素，所以执行时间也是 $\Theta(n)$。两个 $\Theta(n)$再加上若干常数项，merge 函数总的执行时间仍是 $\Theta(n)$，其中 n=end-start+1。

然后分析 sort 函数的时间复杂度，当输入长度 n=1，也就是 start==end 时，if 条件不成立，执行时间为常数 Θ(1)，当输入长度 n>1 时：

总的执行时间 = 2 × 输入长度为 $n/2$ 的 sort 函数的执行时间 + merge 函数的执行时间 Θ(n)

设输入长度为 n 的 sort 函数的执行时间为 $T(n)$，综上所述：

$$T(n)\begin{cases}\Theta(1) & \text{if } n=1,\\ 2T(n/2)+\Theta(n) & \text{if } n>1。\end{cases}$$

这是一个递推公式（Recurrence），我们需要消去等号右侧的 $T(n)$，把 $T(n)$写成 n 的函数。其实符合一定条件的 Recurrence 的展开有数学公式可以套，这里我们略去严格的数学证明，只是从直观上看一下这个递推公式的结果。当 n=1 时可以设 $T(1)=c_1$，当 n>1 时可以设 $T(n)=2T(n/2)+c_2n$，我们取 c_1 和 c_2 中较大的一个设为 c，把原来的公式改为：

$$T(n)\begin{cases}c & \text{if } n=1,\\ 2T(n/2)+cn & \text{if } n>1。\end{cases}$$

这样计算出的结果应该是 $T(n)$的上界。下面我们把 $T(n/2)$展开成 $2T(n/4)+cn/2$（图 11.5 中的(c)），然后再把 $T(n/4)$进一步展开，直到最后全部变成 $T(1)=c$（图 11.5 中的(d)）：

把图 11.5(d)中所有的项加起来就是总的执行时间。这是一个树状结构，每一层的和都是 cn，共有 $\lg n+1$ 层，因此总的执行时间是 $cn\lg n+cn$，相比 $n\lg n$ 来说，cn 项可以忽略，因此 $T(n)$的上界是 $\Theta(n\lg n)$。

如果先前取 c_1 和 c_2 中较小的一个设为 c，计算出的结果应该是 $T(n)$的下界，然而推导过程一样，结果也是 $\Theta(n\lg n)$。既然 $T(n)$的上下界都是 $\Theta(n\lg n)$，显然 $T(n)$就是 $\Theta(n\lg n)$。

和插入排序的平均情况相比归并排序更快一些，虽然 merge 函数的步骤较多，引入了较大的常数、系数和低次项，但是对于较大的输入长度 n，这些都不是主要因素，归并排序的时间复杂度是 $\Theta(n\lg n)$，而插入排序的平均情况是 $\Theta(n^2)$，这就决定了归并排序是更快的算法。但是不是任何情况下归并排序都优于插入排序

呢？哪些情况适用插入排序而不适用归并排序？这些问题留给读者思考。

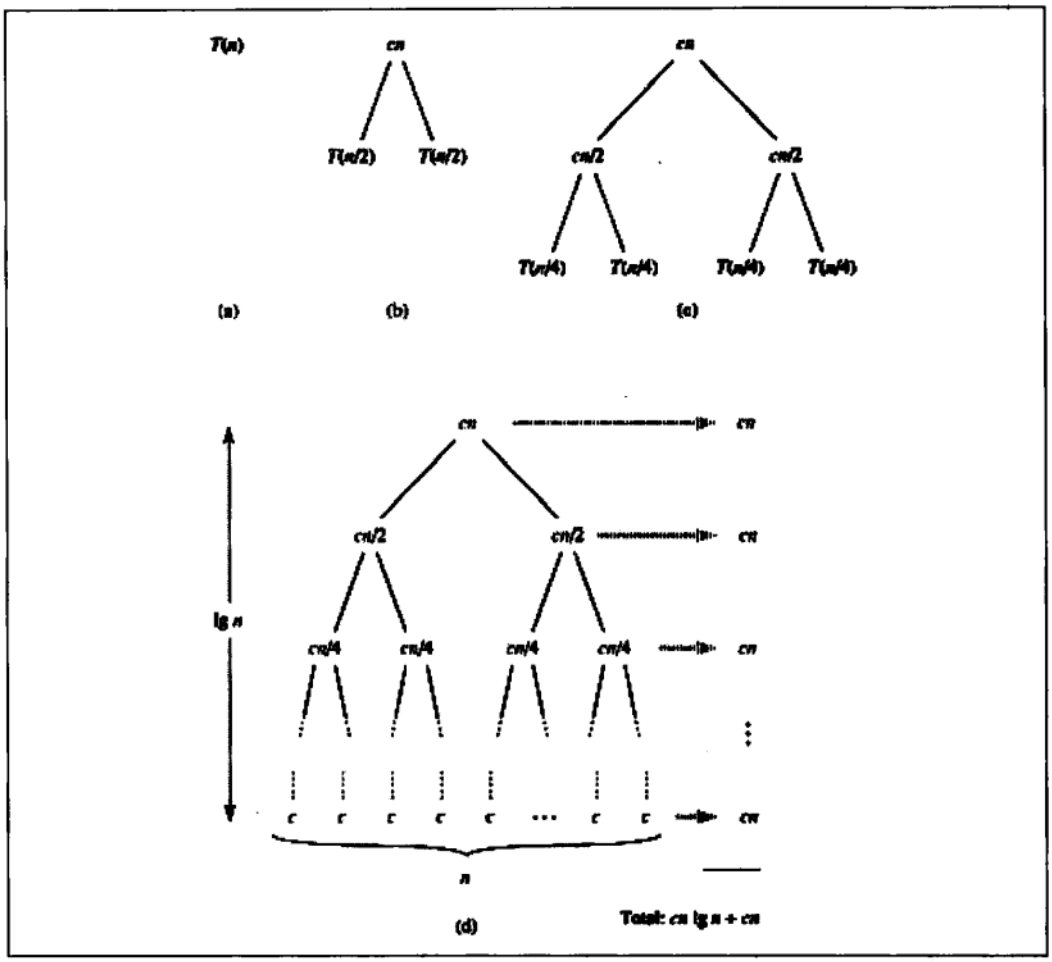


图 11.5 归并排序算法的时间复杂度分析

习题

1. 快速排序是另外一种采用分而治之策略的排序算法，在平均情况下的时间复杂度也是 $\Theta(n\lg n)$，但比归并排序有更小的时间常数。它的基本思想是这样的：

```
int partition(int start, int end)
{
        从a[start..end]中选取一个pivot元素(比如选a[start]为pivot);
        在一个循环中移动a[start..end]的数据，将a[start..end]分成两部分，
        使a[start..mid-1]比pivot元素小,a[mid+1..end]比pivot元素大，
        而a[mid]就是pivot元素;
        return mid;
}

void quicksort(int start, int end)
{
        int mid;
        if (end > start) {
                mid = partition(start, end);
```

```
                quicksort(start, mid-1);
                quicksort(mid+1, end);
        }
}
```

请补完 partition 函数，这个函数有多种写法，请选择时间常数尽可能小的实现方法。想想快速排序在最好和最坏情况下的时间复杂度分别是多少？快速排序在平均情况下的时间复杂度分析起来比较复杂，有兴趣的读者可以查阅参考文献[16]的 7.4.2 节。

2. 总结一下我们见过的算法，哪些算法在平均情况和最坏情况下的时间复杂度差不多？哪些算法在平均情况和最好情况下的时间复杂度差不多？哪些算法的时间复杂度是固定的，分不出最好情况、最坏情况和平均情况？

11.5　线性查找

有些查找问题要用时间复杂度为 O(n)的算法来解决。例如写一个 indexof 函数，从任意输入字符串中找出某个字母的位置并返回这个位置，如果找不到就返回-1：

例 11.3　线性查找

```
#include <stdio.h>

char a[]="hello world";

int indexof(char letter)
{
        int i = 0;
        while (a[i] != '\0') {
                if (a[i] == letter)
                        return i;
                i++;
        }
        return -1;
}

int main(void)
{
         printf("%d %d\n", indexof('o'), indexof('z'));
         return 0;
}
```

这个实现是最直观和最容易想到的，但它是不是最快的算法呢？我们知道插入排序也比归并排序更容易想到，但通常不如归并排序快。那么现在这个问题——给定一个随机排列的序列，找出其中某个元素的位置——有没有比 O(n)更快的算法？比如 O(lg$n$)？请读者思考一下。

习题

1. 实现一个算法，在一组随机排列的数中找出最小的一个。你能想到的最直观的

算法一定是$\Theta(n)$的，想想有没有比$\Theta(n)$更快的算法？

2. 在一组随机排列的数中找出第二小的，这个问题比上一个稍复杂，你能不能想出$\Theta(n)$的算法？

3. 进一步泛化，在一组随机排列的数中找出第 k 小的，这个元素称为 k-th Order Statistic。能想到的最直观的算法肯定是先把这些数排序然后取第 k 个，时间复杂度和排序算法相同，可以是$\Theta(n\lg n)$。这个问题虽然比前两个问题复杂，但它也有平均情况下时间复杂度是$\Theta(n)$的算法，将上一节习题 1 的快速排序算法稍加修改就可以解决这个问题：

```
/* 第 k 小的元素在 start 和 end 之间，找出并返回该元素 */
int order_statistic(int start, int end, int k)
{
        用 partition 函数把序列分成两部分，中间的 pivot 元素是序列中的第 i 个；
        if (k == i)
                返回找到的元素；
        else if (k > i)
                第 k 小的元素在序列后半部分，找出并返回该元素；
        else
                第 k 小的元素在序列前半部分，找出并返回该元素；
}
```

请编程实现这个算法。

11.6 折半查找

如果不是从一组随机的序列里查找，而是从一组排好序的序列里找出某个元素的位置，则可以有更快的算法：

例 11.4 折半查找

```
#include <stdio.h>

#define LEN 8
int a[LEN] = { 1, 2, 2, 2, 5, 6, 8, 9 };

int binarysearch(int number)
{
        int mid, start = 0, end = LEN - 1;

        while (start <= end) {
                mid = (start + end) / 2;
                if (a[mid] < number)
                        start = mid + 1;
                else if (a[mid] > number)
                        end = mid - 1;
                else
                        return mid;
        }
        return -1;
}
```

```
int main(void)
{
        printf("%d\n", binarysearch(5));
        return 0;
}
```

由于这个序列已经从小到大排好序了，每次取中间的元素和待查找的元素比较，如果中间的元素比待查找的元素小，就说明“如果待查找的元素存在，一定位于序列的后半部分”，这样可以把搜索范围缩小到后半部分，然后再次使用这种算法迭代。这种“每次将搜索范围缩小一半”的思想称为折半查找（Binary Search）。思考一下，这个算法的时间复杂度是多少？

这个算法的思想很简单，不是吗？可是参考文献[18]的 4.1 节说作者在课堂上讲完这个算法的思想然后让学生写程序，有 90%的人写出的程序中有各种各样的 Bug，读者不信的话可以不看书自己写一遍试试。这个算法容易出错的地方很多，比如 mid = (start + end) / 2;这一句，在数学概念上其实是 mid = ⌊(start + end) / 2⌋，还有 start = mid + 1;和 end = mid - 1;，如果前者写成了 start = mid;或后者写成了 end = mid;那么很可能会导致死循环（想一想什么情况下会陷入死循环）。

怎样才能保证程序的正确性呢？在第 11.2 节我们讲过借助 Loop Invariant 证明循环的正确性，binarysearch 这个函数的主体也是一个循环，它的 Loop Invariant 可以这样描述：**待查找的元素 number 如果存在于数组 a 之中，那么一定存在于 a[start..end]这个范围之间，换句话说，在这个范围之外的数组 a 的元素中一定不存在 number 这个元素**。以下为了书写方便，我们把这句话表示成 mustbe(start, end, number)。可以一边看算法一边做推理：

```
int binarysearch(int number)
{
        int mid, start = 0, end = LEN - 1;

        /* 假定 a 是排好序的 */
        /* mustbe(start, end, number)，因为 a[start..end]就是整个数组
        a[0..LEN-1] */
        while (start <= end) {
        /* mustbe(start, end, number)，因为一开始进入循环时是正确的，每
         次循环也都维护了这个条件 */
                mid = (start + end) / 2;
                if (a[mid] < number)
                        /* 既然 a 是排好序的，a[start..mid]应该都比 number
                        小，所以 mustbe(mid+1, end, number) */
                        start = mid + 1;
                        /* 维护了 mustbe(start, end, number) */
                else if (a[mid] > number)
                        /* 既然 a 是排好序的，a[mid..end]应该都比 number
                        大，所以 mustbe(start, mid-1, number) */
                        end = mid - 1;
                        /* 维护了 mustbe(start, end, number) */
                else
                        /* a[mid] == number，说明找到了 */
                        return mid;
```

```
         * mustbe(start, end, number)一直被循环维护着，到这里应该仍然成
         立， 在a[start..end]范围之外一定不存在number，
         * 但现在a[start..end]是空序列，在这个范围之外的正是整个数组a，因
         此整个数组a中都不存在number
         */
        return -1;
}
```

注意这个算法有一个非常重要的前提——a 是排好序的。缺了这个前提，“如果a[mid] < number，那么a[start..mid]应该都比number小”这一步推理就不能成立，这个函数就不能正确地完成查找。从更普遍的意义上说，函数的调用者（Caller）和函数的实现者（Callee，被调用者）之间订立了一个契约（Contract），在调用函数之前，Caller 要为 Callee 提供某些条件，比如确保 a 是排好序的，确保a[start..end]都是有效的数组元素而没有访问越界，这称为 Precondition，然后 Callee 对一些 Invariant 进行维护（Maintenance），这些 Invariant 保证了 Callee 在函数返回时能够对 Caller 尽到某些义务，比如确保“如果 number 在数组 a 中存在，一定能找出来并返回它的位置；如果 number 在数组 a 中不存在，一定能返回-1”，这称为 Postcondition。如果每个函数的文档都非常清楚地记录了 Precondition、Maintenance 和 Postcondition 是什么，那么每个函数都可以独立编写和测试，整个系统就会易于维护。这种编程思想是由 Eiffel 语言的设计者 Bertrand Meyer 提出来的，称为 Design by Contract（DbC）。

测试一个函数是否正确需要把 Precondition、Maintenance 和 Postcondition 这三方面都测试到，比如 binarysearch 这个函数，即使它写得非常正确，既维护了 Invariant 也保证了 Postcondition，如果调用它的 Caller 没有保证 Precondition，最后的结果也还是错的。我们编写几个测试用的 Predicate 函数，然后把相关的测试插入到 binarysearch 函数中：

例 11.5 带有测试代码的折半查找

```
#include <stdio.h>
#include <assert.h>

#define LEN 8
int a[LEN] = { 1, 2, 2, 2, 5, 6, 8, 9 };

int is_sorted(void)
{
        int i;
        for (i = 1; i < LEN; i++)
                if (a[i-1] > a[i])
                        return 0;
        return 1;
}

int mustbe(int start, int end, int number)
{
        int i;
        for (i = 0; i < start; i++)
                if (a[i] == number)
                        return 0;
        for (i = end+1; i < LEN; i++)
                if (a[i] == number)
```

```
                        return 0;
        return 1;
}

int contains(int n)
{
        int i;
        for (i = 0; i < LEN; i++)
                if (a[i] == n)
                        return 1;
        return 0;
}

int binarysearch(int number)
{
        int mid, start = 0, end = LEN - 1;

        assert(is_sorted()); /* Precondition */
        while (start <= end) {
                assert(mustbe(start, end, number)); /* Maintenance */
                mid = (start + end) / 2;
                if (a[mid] < number)
                        start = mid + 1;
                else if (a[mid] > number)
                        end = mid - 1;
                else {
                        assert(mid >= start && mid <= end
                        && a[mid] == number); /* Postcondition 1 */
                        return mid;
                }
        }
        assert(!contains(number)); /* Postcondition 2 */
        return -1;
}

int main(void)
{
        printf("%d\n", binarysearch(5));
        return 0;
}
```

assert 是头文件 assert.h 中的一个宏定义，执行到 assert(is_sorted())这句时，如果 is_sorted()返回值为真，则当什么事都没发生过，继续往下执行；如果 is_sorted()返回值为假（例如改变数组的排列顺序），则报错退出程序：

```
$ ./a.out
a.out: main.c:41: binarysearch: Assertion `is_sorted()' failed.
Aborted
```

在代码中适当的地方使用断言（Assertion）可以有效地帮助我们测试程序。也许有人会问：我们用几个测试函数来测试 binarysearch，那么这几个测试函数又用什么来测试呢？在实际工作中我们要测试的代码绝不会像 binarysearch 这么简单，而我们编写的测试函数往往都很简单，比较容易保证正确性，也就是用简单的、不容易出错的代码去测试复杂的、容易出错的代码。

测试代码只在开发和调试时有用，如果正式发布（Release）的软件也要运行这些测试代码就会严重影响性能了，如果在包含 assert.h 之前定义一个 NDEBUG 宏（表示 No Debug），就可以禁用 assert.h 中的 assert 宏定义，这样代码中的所有 assert

测试都不起作用了：

```
#define NDEBUG
#include <stdio.h>
#include <assert.h>
...
```

注意NDEBUG和我们以前使用的宏定义有点不同，例如#define N 20将N定义为20，在预处理时把代码中所有的标识符N替换成20，而#define NDEBUG把NDEBUG定义为空，在预处理时把代码中所有的标识符NDEBUG替换成空。这样的宏定义主要是为了用#ifdef等预处理指示测试它定义过没有，而不是为了做替换，所以定义成什么值都无所谓，一般定义成空就足够了。

还有另一种办法，不必修改源文件，在编译命令行加上选项-DNDEBUG就相当于在源文件开头定义了NDEBUG宏。宏定义和预处理到第20章再详细解释，在第20.4节将给出assert.h的一种实现。

习题

1. 本节的折半查找算法有一个特点：如果待查找的元素在数组中有多个则返回其中任意一个，以本节定义的数组int a[8] = { 1, 2, 2, 2, 5, 6, 8, 9 };为例，如果调用binarysearch(2)则返回3，即a[3]，而有些场合下要求这样的查找返回a[1]，也就是说，如果待查找的元素在数组中有多个则返回第一个。请修改折半查找算法实现这一特性。

2. 编写一个函数double mysqrt(double y)求y的正平方根，参数y是正实数。我们用折半查找来找这个平方根，在从0到y之间必定有一个取值是y的平方根，如果我们查找的数x比y的平方根小，则$x^2<y$，如果我们查找的数x比y的平方根大，则$x^2>y$，我们可以据此缩小查找范围，当我们查找的数足够准确时（比如满足$|x^2-y|<0.001$），就可以认为找到了y的平方根。思考一下这个算法需要迭代多少次？迭代次数的多少由什么因素决定？

3. 编写一个函数double mypow(double x, int n)求x的n次方，参数n是正整数。最简单的算法是：

```
double product = 1;
for (i = 0; i < n; i++)
    product *= x;
```

这个算法的时间复杂度是$\Theta(n)$。其实有更好的办法，比如mypow(x, 8)，第一次循环算出$x \cdot x=x^2$，第二次循环算出$x^2 \cdot x^2=x^4$，第三次循环算出$x^4 \cdot x^4=x^8$。这样只需要三次循环，时间复杂度是$\Theta(\lg n)$。思考一下如果n不是2的整数次幂应该怎么处理。请分别用递归和循环实现这个算法。

从以上几题可以看出，折半查找的思想有非常广泛的应用，不仅限于从一组排好序的元素中找出某个元素的位置，还可以解决很多类似的问题。参考文献[18]对于折半查找的各种应用和优化技巧有非常详细的介绍。

第 12 章

栈与队列

12.1 数据结构的概念

数据结构（Data Structure）是数据的组织方式。程序中用到的数据都不是孤立的，而是有相互联系的，根据访问数据的需求不同，同样的数据可以有多种不同的组织方式。以前学过的复合类型也可以看作数据的组织方式，把同一类型的数据组织成数组，或者把描述同一对象的各成员组织成结构体。数据的组织方式包含了存储方式和访问方式这两层意思，二者是紧密联系的。例如，数组的各元素是一个挨一个存储的，并且每个元素的大小相同，因此数组可以提供按下标访问的方式，结构体的各成员也是一个挨一个存储的，但是每个成员的大小不同，所以只能用.运算符加成员名来访问，而不能按下标访问。

本章主要介绍栈和队列这两种数据结构以及它们的应用。从本章的应用实例可以看出，一个问题中数据的存储方式和访问方式就决定了解决问题可以采用什么样的算法，要设计一个算法就要同时设计相应的数据结构来支持这种算法。所以 Pascal 语言的设计者 Niklaus Wirth 提出：**算法+数据结构=程序**（详见参考文献[20]）。

12.2 堆栈

在第 5.3 节中我们已经对堆栈这种数据结构有了初步认识。堆栈是一组元素的集合，类似于数组，不同之处在于，数组可以按下标随机访问，这次访问 a[5]下次可以访问 a[1]，但是堆栈的访问规则被限制为 Push 和 Pop 两种操作，Push（入栈或压栈）向栈顶添加元素，Pop（出栈或弹出）则取出当前栈顶的元素，也就是说，只能访问栈顶元素而不能访问栈中其他元素。如果所有元素的类型相同，堆栈的存储也可以用数组来实现，访问操作可以通过函数接口提供。看以下的示例程序。

例 12.1 用堆栈实现倒序打印

```
#include <stdio.h>

char stack[512];
int top = 0;

void push(char c)
```

```
{
        stack[top++] = c;
}

char pop(void)
{
        return stack[--top];
}

int is_empty(void)
{
        return top == 0;
}

int main(void)
{
        push('a');
        push('b');
        push('c');

        while (!is_empty())
                putchar(pop());
        putchar('\n');

        return 0;
}
```

运行结果是 cba。运行过程图示如图 12.1 所示。

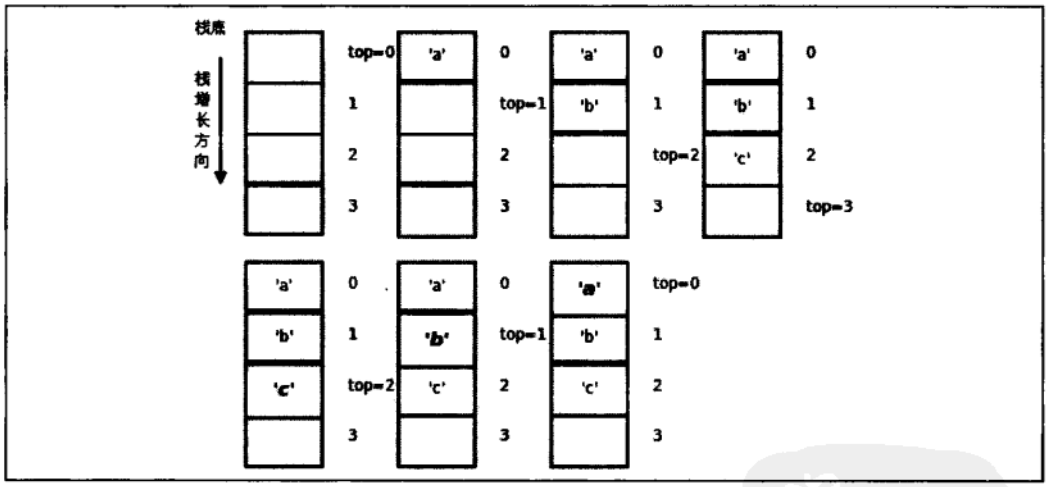


图 12.1 用堆栈实现倒序打印

数组 stack 是堆栈的存储空间，变量 top 总是保存数组中栈顶的下一个元素的下标，我们说“top 总是指向栈顶的下一个元素”，或者把 top 叫做栈顶指针（Pointer）。在第 11.2 节中介绍了 Loop Invariant 的概念，可以用它检验循环的正确性，这里的“top 总是指向栈顶的下一个元素”其实也是一种 Invariant，Push 和 Pop 操作总是维持这个条件不变，这种 Invariant 描述的对象是一个数据结构而不是一个循环，在 DbC 中称为 Class Invariant。Pop 操作的语义是取出栈顶元素，但上例的实现其实并没有清除原来的栈顶元素，只是把 top 指针移动了一下，原来的栈顶元素仍然存在那里，这就足够了，因为此后通过 Push 和 Pop 操作不可能再访问到已

经取出的元素，下次 Push 操作就会覆盖它。putchar 函数的作用是把一个字符打印到屏幕上，和 printf 的%c 作用相同。布尔函数 is_empty 的作用是防止 Pop 操作访问越界。这里我们预留了足够大的栈空间（512 个元素），其实严格来说 Push 操作之前也应该检查栈是否满了。

在 main 函数中，入栈的顺序是'a'、'b'、'c'，而出栈打印的顺序却是'c'、'b'、'a'，最后入栈的'c'最早出来，因此堆栈这种数据结构的特点可以概括为 LIFO（Last In First Out，后进先出）。我们也可以写一个递归函数做倒序打印，利用函数调用的栈帧实现后进先出：

例 12.2　用递归实现倒序打印

```
#include <stdio.h>
#define LEN 3

char buf[LEN] = {'a', 'b', 'c'};

void print_backward(int pos)
{
        if (pos == LEN)
                return;
        print_backward(pos+1);
        putchar(buf[pos]);
}

int main(void)
{
        print_backward(0);
        putchar('\n');

        return 0;
}
```

也许你会说，又是堆栈又是递归的，倒序打印一个数组犯得着这么大动干戈吗？写一个简单的循环不就行了：

```
for (i = LEN-1; i >= 0; i--)
        putchar(buf[i]);
```

对于数组来说确实没必要搞这么复杂，因为数组既可以从前向后访问也可以从后向前访问，甚至可以随机访问，但有些数据结构的访问并没有这么自由，下一节你就会看到这样的数据结构。

12.3　深度优先搜索

现在我们用堆栈解决一个有意思的问题，定义一个二维数组：

```
int maze[5][5] = {
        0, 1, 0, 0, 0,
        0, 1, 0, 1, 0,
        0, 0, 0, 0, 0,
        0, 1, 1, 1, 0,
```

```
        0, 0, 0, 1, 0,
};
```

它表示一个迷宫，其中的1表示墙壁，0表示可以走的路，只能横着走或竖着走，不能斜着走，要求编程序找出从左上角到右下角的路线。程序如下：

例12.3 用深度优先搜索解迷宫问题

```
#include <stdio.h>

#define MAX_ROW 5
#define MAX_COL 5

struct point { int row, col; } stack[512];
int top = 0;

void push(struct point p)
{
        stack[top++] = p;
}

struct point pop(void)
{
        return stack[--top];
}

int is_empty(void)
{
        return top == 0;
}

int maze[MAX_ROW][MAX_COL] = {
        0, 1, 0, 0, 0,
        0, 1, 0, 1, 0,
        0, 0, 0, 0, 0,
        0, 1, 1, 1, 0,
        0, 0, 0, 1, 0,
};

void print_maze(void)
{
        int i, j;
        for (i = 0; i < MAX_ROW; i++) {
                for (j = 0; j < MAX_COL; j++)
                        printf("%d ", maze[i][j]);
                putchar('\n');
        }
        printf("*********\n");
}

struct point predecessor[MAX_ROW][MAX_COL] = {
        {{-1,-1}, {-1,-1}, {-1,-1}, {-1,-1}, {-1,-1}},
        {{-1,-1}, {-1,-1}, {-1,-1}, {-1,-1}, {-1,-1}},
        {{-1,-1}, {-1,-1}, {-1,-1}, {-1,-1}, {-1,-1}},
        {{-1,-1}, {-1,-1}, {-1,-1}, {-1,-1}, {-1,-1}},
        {{-1,-1}, {-1,-1}, {-1,-1}, {-1,-1}, {-1,-1}},
};
```

```
void visit(int row, int col, struct point pre)
{
        struct point visit_point = { row, col };
        maze[row][col] = 2;
        predecessor[row][col] = pre;
        push(visit_point);
}

int main(void)
{
        struct point p = { 0, 0 };

        maze[p.row][p.col] = 2;
        push(p);

        while (!is_empty()) {
                p = pop();
                if (p.row == MAX_ROW - 1  /* goal */
                && p.col == MAX_COL - 1)
                        break;
                if (p.col+1 < MAX_COL    /* right */
                && maze[p.row][p.col+1] == 0)
                        visit(p.row, p.col+1, p);
                if (p.row+1 < MAX_ROW    /* down */
                && maze[p.row+1][p.col] == 0)
                        visit(p.row+1, p.col, p);
                if (p.col-1 >= 0         /* left */
                && maze[p.row][p.col-1] == 0)
                        visit(p.row, p.col-1, p);
                if (p.row-1 >= 0         /* up */
                && maze[p.row-1][p.col] == 0)
                        visit(p.row-1, p.col, p);
                print_maze();
    }
    if (p.row == MAX_ROW - 1 && p.col == MAX_COL - 1) {
            printf("(%d, %d)\n", p.row, p.col);
            while (predecessor[p.row][p.col].row != -1) {
                    p = predecessor[p.row][p.col];
                    printf("(%d, %d)\n", p.row, p.col);
            }
    } else
            printf("No path!\n");

    return 0;
}
```

运行结果如下：

```
2 1 0 0 0
2 1 0 1 0
0 0 0 0 0
0 1 1 1 0
0 0 0 1 0
*********
2 1 0 0 0
2 1 0 1 0
2 0 0 0 0
0 1 1 1 0
0 0 0 1 0
```

```
*********
2 1 0 0 0
2 1 0 1 0
2 2 0 0 0
2 1 1 1 0
0 0 0 1 0
*********
2 1 0 0 0
2 1 0 1 0
2 2 0 0 0
2 1 1 1 0
2 0 0 1 0
*********
2 1 0 0 0
2 1 0 1 0
2 2 0 0 0
2 1 1 1 0
2 2 0 1 0
*********
2 1 0 0 0
2 1 0 1 0
2 2 0 0 0
2 1 1 1 0
2 2 2 1 0
*********
2 1 0 0 0
2 1 0 1 0
2 2 0 0 0
2 1 1 1 0
2 2 2 1 0
*********
2 1 0 0 0
2 1 0 1 0
2 2 2 0 0
2 1 1 1 0
2 2 2 1 0
*********
2 1 0 0 0
2 1 2 1 0
2 2 2 2 0
2 1 1 1 0
2 2 2 1 0
*********
2 1 2 0 0
2 1 2 1 0
2 2 2 2 0
2 1 1 1 0
2 2 2 1 0
*********
2 1 2 2 0
2 1 2 1 0
2 2 2 2 0
2 1 1 1 0
2 2 2 1 0
*********
2 1 2 2 2
2 1 2 1 0
2 2 2 2 0
2 1 1 1 0
2 2 2 1 0
```

```
*********
2 1 2 2 2
2 1 2 1 2
2 2 2 2 0
2 1 1 1 0
2 2 2 1 0
*********
2 1 2 2 2
2 1 2 1 2
2 2 2 2 2
2 1 1 1 0
2 2 2 1 0
*********
2 1 2 2 2
2 1 2 1 2
2 2 2 2 2
2 1 1 1 2
2 2 2 1 0
*********
2 1 2 2 2
2 1 2 1 2
2 2 2 2 2
2 1 1 1 2
2 2 2 1 2
*********
(4, 4)
(3, 4)
(2, 4)
(1, 4)
(0, 4)
(0, 3)
(0, 2)
(1, 2)
(2, 2)
(2, 1)
(2, 0)
(1, 0)
(0, 0)
```

这次堆栈里的元素是结构体类型的，用来表示迷宫中一个点的 *X* 和 *Y* 坐标。我们用一个新的数据结构保存走迷宫的路线，每个走过的点都有一个前趋（Predecessor）点，表示是从哪儿走到当前点的，比如 predecessor[4][4]是坐标为(3, 4)的点，就表示从(3, 4)走到了(4, 4)，一开始 predecessor 的各元素初始化为无效坐标(-1, -1)。在迷宫中探索路线的同时就把路线保存在 predecessor 数组中，已经走过的点在 maze 数组中记为 2 防止重复走，最后找到终点时就根据 predecessor 数组保存的路线从终点打印到起点。为了帮助理解，我把这个算法改写成伪代码（Pseudocode）如下：

```
将起点标记为已走过并压栈;
while (栈非空) {
        从栈顶弹出一个点 p;
        if (p 这个点是终点)
                break;
        否则沿右、下、左、上四个方向探索相邻的点
        if (和 p 相邻的点有路可走，并且还没走过)
                将相邻的点标记为已走过并压栈，它的前趋就是 p 点;
```

```
    }
    if (p 点是终点) {
            打印 p 点的坐标;
            while (p 点有前趋) {
                p 点 = p 点的前趋;
                打印 p 点的坐标;
            }
    } else
            没有路线可以到达终点;
```

我在 while 循环的末尾插了打印语句，每探索一步都打印出当前迷宫的状态（标记了哪些点），从打印结果可以看出这种搜索算法的特点是：每次探索完各个方向相邻的点之后，取其中一个相邻的点走下去，一直走到无路可走了再退回来，取另一个相邻的点再走下去。这称为深度优先搜索（DFS，Depth First Search）。探索迷宫和堆栈变化的过程如图 12.2 所示。

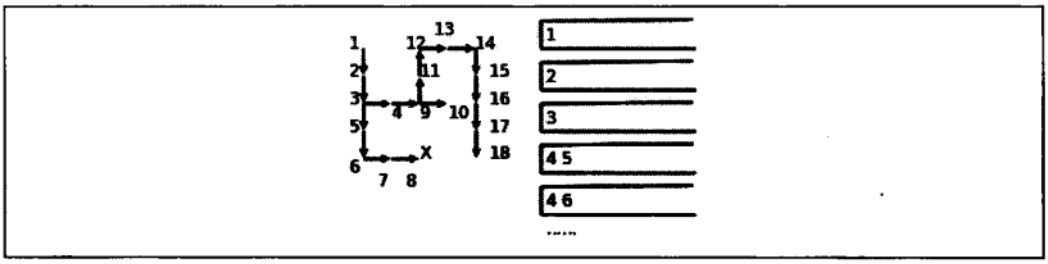


图 12.2 深度优先搜索

图中各点的编号表示探索顺序，堆栈中保存的应该是坐标，我在画图时为了直观就把各点的编号写在堆栈里了。可见正是堆栈后进先出的性质使这个算法具有了深度优先的特点。如果在探索问题的解时走进了死胡同，则需要退回来从另一条路继续探索，这种思想称为回溯（Backtrack），一个典型的例子是很多编程书上都会讲的八皇后问题。

最后我们打印终点的坐标并通过 predecessor 数据结构找到它的前趋，这样顺藤摸瓜一直打印到起点。那么能不能从起点到终点正向打印路线呢？在上一节我们看到，数组支持随机访问也支持顺序访问，如果在一个循环里打印数组，既可以正向打印也可以反向打印。但 predecessor 这种数据结构却有很多限制：

1．不能随机访问一条路线上的任意点，只能通过一个点找到另一个点，通过另一个点再找第三个点，因此只能顺序访问。

2．每个点只知道它的前趋是谁，而不知道它的后继（Successor）是谁，所以只能反向顺序访问。

可见，**有什么样的数据结构就决定了可以用什么样的算法**。那为什么不再建一个 successor 数组来保存每个点的后继呢？从 DFS 算法的过程可以看出，虽然每个点的前趋只有一个，后继却不止一个，如果我们为每个点只保存一个后继，则无法保证这个后继指向正确的路线。由此可见，**有什么样的算法就决定了可以用什么样的数据结构**。设计算法和设计数据结构这两件工作是紧密联系的。

习题

1．修改本节的程序，要求从起点到终点正向打印路线。你能想到几种办法？

2．本节程序中 predecessor 这个数据结构占用的存储空间太多了，改变它的存储方式可以节省空间，想想该怎么改。

3．上一节我们实现了一个基于堆栈的程序，然后改写成递归程序，用函数调用的栈帧替代自己实现的堆栈。本节的 DFS 算法也是基于堆栈的，请把它改写成递归程序，这样改写可以避免使用 predecessor 数据结构，想想该怎么做。

4．本节的程序只要找到一条路线就退出了，而不再回溯，如果要求找到从起点到终点的所有路线，想想该怎么做。

12.4　队列与广度优先搜索

队列也是一组元素的集合，也提供两种基本操作：Enqueue（入队）将元素添加到队尾，Dequeue（出队）从队头取出元素并返回。就像排队买票一样，先来先服务，先入队的也是先出队的，这种方式称为 FIFO（First In First Out，先进先出），有时候队列本身也被称为 FIFO。

下面我们用队列解决迷宫问题。程序如下：

例 12.4　用广度优先搜索解迷宫问题

```
#include <stdio.h>

#define MAX_ROW 5
#define MAX_COL 5

struct point { int row, col, predecessor; } queue[512];
int head = 0, tail = 0;

void enqueue(struct point p)
{
        queue[tail++] = p;
}

struct point dequeue(void)
{
        return queue[head++];
}

int is_empty(void)
{
        return head == tail;
}

int maze[MAX_ROW][MAX_COL] = {
        0, 1, 0, 0, 0,
        0, 1, 0, 1, 0,
        0, 0, 0, 0, 0,
```

```
        0, 1, 1, 1, 0,
        0, 0, 0, 1, 0,
};

void print_maze(void)
{
        int i, j;
        for (i = 0; i < MAX_ROW; i++) {
                for (j = 0; j < MAX_COL; j++)
                        printf("%d ", maze[i][j]);
                putchar('\n');
        }
        printf("*********\n");
}

void visit(int row, int col)
{
        struct point visit_point = { row, col, head-1 };
        maze[row][col] = 2;
        enqueue(visit_point);
}

int main(void)
{
        struct point p = { 0, 0, -1 };

        maze[p.row][p.col] = 2;
        enqueue(p);

        while (!is_empty()) {
                p = dequeue();
                if (p.row == MAX_ROW - 1  /* goal */
                && p.col == MAX_COL - 1)
                        break;
                if (p.col+1 < MAX_COL    /* right */
                && maze[p.row][p.col+1] == 0)
                        visit(p.row, p.col+1);
                if (p.row+1 < MAX_ROW    /* down */
                && maze[p.row+1][p.col] == 0)
                        visit(p.row+1, p.col);
                if (p.col-1 >= 0         /* left */
                && maze[p.row][p.col-1] == 0)
                        visit(p.row, p.col-1);
                if (p.row-1 >= 0         /* up */
                && maze[p.row-1][p.col] == 0)
                        visit(p.row-1, p.col);
                print_maze();
        }
        if (p.row == MAX_ROW - 1 && p.col == MAX_COL - 1) {
                printf("(%d, %d)\n", p.row, p.col);
                while (p.predecessor != -1) {
                        p = queue[p.predecessor];
                        printf("(%d, %d)\n", p.row, p.col);
                }
        } else
                printf("No path!\n");

        return 0;
}
```

运行结果如下：

```
2 1 0 0 0
2 1 0 1 0
0 0 0 0 0
0 1 1 1 0
0 0 0 1 0
*********
2 1 0 0 0
2 1 0 1 0
2 0 0 0 0
0 1 1 1 0
0 0 0 1 0
*********
2 1 0 0 0
2 1 0 1 0
2 2 0 0 0
2 1 1 1 0
0 0 0 1 0
*********
2 1 0 0 0
2 1 0 1 0
2 2 2 0 0
2 1 1 1 0
0 0 0 1 0
*********
2 1 0 0 0
2 1 0 1 0
2 2 2 0 0
2 1 1 1 0
2 0 0 1 0
*********
2 1 0 0 0
2 1 2 1 0
2 2 2 2 0
2 1 1 1 0
2 0 0 1 0
*********
2 1 0 0 0
2 1 2 1 0
2 2 2 2 0
2 1 1 1 0
2 2 0 1 0
*********
2 1 0 0 0
2 1 2 1 0
2 2 2 2 2
2 1 1 1 0
2 2 0 1 0
*********
2 1 2 0 0
2 1 2 1 0
2 2 2 2 2
2 1 1 1 0
2 2 0 1 0
*********
2 1 2 0 0
2 1 2 1 0
2 2 2 2 2
```

```
2 1 1 1 0
2 2 2 1 0
*********
2 1 2 0 0
2 1 2 1 2
2 2 2 2 2
2 1 1 1 2
2 2 2 1 0
*********
2 1 2 2 0
2 1 2 1 2
2 2 2 2 2
2 1 1 1 2
2 2 2 1 0
*********
2 1 2 2 0
2 1 2 1 2
2 2 2 2 2
2 1 1 1 2
2 2 2 1 0
*********
2 1 2 2 0
2 1 2 1 2
2 2 2 2 2
2 1 1 1 2
2 2 2 1 2
*********
2 1 2 2 2
2 1 2 1 2
2 2 2 2 2
2 1 1 1 2
2 2 2 1 2
*********
2 1 2 2 2
2 1 2 1 2
2 2 2 2 2
2 1 1 1 2
2 2 2 1 2
*********
(4, 4)
(3, 4)
(2, 4)
(2, 3)
(2, 2)
(2, 1)
(2, 0)
(1, 0)
(0, 0)
```

其实仍然可以像例 12.3 一样用 predecessor 数组表示每个点的前趋，但我想换一种更方便的数据结构，直接在每个点的结构体中加一个成员表示前趋：

```
struct point { int row, col, predecessor; } queue[512];
int head = 0, tail = 0;
```

变量 head 和 tail 是队头和队尾指针，head 总是指向队头，tail 总是指向队尾的下一个元素。每个点的 predecessor 成员也是一个指针，指向它的前趋在 queue 数组

中的位置，如图 12.3 所示。

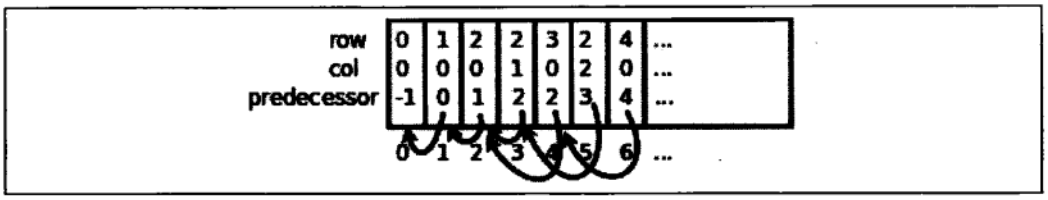


图 12.3　广度优先搜索的队列数据结构

为了帮助理解，我把这个算法改写成如下伪代码：

```
将起点标记为已走过并入队;
while (队列非空) {
        出队一个点 p;
        if (p 这个点是终点)
                break;
        否则沿右、下、左、上四个方向探索相邻的点
        if (和 p 相邻的点有路可走，并且还没走过)
                将相邻的点标记为已走过并入队，它的前趋就是刚出队的 p 点;
}
if (p 点是终点) {
        打印 p 点的坐标;
        while (p 点有前趋) {
                p 点 = p 点的前趋;
                打印 p 点的坐标;
        }
} else
        没有路线可以到达终点;
```

从打印的搜索过程可以看出，这个算法的特点是沿各个方向同时展开搜索，每个可以走通的方向轮流往前走一步，这称为广度优先搜索（BFS，Breadth First Search）。探索迷宫和队列变化的过程如图 12.4 所示。

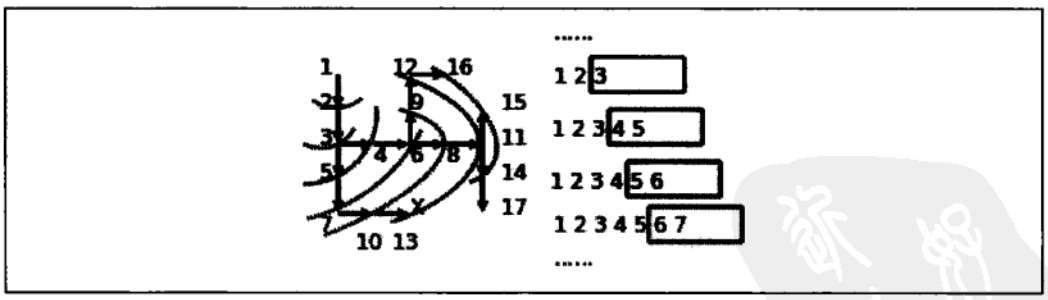


图 12.4　广度优先搜索

广度优先是一种步步为营的策略，每次都从各个方向探索一步，将前线推进一步，图 12.4 中的虚线就表示这个前线，队列中的元素总是由前线的点组成的，可见正是队列先进先出的性质使这个算法具有了广度优先的特点。广度优先搜索还有一个特点是可以找到从起点到终点的最短路径，而深度优先搜索找到的不一定是最短路径，比较本节和上一节程序的运行结果可以看出这一点，想一想为什么。

习题

1. 本节的例子直接在队列元素中加一个指针成员表示前趋，想一想为什么上一节的例 12.3 不能采用这种方法表示前趋？

2. 本节例子中给队列分配的存储空间是 512 个元素，其实没必要这么多，那么解决这个问题至少要分配多少个元素的队列空间呢？跟什么因素有关？

12.5 环形队列

比较例 12.3 的栈操作和例 12.4 的队列操作可以发现，栈操作的 top 指针在 Push 时增大而在 Pop 时减小，栈空间是可以重复利用的，而队列的 head、tail 指针都在一直增大，虽然前面的元素已经出队了，但它所占的存储空间却不能重复利用。在例 12.4 的解法中，出队的元素仍然有用，保存着走过的路径和每个点的前趋，但大多数程序并不是这样使用队列的，一般情况下出队的元素就不再有保存价值了，这些元素的存储空间应该回收利用，由此想到把队列改造成环形队列（Circular Queue）：把 queue 数组想象成一个圈，head 和 tail 指针仍然是一直增大的，当指到数组末尾时就自动回到数组开头，就像两个人围着操场赛跑，沿着它们跑的方向看，从 head 到 tail 之间是队列的有效元素，从 tail 到 head 之间是空的存储位置，如果 head 追上 tail 就表示队列空了，如果 tail 追上 head 就表示队列的存储空间满了，如图 12.5 所示。

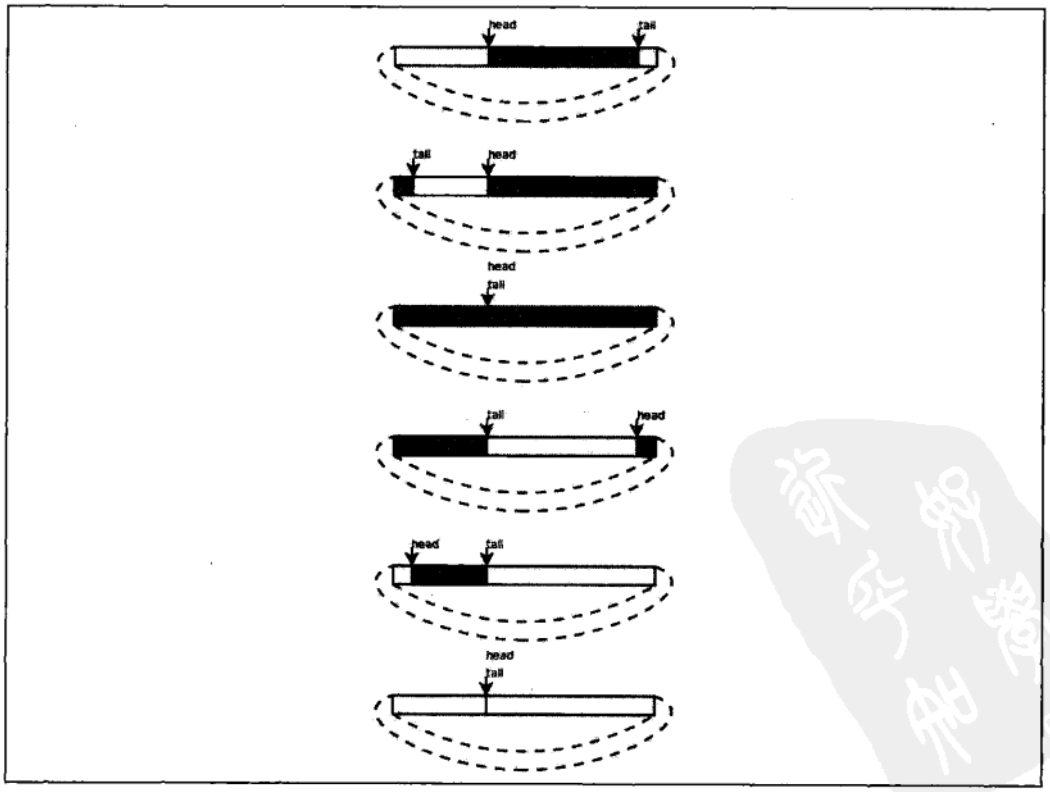


图 12.5 环形队列

注意图 12.5 中的第三个小图和最后一个小图，“head 和 tail 指向相同的位置”既可能表示“队列空”也可能表示“队列满”，这是肯定不行的，在实现环形队列时必须要区分这两种状态，请思考一下可以用哪些办法来区分。

习题

1．现在把迷宫问题的要求改一下，只要求程序给出最后结论就可以了，回答“有路能到达终点”或者“没有路能到达终点”，而不需要把路径打印出来。请把例 12.4 改用环形队列实现，然后试验一下解决这个问题至少需要分配多少个元素的队列空间。

本阶段总结

善于学习的人都应该善于总结。本书的编排顺序充分考虑到知识的前后依赖关系，保证在讲解每个新知识点的时候都只用到前面章节讲过的知识，但正因为如此，很多相互关联的知识点被拆散到多个章节中了。我们一章一章地纵向学习过来之后，应该理出几个横切面，把拆散到各章节中的知识点串起来。请从以下几个方面整理和复习。

1．C 的语法规则

- 源文件中所有函数定义之外可以出现哪些语法元素？
- 函数定义之中可以出现哪些语法元素？
- 语句有哪几种？
- 哪些语法元素需要遵循标识符的命名规则？
- 表达式由哪些语法元素组成？
- 到目前为止学过哪些运算符？它们的优先级和结合性是怎样的？
- 哪些运算符取操作数的左值？哪些运算符有 Side Effect？
- 哪些运算符的操作数必须是整型？哪些运算符的操作数必须是算术类型？哪些运算符的操作数必须是标量类型？
- 哪些表达式可以做左值？哪些表达式只能做右值？
- 哪些地方必须用常量表达式？哪些地方必须用整数常量表达式？

2．思维方法与编程思想

- 以概念为中心，第 1.1 节
- 组合规则，第 2.5 节
- Least Surprise，第 3.3 节
- 充分条件与必要条件，第 3.4 节
- 封装，第 4.2 节
- 布尔逻辑，第 4.3 节
- 递归，第 5.3 节

- 函数式编程，第 6.1 节
- 迭代（第 6 章）与增量式求解（第 11.2 节）
- 抽象，第 7.2 节
- 避免硬编码，第 8.2 节
- 数据驱动，第 8.5 节
- 分而治之，第 11.4 节
- 折半查找，第 11.6 节
- 回溯，例 12.3

3．调试方法

- 编译错误、运行时错误与语义错误，第 1.3 节
- 增量式开发，第 5.2 节
- 打印语句与 Scaffold，第 5.2 节
- gdb，第 10 章
- DbC 与 Assertion，第 11.6 节

下篇 C 语言本质

第 13 章

计算机中数的表示

13.1 为什么计算机用二进制计数

人类的计数方式通常是“逢十进一”，称为十进制（Decimal），大概因为人有十个手指，所以十进制是最自然的计数方式，很多民族的语言文字中都有十个数字，而阿拉伯数字 0～9 是目前最广泛采用的。

计算机是用数字电路搭成的，数字电路中只有 1 和 0 两种状态，所以对计算机来说二进制（Binary）是最自然的计数方式。根据“逢二进一”的原则，十进制的 1、2、3、4 分别对应二进制的 1、10、11、100。二进制的一位数字称为一个位（bit）[①]，三个 bit 能够表示的最大的二进制数是 111，也就是十进制的 7。不管用哪种计数方式，数的大小并没有变，十进制的 1+1 等于 2，二进制的 1+1 等于 10，二进制的 10 和十进制的 2 大小是相等的。事实上，计算机采用如图 13.1 所示的逻辑电路计算两个 bit 的加法：

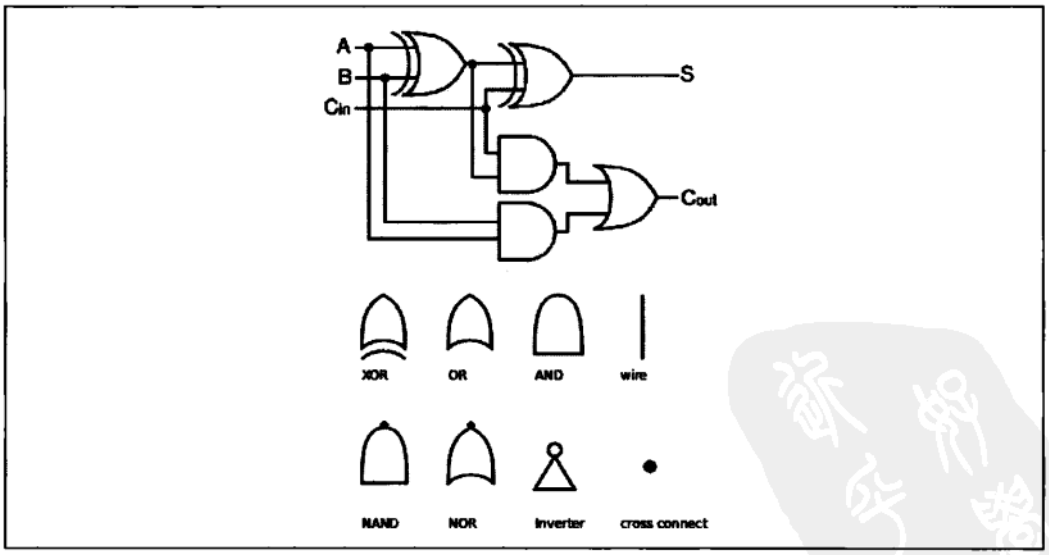


图 13.1 1-bit Full Adder

图 13.1 的上半部分（出自 Wikipedia）的电路称为一位全加器（1-bit Full Adder），

① bit 通常首字母小写，因为 bit 简写为 b，而 Byte 简写为 B。

图的下半部分是一些逻辑电路符号的图例。我们首先解释这些图例，逻辑电路由门电路（Gate）和导线（Wire）组成，同一条导线上在某一时刻的电压值只能是高和低两种状态之一，分别用 1 和 0 表示。如果两条导线短接在一起则它们的电压值相同，在接点处画一个黑点，如果接点处没有画黑点则表示这两条线并没有短接在一起，只是在画图时无法避免交叉。导线的电压值进入门电路的输入端，经过逻辑运算后在门电路的输出端输出运算结果的电压值，任何复杂的加减乘除运算都可以分解成简单的逻辑运算。AND、OR 和 NOT 运算在第 4.3 节中讲过了，这三种逻辑运算分别用与门、或门和反相器（Inverter）实现。另外几种逻辑运算在这里补充一下。异或（XOR，eXclusive OR）运算的真值表如表 13.1 所示。

表 13.1　XOR 的真值表

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

用一句话概括就是：两个操作数相同则结果为 0，两个操作数不同则结果为 1。与非（NAND）和或非（NOR）运算就是在与、或运算的基础上取反，其真值表分别如表 13.2 和表 13.3 所示。

表 13.2　NAND 的真值表

A	B	A NAND B
0	0	1
0	1	1
1	0	1
1	1	0

表 13.3　NOR 的真值表

A	B	A NOR B
0	0	1
0	1	0
1	0	0
1	1	0

如果把与门、或门和反相器组合来实现 NAND 和 NOR 运算，则电路过于复杂了，因此逻辑电路中通常有专用的与非门和或非门。现在我们看看图 13.1 中的 AND、OR、XOR 是怎么实现两个 bit 的加法的。A、B 是两个加数，C_{in} 是低位传上来的进位（Carry），相当于三个加数求和，三个加数都是 0 则结果为 0，三个加数都是 1 则结果为 11，即输出位 S 是 1，产生的进位 C_{out} 也是 1。下面根据加法的规则用真值表列出所有可能的情况，如表 13.4 所示。

表 13.4 1-bit Full Adder 的真值表

A	B	C_{in}	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

请读者对照电路图验证一下真值表是否正确。如果把很多个一位全加器串接起来，就成了多位加法器，如图 13.2 所示（该图出自 Wikipedia）：

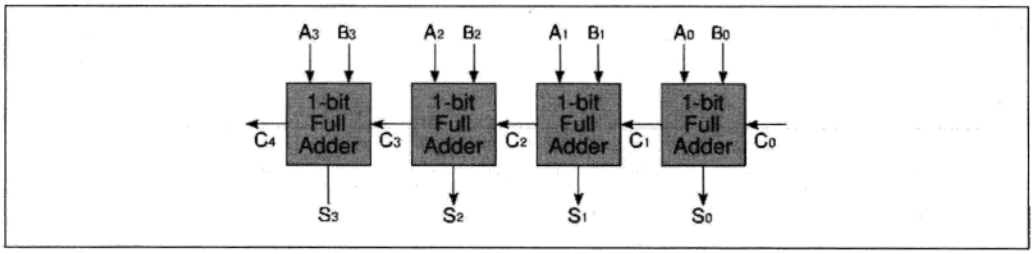


图 13.2 4-bit Ripple Carry Adder

图中的一位全加器用方框表示，上一级全加器的 C_{out} 连接到下一级全加器的 C_{in}，让进位像涟漪一样一级一级传开，所以叫做 Ripple Carry Adder，这样就可以把两个 4 bit 二进制数 $A_3A_2A_1A_0$ 和 $B_3B_2B_1B_0$ 加起来了。在这里介绍 Ripple Carry Adder 只是为了让读者理解计算机是如何通过逻辑运算来做算术运算的，实际上这种加法器效率很低，只能加完了一位再加下一位，更实用、更复杂的加法器可以多个位一起计算，有兴趣的读者可查阅参考文献[9]的 5.4 节。

13.2 不同进制之间的换算

在十进制中，个位的 1 代表 10^0=1，十位的 1 代表 10^1=10，百位的 1 代表 10^2=100，所以

$$123=1\times10^2+2\times10^1+3\times10^0$$

同样道理，在二进制中，个位的 1 代表 2^0=1，十位的 1 代表 2^1=2，百位的 1 代表 2^2=4，所以

$$(A_3A_2A_1A_0)_2=A_3\times2^3+A_2\times2^2+A_1\times2^1+A_0\times2^0$$

如果二进制和十进制数出现在同一个等式中，为了区别我们用$(A_3A_2A_1A_0)_2$这种形式表示 $A_3A_2A_1A_0$ 是二进制数，每个数字只能是 0 或 1，其他没有套括号加下标的

数仍表示十进制数。对于$(A_3A_2A_1A_0)_2$这样一个二进制数，最左边的 A_3 位称为最高位（Most Significant Bit，MSB），最右边的 A_0 位称为最低位（Least Significant Bit，LSB）。以后我们遵循这样的惯例：**LSB 称为第 0 位而不是第 1 位，所以如果一个数是 32 位的，则 MSB 是第 31 位**。上式就是从二进制到十进制的换算公式。作为练习，请读者算一下$(1011)_2$和$(1111)_2$换算成十进制分别是多少。

下面来看十进制数怎么换算成二进制数。我们知道

$$13=1\times2^3+1\times2^2+0\times2^1+1\times2^0$$

所以 13 换算成二进制数应该是$(1101)_2$。问题是怎么把 13 分解成等号右边的形式呢？注意到等号右边可以写成

$$13=(((0\times2+1_3)\times2+1_2)\times2+0_1)\times2+1_0$$

我们将 13 反复除以 2 取余数就可以提取出上式中的 1101 四个数字，为了让读者更容易看清楚是哪个 1 和哪个 0，上式和下式中对应的数字都加了下标：

$$13\div2=6...1_0$$
$$6\div2=3...0_1$$
$$3\div2=1...1_2$$
$$1\div2=0...1_3$$

把这四步得到的余数按相反的顺序排列就是 13 的二进制表示，因此这种方法称为除二反序取余法。

计算机用二进制表示数，程序员也必须习惯使用二进制，但二进制数写起来太啰嗦了，所以通常将二进制数分成每三位一组或者每四位一组，每组用一个数字表示。比如把$(10100011)_2$从最低位开始每三位分成一组，分别是 011、100、10（从高位到低位排列则是 10、100、011），然后把每组写成一个 0～7 的数字，就是$(243)_8$，这种表示法的特点是逢八进一，称为八进制（Octal）。类似地，我们也可以把$(10100011)_2$按每四位分成一组，即 1010、0011，然后把每组写成一个数字，这个数的低位是 3，高位已经大于 9 了，我们规定用字母 A～F 表示 10～15，则这个数可以写成$(A3)_{16}$，每一位数字的取值范围是 0～F，逢十六进一，称为十六进制（Hexadecimal）。所以，八进制、十六进制是程序员为了书写二进制数方便而发明的简便写法，好比草书和正楷的关系一样。

习题

1．二进制小数可以这样定义：

$$(0.A_1A_2A_3...)_2=A_1\times2^{-1}+A_2\times2^{-2}+A_3\times2^{-3}+...$$

这个定义同时也是从二进制小数到十进制小数的换算公式。从本节讲的十进制数转二进制数的推导过程出发类比一下，十进制小数换算成二进制小数应该怎么算？

2. 再类比一下，八进制数（或十六进制数）与十进制数之间如何相互换算？

13.3 整数的加减运算

我们已经了解了计算机中正整数如何表示，加法如何计算，那么负数如何表示，减法又如何计算呢？本节讨论这些问题。为了书写方便，本节举的例子都用 8 个 bit 表示一个数，实际计算机做整数加减运算的操作数可以是 8 位、16 位、32 位甚至 64 位的。

13.3.1 Sign and Magnitude 表示法

要用 8 个 bit 表示正数和负数，一种简单的想法是把最高位规定为符号位（Sign Bit），0 表示正 1 表示负，剩下的 7 位表示绝对值的大小，这称为 Sign and Magnitude 表示法。例如-1 表示成 10000001，+1 表示成 00000001。这样用 8 个 bit 表示整数的取值范围是$-(2^7-1)$～2^7-1，即-127～127。

采用这种表示法，计算机做加法运算需要处理以下逻辑：

1. 如果两数符号位相同，就把它们的低 7 位相加，符号位不变。如果低 7 位相加时在最高位产生进位，说明结果的绝对值大于 127，超出 7 位所能表示的数值范围，这称为溢出（Overflow）[②]，这时通常把计算机中的一个标志位置 1 表示当前运算产生了溢出。

2. 如果两数符号位不同，首先比较它们的低 7 位谁大，然后用大数减小数，结果的符号位和大数相同。

那么减法如何计算呢？由于我们规定了负数的表示，可以把减法转换成加法来计算，要计算 a-b，可以先把 b 变号然后和 a 相加，相当于计算 a+(-b)。但如果两个加数的符号位不同就要用大数的绝对值减小数的绝对值，这一步减法计算仍然是免不了的。我们知道加法要进位，减法要借位，计算过程是不同的，所以除了要有第 13.1 节提到的加法器电路之外，还要另外有一套减法器电路。

如果采用 Sign and Magnitude 表示法，计算机做加减运算需要处理很多逻辑：比较符号位、比较绝对值、加法改减法、减法改加法、小数减大数改成大数减小数……这是非常低效率的。还有一个缺点是 0 的表示不唯一，既可以表示成 10000000 也可以表示成 00000000，这进一步增加了逻辑的复杂性，所以我们迫切需要重新设计整数的表示方法使计算过程更简单。

13.3.2 1's Complement 表示法

本节介绍一种二进制补码表示法，为了便于理解，先从我们熟悉的十进制讲起。现在我们用三位十进制数字表示正数和负数，具体规定如表 13.5 所示。

② 有时候会进一步细分，把正整数溢出称为上溢（Overflow），负整数溢出称为下溢（Underflow），详见 strtol(3)。

表 13.5　9's Complement 表示法

数值	补码表示	数值	补码表示
-499	500	0	0
-498	501	1	1
...	...	...	...
-1	998	498	498
0	999	499	499

下面看一个十进制计算的例子：

167-59=167+(-59)=167+(999-59)-1000+1=167+940-1000+1=1107-1000+1=107+1=108

167-59 → 减法转换成加法 167+(-59) → 负数取 9 的补码表示 167+940 → 107 进 1 → 高位进的 1 加到低位上去，结果为 108。

首先-59 要用 999-59 表示，就是 940，这称为取 9 的补码（9's Complement）；然后把 167 和 940 相加，得到 107 进 1；再把高位进的 1 加到低位上去，得 108，本来应该加 1000，结果加了 1，少加了 999，正好把先前取 9 的补码多加的 999 抵消掉了。我们本来要做 167-59 的减法运算，结果变成做 999-59 的减法运算，后者显然要容易一些，因为没有借位。这种补码表示法的计算规则用一句话概括就是：**负数用 9 的补码表示，减法转换成加法，计算结果的最高位如果有进位则要加回到最低位上去**。要验证这条规则得考虑四种情况：

1．两个正数，相加得正

2．一正一负，相加得正

3．一正一负，相加得负

4．两个负数，相加得负

我们举的例子验证了第二种情况，另外三种情况请读者自己验证。注意，如果计算结果超出了三位十进制数字所能表示的范围（–499～499）则产生溢出，我们暂时不考虑溢出的问题，稍后会讲到如何判定溢出。

上述规则也适用于二进制数：**负数用 1 的补码（1's Complement）表示，减法转换成加法，计算结果的最高位如果有进位则要加回到最低位上去**。取 1 的补码更简单，连减法都不用做，因为 1-1=0，1-0=1，取 1 的补码就是把每个 bit 取反，所以 1 的补码也称为反码。比如：

00001000-00000100 → 00001000+(-00000100) → 00001000+11111011 → 00000011 进 1 → 高位进的 1 加到低位上去，结果为 00000100。

1's Complement 表示法相对于 Sign and Magnitude 表示法的优势是非常明显的：不需要把符号和绝对值分开考虑，正数和负数的加法都一样算，计算逻辑更简单，甚至连减法器电路都省了，只要有一套加法器电路，再有一套把每个 bit 取反的

电路，就可以做加法和减法运算。如果 8 个 bit 采用 1's Complement 表示法，负数的取值范围是从 10000000 到 11111111（-127～0），正数是从 00000000 到 01111111（0～127），仍然可以根据最高位判断一个数是正是负。美中不足的是 0 的表示仍然不唯一，既可以表示成 11111111 也可以表示成 00000000，为了解决这最后一个问题，我们引入 2's Complement 表示法。

13.3.3 2's Complement 表示法

2's Complement 表示法规定：正数不变，负数先取反码再加 1。如果 8 个 bit 采用 2's Complement 表示法，负数的取值范围是从 10000000 到 11111111（-128～-1），正数是从 00000000 到 01111111（0～127），也可以根据最高位判断一个数是正是负，并且 0 的表示是唯一的，目前绝大多数计算机都采用这种表示法。为什么称为“2 的补码”呢？因为对一位二进制数 b 取补码就是 1-b+1=10-b，相当于从 2 里面减去 b。类似地，要表示-4 需要对 00000100 取补码，11111111-00000100+1=100000000-00000100，相当于从 2^8 里面减去 4。2's Complement 表示法的计算规则有些不同：减法转换成加法，忽略计算结果最高位的进位，不必加回到最低位上去。请读者自己验证上一节提到的四种情况下这条规则都能算出正确结果。

8 个 bit 采用 2's Complement 表示法的取值范围是-128～127，如果计算结果超出这个范围就会产生溢出，如图 13.3 所示。

```
  10000010          -126
+ 11111000        +   -8
  100000000 进位
 ----------       ------
  01111010    =     122 ?
```

图 13.3 有符号数加法溢出

如何判断产生了溢出呢？我们还是分四种情况讨论：如果两个正数相加溢出，结果一定是负数；如果两个负数相加溢出，结果一定是正数；一正一负相加，无论结果是正是负都不可能溢出，如图 13.4 所示。

```
  0xxxxxxx        0xxxxxxx        0xxxxxxx        1xxxxxxx
+ 0xxxxxxx      + 1xxxxxxx      + 1xxxxxxx      + 1xxxxxxx
  00xxxxxxx 进位  11xxxxxxx 进位  00xxxxxxx 进位  11xxxxxxx 进位
 ----------      ----------      ----------      ----------
  0xxxxxxx        0xxxxxxx        1xxxxxxx        1xxxxxxx

  两个正数        一正一负        一正一负        两个负数
  相加得正        相加得正        相加得负        相加得负


  0xxxxxxx        1xxxxxxx
+ 0xxxxxxx      + 1xxxxxxx
  01xxxxxxx 进位  10xxxxxxx 进位
 ----------      ----------
  1xxxxxxx        0xxxxxxx

  两个正数        两个负数
相加得负（溢出）  相加得正（溢出）
```

图 13.4 如何判定溢出

从图 13.4 可以得出结论：**在相加过程中最高位产生的进位和次高位产生的进位如**

果相同则没有溢出，如果不同则表示有溢出。逻辑电路的实现可以把这两个进位连接到一个异或门，把异或门的输出连接到溢出标志位。

13.3.4　有符号数和无符号数

前面几节我们用 8 个 bit 表示正数和负数，讲了三种表示法，每种表示法对应一种计算规则，这称为有符号数（Signed Number）；如果 8 个 bit 全部表示正数则取值范围是 0～255，这称为无符号数（Unsigned Number）。其实计算机做加法时并不区分操作数是有符号数还是无符号数，计算过程都一样，比如上面的例子也可以看作无符号数的加法，如图 13.5 所示。

```
  10000010          130
+ 11111000        + 248
  100000000 进位
  --------          ---
  01111010    =     122 +256
```

图 13.5　无符号数加法进位

如果把这两个操作数看作有符号数-126 和-8 相加，计算结果是错的，因为产生了溢出；但如果看作无符号数 130 和 248 相加，计算结果是 122 进 1，也就是 122+256，这个结果是对的。计算机的加法器在做完计算之后，根据最高位产生的进位设置**进位标志**，同时根据最高位和次高位产生的进位的异或设置**溢出标志**。至于这个加法到底是有符号数加法还是无符号数加法则取决于程序怎么理解了，如果程序把它理解成有符号数加法，下一步就要检查溢出标志，如果程序把它理解成无符号数加法，下一步就要检查进位标志。通常计算机在做算术运算之后还可能设置另外两个标志，如果计算结果的所有 bit 都是零则设置**零标志**，如果计算结果的最高位是 1 则设置**负数标志**，如果程序把计算结果理解成有符号数，也可以检查负数标志判断结果是正是负。

13.4　浮点数

浮点数在计算机中的表示是基于科学计数法（Scientific Notation）的，我们知道 32767 这个数用科学计数法可以写成 3.2767×10^4，3.2767 称为尾数（Mantissa，或者叫 Significand），4 称为指数（Exponent）。浮点数在计算机中的表示与此类似，只不过基数（Radix）是 2 而不是 10。下面我们用一个简单的模型来解释浮点数的基本概念。我们的模型由三部分组成：符号位、指数部分（表示 2 的多少次方）和尾数部分（小数点前面是 0，尾数部分只表示小数点后的数字，如图 13.6 所示）。

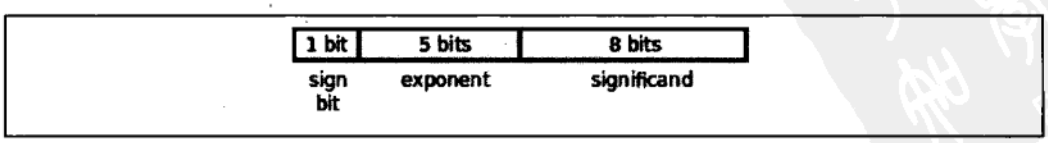


图 13.6　一种浮点数格式

如果要表示 17 这个数，我们知道 $17=17.0\times10^0=0.17\times10^2$，类似地，$17=(10001)_2$

$\times 2^0=(0.10001)_2\times 2^5$，把尾数的有效数字全部移到小数点后，这样就可以表示为如图 13.7 所示的形式。

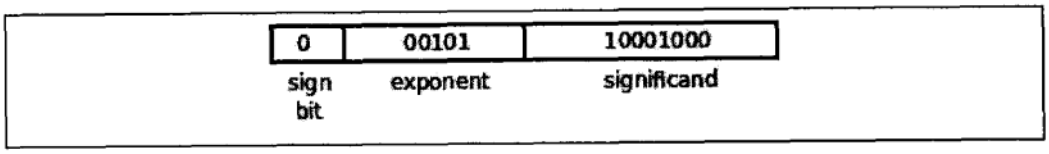


图 13.7　17 的浮点数表示

如果我们要表示 0.25 就遇到新的困难了，因为 $0.25=1\times 2^{-2}=(0.1)_2\times 2^{-1}$，而我们的模型中指数部分没有规定如何表示负数。我们可以在指数部分规定一个符号位，然而更广泛采用的办法是使用偏移的指数（Biased Exponent）。规定一个偏移值，比如 16，实际的指数要加上这个偏移值再填写到指数部分，这样比 16 大的就表示正指数，比 16 小的就表示负指数。要表示 0.25，指数部分应该填 16-1=15，如图 13.8 所示。

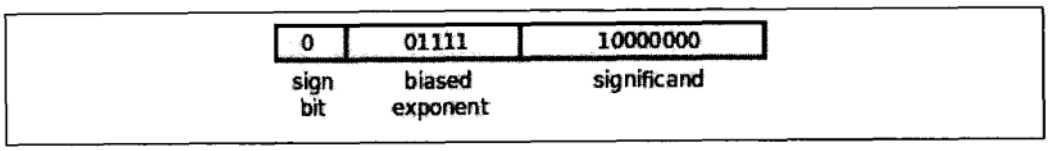


图 13.8　0.25 的偏移指数浮点数表示

现在还有一个问题需要解决：每个浮点数的表示都不唯一，例如 $17=(0.10001)_2\times 2^5=(0.010001)_2\times 2^6$，这样给计算机处理增加了复杂性。为了解决这个问题，我们规定尾数部分的最高位必须是 1，也就是说尾数必须以 0.1 开头，对指数做相应的调整，这称为正规化（Normalize）。由于尾数部分的最高位必须是 1，这个 1 就不必保存了，可以节省出一位来用于提高精度，我们说最高位的 1 是隐含的（Implied）。这样 17 就只有一种表示方法了，指数部分应该是 $16+5=21=(10101)_2$，尾数部分去掉最高位的 1 是 0001，如图 13.9 所示。

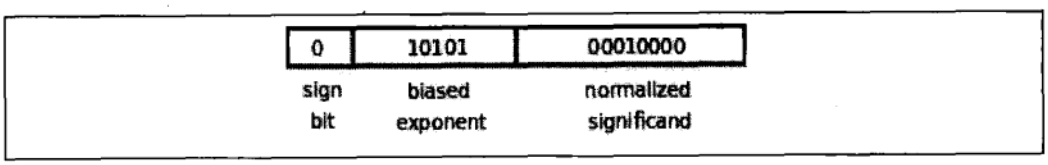


图 13.9　17 的正规化尾数浮点数表示

两个浮点数相加，首先把小数点对齐然后相加，如图 13.10 所示。

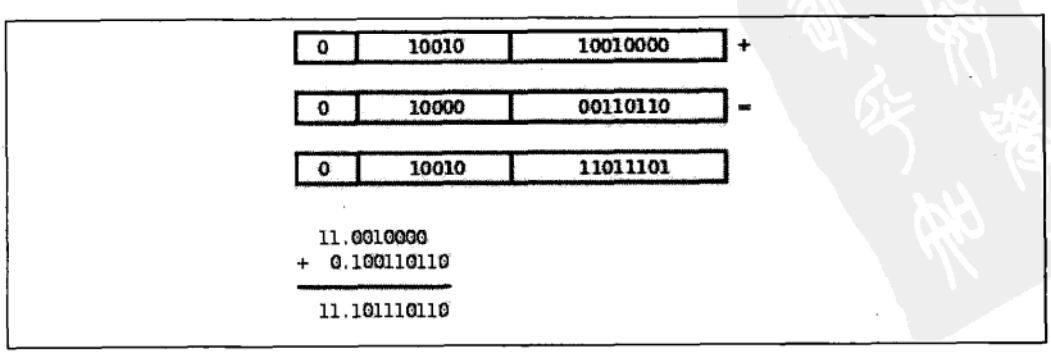


图 13.10　浮点数相加

由于浮点数表示的精度有限，计算结果末尾的 10 两位被舍去了。做浮点运算时要注意精度损失（Significance Loss）问题，有时计算顺序不同也会导致不同的结果，比如 11.0010000+0.00000001+0.00000001=11.0010000+0.00000001=11.0010000，后面加的两个很小的数全被舍去了，没有对计算结果产生任何影响，但如果调一下计算顺序它们就能影响到计算结果了，0.00000001+0.00000001+11.0010000=0.00000010+11.0010000=11.0010001。再比如 128.25=$(10000000.01)_2$，需要 10 个有效位，而我们的模型中尾数部分是 8 位，算上隐含的最高位 1，一共有 9 个有效位，那么 128.25 的浮点数表示只能舍去末尾的 1，表示成$(10000000.0)_2$，其实跟 128 相等。在第 4.2 节讲过浮点数不能做精确比较，现在读者应该知道为什么不能精确比较了。

整数运算会产生溢出，浮点运算也会产生溢出，浮点运算的溢出也分上溢和下溢两种，但和整数运算的定义不同。假设整数采用 8 位 2's Complement 表示法，取值范围是-128～127，如果计算结果是-130 则称为下溢，计算结果是 130 则称为上溢。假设按本节介绍的浮点数表示法，取值范围是$-(0.111111111)_2\times2^{15}$～$(0.111111111)_2\times2^{15}$，如果计算结果超出这个范围则称为上溢；如果计算结果未超出这个范围但绝对值太小了，在$-(0.1)_2\times2^{-16}$～$(0.1)_2\times2^{-16}$之间，那么也同样无法表示，称为下溢。

浮点数是一个相当复杂的话题，不同平台的浮点数表示和浮点运算也有较大差异，本节只是通过这个简单的模型介绍一些基本概念而不深入讨论，理解了这些基本概念有助于你理解浮点数标准，目前业界广泛采用的浮点数标准是由 IEEE（Institute of Electrical and Electronics Engineers）制定的 IEEE 754。

最后讨论一个细节问题。我们知道，定义全局变量时如果没有 Initializer 就用 0 初始化，定义数组时如果 Initializer 中提供的元素不够那么剩下的元素也用 0 初始化。例如：

```
int i;
double d;
double a[10] = { 1.0 };
```

“用 0 初始化”的意思是变量 i、变量 d 和数组元素 a[1]～a[9]的所有字节都用 0 填充，或者说所有 bit 都是 0。无论是用 Sign and Magnitude 表示法、1's Complement 表示法还是 2's Complement 表示法，一个整数的所有 bit 都是 0 表示 0 值，但一个浮点数的所有 bit 是 0 一定表示 0 值吗？严格来说不一定，某种平台可能会规定一个浮点数的所有 bit 是 0 并不表示 0 值，但参考文献[6]第 6.7.8 节提到：As far as the committee knows, all machines treat all bits zero as a representation of floating-point zero. But, all bits zero might not be the canonical representation of zero. 因此在绝大多数平台上，一个浮点数的所有 bit 是 0 就表示 0 值。

第 14 章

数据类型详解

14.1 整型

我们知道，在 C 语言中 char 型占一个字节的存储空间，一个字节通常是 8 个 bit。如果这 8 个 bit 按无符号整数来解释，取值范围是 0～255，如果按有符号整数来解释，采用 2's Complement 表示法，取值范围是-128～127。C 语言规定了 signed 和 unsigned 两个关键字，unsigned char 型表示无符号数，signed char 型表示有符号数。

那么以前我们常用的不带 signed 或 unsigned 关键字的 char 型是无符号数还是有符号数呢？C 标准规定这是 Implementation Defined，编译器可以定义 char 型是无符号的，也可以定义 char 型是有符号的，在该编译器所对应的体系结构上哪种实现效率高就可以采用哪种实现，x86 平台的 gcc 定义 char 型是有符号的。这也是 C 标准的 Rationale 之一：**优先考虑效率，而可移植性尚在其次。**这就要求程序员非常清楚这些规则，如果你要写可移植的代码，就必须清楚哪些写法是不可移植的，应该避免使用。另一方面，写不可移植的代码有时候也是必要的，比如 Linux 内核代码使用了很多只有 gcc 支持的语法特性以得到最佳的执行效率，在写这些代码的时候就没打算用别的编译器编译，也就没考虑可移植性的问题。如果要写不可移植的代码，你也必须清楚代码中的哪些部分是不可移植的，以及为什么要这样写，如果不是为了效率，一般来说就没有理由故意写不可移植的代码。从现在开始，我们会接触到很多 Implementation Defined 的特性，C 语言与平台和编译器是密不可分的，离开了具体的平台和编译器讨论 C 语言，就只能讨论到本书第一部分的程度了。注意，ASCII 码的取值范围是 0～127，所以不管 char 型是有符号的还是无符号的，存一个 ASCII 码都没有问题，一般来说，如果用 char 型存 ASCII 码字符，就不必明确写是 signed 还是 unsigned，如果用 char 型表示 8 位的整数，为了可移植性就必须写明是 signed 还是 unsigned。

> ✍ Implementation-defined、Unspecified 和 Undefined
>
> 在 C 标准中没有做明确规定的地方会用 Implementation-defined、Unspecified 或 Undefined 来表述，在本书中有时把这三种情况统称为“未明确定义”的。这三种情况到底有什么不同呢？
>
> 我们刚才看到一种 Implementation-defined 的情况，C 标准没有明确规定 char 是有符号的还是无符号的，但是要求编译器必须对此做出明确规定，

并写在编译器的文档中。

有些代码属于 Unspecified 的情况，通常有几种可选的处理方式，C 标准没有规定必须按哪种方式处理，编译器可以任选一种处理方式，编译器选择不同的处理方式可能会导致程序的运行结果不同，但 C 标准承认这几种不同的结果都算对。比如求表达式 i++ +i++的值，假设 i 的初值是 2，则可以按以下任意一种方式处理：

- 根据 i 的初值是 2，得出两个 i++的值都是 2，整个表达式的值是 2+2=4；两个 i++都使变量 i 从 2 变成 3，最后 i 的值就是 3。
- 根据 i 的初值是 2，得出第一个 i++的值是 2，然后第一个 i++使 i 从 2 变成 3，这时再得出第二个 i++的值是 3，整个表达式的值是 2+3=5，最后第二个 i++使 i 从 3 变成 4。
- 根据 i 的初值是 2，得出第二个 i++的值是 2，然后第二个 i++使 i 从 2 变成 3，这时再得出第一个 i++的值是 3，整个表达式的值是 3+2=5，最后第一个 i++使 i 从 3 变成 4。

可选的处理方式有三种，得到的结果有两种，C 标准规定不管按哪种方式处理得到的结果都算对，之所以这样规定是为了编译器在生成指令时可以根据各自平台的特点和寄存器的使用情况做一些优化，在第 15.3 节我们还会详细讨论表达式求值顺序的问题。

对于 Unspecified 的代码的处理方式不必写在编译器的文档中，同样的代码即使用同一个编译器的不同版本编译也可能得到不同的结果，因为编译器没有在文档中明确写它会怎么处理，那么不同版本的编译器就可以选择不同的处理方式。很显然，我们应该避免写 Unspecified 的代码。

有些代码属于 Undefined 的情况，C 标准没规定该怎么处理，也没规定这种代码的运行结果是对的，编译器很可能也没规定，甚至有很多 Undefined 的代码编译器是检查不出来的，有些会导致运行时错误，比如数组访问越界就是 Undefined 的。

初学者看到这些规则通常会很不舒服，觉得这不是在学编程而是在啃法律条文，结果越学越泄气。是的，C 语言并不像一个数学定理那么完美，现实世界里的东西总是不够完美。但还好啦，C 程序员已经很幸福了，只要严格遵照 C 标准来写代码，不要去触碰那些阴暗角落，写出来的代码就有很好的可移植性。想想那些可怜的 JavaScript 程序员吧，他们甚至连一个可以遵照的标准都没有，一个浏览器一个样，甚至同一个浏览器的不同版本也差别很大，程序员不得不为每一种浏览器的每一个版本分别写不同的代码。

除了 char 型之外，整型还包括 short int（或者简写为 short）、int、long int（或者简写为 long）、long long int（或者简写为 long long）等几种[①]。这些类型都可以加上 signed 或 unsigned 关键字表示有符号或无符号数。用这两个关键字修饰 int

① 我们在第 18.4 节还要介绍一种特殊的整型——Bit-field。

型时，signed int 和 unsigned int 可以简写为 signed 和 unsigned。

其实，对于有符号数在计算机中的表示是 Sign and Magnitude、1's Complement 还是 2's Complement，C 标准也没有明确规定，也是 Implementation Defined。大多数体系结构都采用 2's Complement 表示法，x86 平台也是如此，从现在开始我们只讨论 2's Complement 表示法的情况。还有一点要注意，除了 char 型以外的这些类型如果不明确写 signed 或 unsigned 关键字都表示 signed，这一点是 C 标准明确规定的，不是 Implementation Defined。

除了 char 型在 C 标准中明确规定占一个字节之外，其他整型占几个字节都是 Implementation Defined。通常的编译器实现遵守 ILP32 或 LP64 规范，如表 14.1 所示。

表 14.1 ILP32 和 LP64

类型	ILP32（位数）	LP64（位数）
char	8	8
short	16	16
int	32	32
long	32	64
long long	64	64
指针	32	64

ILP32 这个缩写的意思是 int（I）、long（L）和指针（P）类型都占 32 位，通常 32 位计算机的 C 编译器采用这种规范，x86 平台的 gcc 也是如此。LP64 是指 long（L）和指针（P）占 64 位，通常 64 位计算机的 C 编译器采用这种规范。指针类型的长度总是和计算机的位数一致，至于什么是计算机的位数，指针又是一种什么样的类型，我们到第 16 章和第 22 章再分别详细解释。从现在开始本书做以下约定：**在以后的陈述中，缺省平台是 x86/Linux/gcc，遵循 ILP32，并且 char 是有符号的，我不会每次都加以说明，但说到其他平台时我会明确指出是什么平台。**

在第 2.2 节讲过 C 语言的常量有整数常量、字符常量、枚举常量和浮点数常量四种，其实字符常量和枚举常量的类型都是 int 型，因此前三种常量的类型都属于整型。整数常量有很多种，不全是 int 型的，下面我们详细讨论整数常量。

以前我们只用到十进制的整数常量，其实在 C 语言中也可以用八进制和十六进制的整数常量[②]。八进制整数常量以 0 开头，后面的数字只能是 0～7，例如 022，因此十进制的整数常量就不能以 0 开头了，否则无法和八进制区分。十六进制整数常量以 0x 或 0X 开头，后面的数字可以是 0～9、a～f 和 A～F。在第 2.6 节讲

② 有些编译器（比如 gcc）也支持二进制的整数常量，以 0b 或 0B 开头，比如 0b0001111，但二进制的整数常量从未进入 C 标准，只是某些编译器的扩展，所以不建议使用，由于二进制和八进制、十六进制的对应关系非常明显，用八进制或十六进制常量完全可以代替使用二进制常量。

过一种转义序列，以\或\x 加八进制或十六进制数字表示，这种表示方式相当于把八进制和十六进制整数常量开头的 0 替换成\了。

整数常量还可以在末尾加 u 或 U 表示“unsigned”，加 l 或 L 表示“long”，加 ll 或 LL 表示“long long”，例如 0x1234U，98765ULL 等。但事实上 u、l、ll 这几种后缀和上面讲的 unsigned、long、long long 关键字并不是一一对应的。这个对应关系比较复杂，准确的描述如表 14.2 所示（出自参考文献[8]的 6.4.4.1 节条款 5）。

表 14.2　整数常量的类型

后缀	十进制常量	八进制或十六进制常量
无	int long int long long int	int unsigned int long int unsigned long int long long int unsigned long long int
u 或 U	unsigned int unsigned long int unsigned long long int	unsigned int unsigned long int unsigned long long int
l 或 L	long int long long int	long int unsigned long int long long int unsigned long long int
既有 u 或 U，又有 l 或 L	unsigned long int unsigned long long int	unsigned long int unsigned long long int
ll 或 LL	long long int	long long int unsigned long long int
既有 u 或 U，又有 ll 或 LL	unsigned long long int	unsigned long long int

给定一个整数常量，比如 1234U，那么它应该属于“u 或 U”这一行的“十进制常量”这一列，这个表格单元中列了三种类型 unsigned int、unsigned long int、unsigned long long int，从上到下找出第一个足够长的类型可以表示 1234 这个数，那么它就是这个整数常量的类型，如果 int 是 32 位的那么 unsigned int 就可以表示。

再比如 0xffff0000，应该属于第一行“无”的第二列“八进制或十六进制常量”，这一列有六种类型 int、unsigned int、long int、unsigned long int、long long int、unsigned long long int，第一个类型 int 表示不了 0xffff0000 这么大的数，我们写这个十六进制常量是要表示一个正数，而它的 MSB（第 31 位）是 1，如果按有符号 int 类型来解释就成了负数了，第二个类型 unsigned int 可以表示这个数，所以这个十六进制常量的类型应该算 unsigned int。所以请注意，0x7fffffff 和 0xffff0000 这两个常量虽然看起来差不多，但前者是 int 型，而后者是 unsigned int 型。

讲一个有意思的问题。我们知道 x86 平台上 int 的取值范围是-2147483648～2147483647，那么用 printf("%d\n", -2147483648);打印 int 类型的下界有没有问题呢？如果用 gcc main.c -std=c99 编译会有警告信息：warning: format '%d' expects type 'int', but argument 2 has type 'long long int'。这是因为，虽然-2147483648 这个数值能够用 int 型表示，但在 C 语言中却没法写出对应这个数值的 int 型常量，C 编译器会把它当成一个整数常量 2147483648 和一个负号运算符组成的表达式，而整数常量 2147483648 已经超过了 int 型的取值范围，在 x86 平台上 int 和 long 的取值范围相同，所以这个常量也超过了 long 型的取值范围，根据表 14.2 第一行“无”的第一列十进制常量，这个整数常量应该算 long long 型的，前面再加个负号组成的表达式仍然是 long long 型，而 printf 的%d 转换说明要求后面的参数是 int 型，所以编译器报警告。之所以编译命令要加-std=c99 选项是因为 C99 以前对于整数常量的类型规定和表 14.2 有一些出入，即使不加这个选项也会报警告，但警告信息不准确，读者可以试试。如果改成 printf("%d\n", -2147483647-1);编译器就不会报警告了，-号运算符的两个操作数-2147483647 和 1 都是 int 型，计算结果也应该是 int 型，并且它的值也没有超出 int 型的取值范围；或者改成 printf("%lld\n", -2147483648);也可以，转换说明%lld 告诉 printf 后面的参数是 long long 型，有些转换说明格式目前还没讲到，详见第 24.2.9 节。

怎么样，整数常量没有你原来想的那么简单吧。再看一个不简单的问题。long long i = 1234567890 * 1234567890;编译时会有警告信息：warning: integer overflow in expression。1234567890 是 int 型，两个 int 型相乘的表达式仍然是 int 型，而乘积已经超过 int 型的取值范围了，因此提示计算结果溢出。如果改成 long long i = 1234567890LL * 1234567890;，其中一个常量是 long long 型，另一个常量也会先转换成 long long 型再做乘法运算，两数相乘的表达式也是 long long 型，编译器就不会报警告了。有关类型转换的规则将在第 14.3 节详细介绍。

14.2 浮点型

C 标准规定的浮点型有 float、double、long double，和整型一样，既没有规定每种类型占多少字节，也没有规定采用哪种表示形式。浮点数的实现在各种平台上差异很大，有的处理器有浮点运算单元（Floating Point Unit，FPU），称为硬浮点（Hard-float）实现；有的处理器没有浮点运算单元，只能做整数运算，需要用整数运算来模拟浮点运算，称为软浮点（Soft-float）实现。大部分平台的浮点数实现遵循 IEEE 754，float 型通常是 32 位，double 型通常是 64 位。

long double 型通常是比 double 型精度更高的类型，但各平台的实现有较大差异。在 x86 平台上，大多数编译器实现的 long double 型是 80 位，因为 x86 的浮点运算单元具有 80 位精度，gcc 实现的 long double 型是 12 字节（96 位），这是为了对齐到 4 字节边界（在第 18.4 节详细讨论对齐的问题），也有些编译器实现的 long double 型和 double 型精度相同，没有充分利用 x86 浮点运算单元的精度。其他体系结构的浮点运算单元的精度不同，编译器实现也会不同，例如 PowerPC 上的 long double 型通常是 128 位。

以前我们只用到最简单的浮点数常量，例如 3.14，现在看看浮点数常量还有哪些写法。由于浮点数在计算机中的表示是基于科学计数法的，所以浮点数常量也可以写成科学计数法的形式，尾数和指数之间用 e 或 E 隔开，例如 314e-2 表示 314×10^{-2}，注意这种表示形式基数是 10[③]，如果尾数的小数点左边或右边没有数字则表示这一部分为零，例如 3.e-1，.987 等。浮点数也可以加一个后缀，例如 3.14f、.01L，浮点数的后缀和类型之间的对应关系比较简单，没有后缀的浮点数常量是 double 型的，有后缀 f 或 F 的浮点数常量是 float 型的，有后缀 l 或 L 的浮点数常量是 long double 型的。

14.3　类型转换

如果有人问 C 语法规则中最复杂的是哪一部分，我一定会说是类型转换。从上面两节可以看出，有符号、无符号整数和浮点数加起来有那么多种类型，每两种类型之间都要定义一个转换规则，转换规则的数量自然很庞大，更何况由于各种体系结构对于整数和浮点数的实现很不相同，很多类型转换的情况都是 C 标准未做明确规定的阴暗角落。虽然我们写代码时不会故意去触碰这些阴暗角落，但有时候会不小心犯错，所以了解一些未明确规定的情况还是有必要的，可以在出错时更容易分析错误原因。本节分成几小节，首先介绍哪些情况下会发生类型转换，会把什么类型转成什么类型，然后介绍编译器如何处理这样的类型转换。

14.3.1　Integer Promotion

在一个表达式中，凡是可以使用 int 或 unsigned int 类型做右值的地方也都可以使用有符号或无符号的 char 型、short 型和 Bit-field。如果原始类型的取值范围都能用 int 型表示，则其类型被提升为 int，如果原始类型的取值范围用 int 型表示不了，则提升为 unsigned int 型，这称为 Integer Promotion。做 Integer Promotion 只影响上述几种类型的值，对其他类型无影响。C99 规定 Integer Promotion 适用于以下几种情况：

1．如果一个函数的形参类型未知，例如使用了 Old Style C 风格的函数声明（详见第 3.2 节），或者函数的参数列表中有...，那么调用函数时要对相应的实参做 Integer Promotion，此外，相应的实参如果是 float 型的也要被提升为 double 型，这条规则称为 Default Argument Promotion。我们知道 printf 的参数列表中有...，除了第一个形参之外，其他形参的类型都是未知的，比如有这样的代码：

```
char ch = 'A';
printf("%c", ch);
```

ch 要被提升为 int 型之后再传给 printf。

③ C99 引入一种新的十六进制浮点数表示，基数是 2，本书不做详细介绍。

2．算术运算中的类型转换。有符号或无符号的 char 型、short 型和 Bit-field 在做算术运算之前首先要做 Integer Promotion，然后才能参与计算。例如：

```
unsigned char c1 = 255, c2 = 2;
int n = c1 + c2;
```

计算表达式 c1 + c2 的过程其实是先把 c1 和 c2 提升为 int 型然后再相加(unsigned char 的取值范围是 0～255，完全可以用 int 表示，所以提升为 int 就可以了，不需要提升为 unsigned int)，整个表达式的值也是 int 型，最后的结果是 257。假如没有这个提升的过程，c1 + c2 就溢出了，溢出会得到什么结果是 Undefined，在大多数平台上会把进位截掉，得到的结果应该是 1。

除了+号之外还有哪些运算符在计算之前需要做 Integer Promotion 呢？我们在下一小节先介绍 Usual Arithmetic Conversion 规则，然后再解答这个问题。

14.3.2 Usual Arithmetic Conversion

两个算术类型的操作数做算术运算，比如 a + b，如果两边操作数的类型不同，编译器会自动做类型转换，使两边类型相同之后才做运算，这称为 Usual Arithmetic Conversion。转换规则如下：

1．如果有一边的类型是 long double，则把另一边也转成 long double。

2．否则，如果有一边的类型是 double，则把另一边也转成 double。

3．否则，如果有一边的类型是 float，则把另一边也转成 float。

4．否则，两边应该都是整型，首先按上一小节讲过的规则对 a 和 b 做 Integer Promotion，然后如果类型仍不相同，则需要继续转换。我们规定 char、short、int、long、long long 的转换级别（Integer Conversion Rank）一个比一个高，同一类型的有符号和无符号数具有相同的 Rank。转换规则如下：

a．如果两边都是有符号数，或者都是无符号数，那么较低 Rank 的类型转换成较高 Rank 的类型。例如 unsigned int 和 unsigned long 做算术运算时都转成 unsigned long。

b．否则，如果一边是无符号数另一边是有符号数，无符号数的 Rank 不低于有符号数的 Rank，则把有符号数转成另一边的无符号类型。例如 unsigned long 和 int 做算术运算时都转成 unsigned long，unsigned long 和 long 做算术运算时也都转成 unsigned long。

c．剩下的情况是：一边有符号另一边无符号，并且无符号数的 Rank 低于有符号数的 Rank。这时又分为两种情况，如果这个有符号数类型能够覆盖这个无符号数类型的取值范围，则把无符号数转成另一边的有符号类型。例如遵循 LP64 的平台上 unsigned int 和 long 在做算术运算时都转成 long。

d. 否则，也就是这个有符号数类型不足以覆盖这个无符号数类型的取值范围，则把两边都转成有符号数的 Rank 对应的无符号类型。例如在遵循 ILP32 的平台上 unsigned int 和 long 在做算术运算时都转成 unsigned long。

可见有符号和无符号整数的转换规则是十分复杂的，虽然这是有明确规定的，不属于阴暗角落，但为了程序的可读性不应该依赖这些规则来写代码。我讲这些规则，不是为了让你用，而是为了让你了解有符号数和无符号数混用会非常麻烦，从而避免触及这些规则，并且在程序出错时记得往这上面找原因。所以这些规则不需要牢记，但要知道有这么回事，以便在用到的时候能找到我书上的这一段。

到目前为止我们学过的+ - * / % ><>= <= == !=运算符都需要做 Usual Arithmetic Conversion，因为都要求两边操作数的类型一致，在下一章会介绍几种新的运算符也需要做 Usual Arithmetic Conversion。单目运算符+ - ~只有一个操作数，移位运算符<<>>两边的操作数类型不要求一致，这些运算不需要做 Usual Arithmetic Conversion，但也需要做 Integer Promotion，运算符~ <<>>将在下一章介绍。

14.3.3 由赋值产生的类型转换

如果赋值或初始化时等号两边的类型不相同，则编译器会把等号右边的类型转换成等号左边的类型再做赋值。例如 int c = 3.14;，编译器会把右边的 double 型转成 int 型再赋给变量 c。

我们知道，函数调用传参的过程相当于定义形参并且用实参对其做初始化，函数返回的过程相当于定义一个临时变量并且用 return 的表达式对其做初始化，所以由赋值产生的类型转换也适用于这两种情况。例如一个函数的原型是 int foo(int, int);，则调用 foo(3.1, 4.2)时会自动把两个 double 型的实参转成 int 型赋给形参，如果这个函数定义中有返回语句 return 1.2;，则返回值 1.2 会自动转成 int 型再返回。

在函数调用和返回过程中发生的类型转换往往容易被忽视，因为函数原型和函数调用并没有写在一起。例如 char c = getchar();，看到这一句往往会想当然地认为 getchar 的返回值是 char 型，而事实上 getchar 的返回值是 int 型，这样赋值会引起类型转换，可能产生 Bug，我们在第 24.2.5 节详细讨论这个问题。

14.3.4 强制类型转换

以上三种情况通称为隐式类型转换（Implicit Conversion，或者叫 Coercion），编译器根据它自己的一套规则将一种类型自动转换成另一种类型。除此之外，程序员也可以通过类型转换运算符（Cast Operator）自己规定某个表达式要转换成何种类型，这称为显式类型转换（Explicit Conversion）或强制类型转换（Type Cast）。例如计算表达式(double)3 + i，首先将整数 3 强制转换成 double 型（值为 3.0），然后和整型变量 i 相加，这时适用 Usual Arithmetic Conversion 规则，首先把 i 也转成 double 型，然后两者相加，最后整个表达式也是 double 型的。这里的(double)

就是一个类型转换运算符，这种运算符由一个类型名套()括号组成，属于单目运算符，后面的 3 是这个运算符的操作数。注意操作数的类型必须是标量类型，转换之后的类型必须是标量类型或者 void 型。

14.3.5 编译器如何处理类型转换

以上几小节介绍了哪些情况下会发生类型转换，并且明确了每种情况下会把什么类型转成什么类型，本节介绍编译器如何处理任意两种类型之间的转换。现在要把一个 M 位的类型（值为 X）转换成一个 N 位的类型，所有可能的情况如表 14.3 所示。

表 14.3 如何做类型转换

待转换的类型	M > N 的情况	M == N 的情况	M < N 的情况
signed integer to signed integer	如果 X 在目标类型的取值范围内则值不变，否则 Implementation-defined	值不变	值不变
unsigned integer to signed integer	如果 X 在目标类型的取值范围内则值不变，否则 Implementation-defined	如果 X 在目标类型的取值范围内则值不变，否则 Implementation-defined	值不变
signed integer to unsigned integer	$X \% 2^N$	$X \% 2^N$	$X \% 2^N$
unsigned integer to unsigned integer	$X \% 2^N$	值不变	值不变
floating-point to signed or unsigned integer	Truncate towards Zero，如果 X 的整数部分超出目标类型的取值范围则 Undefined		
signed or unsigned integer to floating-point	如果 X 在目标类型的取值范围内则值不变，但有可能损失精度，如果 X 超出目标类型的取值范围则 Undefined		
floating-point to floating-point	如果 X 在目标类型的取值范围内则值不变，但有可能损失精度，如果 X 超出目标类型的取值范围则 Undefined	值不变	值不变

注意表 14.3 中的“$X \% 2^N$”，我想表达的意思是“把 X 加上或者减去 2^N 的整数倍，使结果落入[0, 2^N-1]的范围内”，当 X 是负数时运算结果也得是正数，即运算结果和除数同号而不是和被除数同号，这不同于 C 语言%运算的定义。写程序时不要故意用表 14.3 中的规则，尤其不要触碰 Implementation-defined 和 Undefined 的情况，但程序出错时可以借助表 14.3 分析错误原因。

下面举几个例子说明表 14.3 的用法。比如把 double 型转换成 short 型，对应表中的“floating-point to signed or unsigned integer”，如果原值在(-32769.0, 32768.0)

之间则截掉小数部分得到转换结果，否则产生溢出，结果是 Undefined，例如对于 short s = 32768.4;这样的代码 gcc 会报警告。

比如把 int 型转换成 unsigned short 型，对应表中的“signed integer to unsigned integer”，如果原值是正的，则把它除以 2^{16} 取模，其实就是取它的低 16 位，如果原值是负的，则加上 2^{16} 的整数倍，使结果落在[0, 65535]之间。

比如把 int 类型转换成 short 类型，对应表中的“signed integer to signed integer”，如果原值在[-32768, 32767]之间则值不变，否则产生溢出，结果是 Implementation-defined，例如对于 short s = -32769;这样的代码 gcc 会报警告。

最后一个例子，把 short 型转换成 int 型，对应表中的“signed integer to signed integer”，转换之后应该值不变。那怎么维持值不变呢？是不是在高位补 16 个 0 就行了呢？如果原值是-1，十六进制表示就是 ffff，要转成 int 型的-1 需要变成 ffffffff，因此需要在高位补 16 个 1 而不是 16 个 0。换句话说，要维持值不变，在高位补 1 还是补 0 取决于原来的符号位，这称为符号扩展（Sign Extension）。

第 15 章

运算符详解

本章介绍很多前面没有讲过的运算符，重点是位运算，然后引出一个重要的概念 Sequence Point，在最后一节总结 C 语言各种运算符的优先级和结合性。

15.1 位运算

整数在计算机中用二进制的位来表示，C 语言提供一些运算符可以直接操作整数中的位，称为位运算，这些运算符的操作数都必须是整型的。在以后的学习中你会发现，有些信息利用整数中的某几个位来存储，要访问这些位，仅仅有对整数的操作是不够的，必须借助位运算，例如第 A.2 节介绍的 UTF-8 编码就是如此，学完本节之后你应该能自己写出 UTF-8 的编码和解码程序。本节首先介绍各种位运算符，然后介绍与位运算有关的编程技巧。

15.1.1 按位与、或、异或、取反运算

在第 4.3 节讲过逻辑与、或、非运算，并列出了真值表，对于整数中的位也可以做与、或、非运算，&是按位与（Bitwise AND）运算符、| 是按位或（Bitwise OR）运算符，~（Tilde）是按位取反（Bitwise NOT）运算符，此外还有 ^（Caret）是按位异或（Bitwise XOR）运算符，我们在第 13.1 节讲过异或运算。下面用二进制的形式举几个例子，如图 15.1 所示。

```
  00000011     00000011     00000011
& 00000101   | 00000101   ^ 00000101   ~ 11111100
----------   ----------   ----------   ----------
  00000001     00000111     00000110     00000011
```

图 15.1 位运算

注意，&、|、^运算符都是要做 Usual Arithmetic Conversion 的（其中有一步是 Integer Promotion），~运算符也要做 Integer Promotion，所以在 C 语言中其实并不存在 8 位整数的位运算，操作数在做位运算之前都至少被提升为 int 型了，上面用 8 位整数举例只是为了书写方便。比如：

```
unsigned char c = 0xfc;
unsigned int i = ~c;
```

计算过程是这样的：常量 0xfc 是 int 型的，赋给 c 要转成 unsigned char，值不变；c 的十六进制表示是 fc，计算～c 时先提升为整型（000000fc）然后取反，最后结果是 ffffff03。注意，如果把～c 看成是 8 位整数的取反，最后结果就得 3 了，这就错了。为了避免出错，一是尽量避免不同类型之间的赋值，二是每一步计算都要按上一章讲的类型转换规则仔细检查。

15.1.2　移位运算

移位运算符（Bitwise Shift）包括左移<<和右移>>。左移将一个整数的各二进制位全部左移若干位，例如 0xfffffff3<<2 得到 0x3fffffcc，如图 15.2 所示。

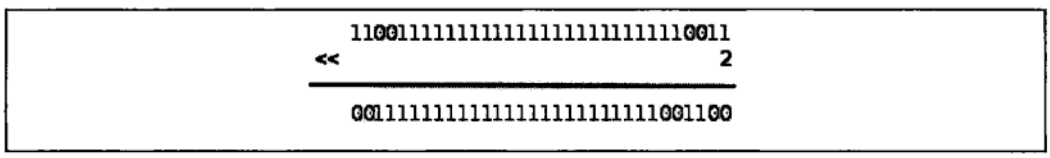


图 15.2　左移运算

最高两位的 11 被移出去了，最低两位又补了两个 0，其他位依次左移两位。但要注意，移动的位数必须小于左操作数的总位数，比如上面的例子，左边是 unsigned int 型，如果左移的位数大于等于 32 位，则结果是 Undefined。移位运算符不同于 + - * / ==等运算符，两边操作数的类型不要求一致，但两边操作数都要做 Integer Promotion，整个表达式的类型和左操作数提升后的类型相同。

复习一下第 13.2 节讲过的知识可以得出结论，**在一定的取值范围内，将一个整数左移 1 位相当于乘以 2**。比如二进制 11（十进制 3）左移一位变成 110，就是 6，再左移一位变成 1100，就是 12。读者可以自己验证这条规律对有符号数和无符号数都成立，对负数也成立。当然，如果左移改变了最高位（符号位），那么结果肯定不是乘以 2 了，所以我加了个前提“在一定的取值范围内”。由于计算机做移位比做乘法快得多，编译器可以利用这一点做优化，比如看到源代码中有 i * 8，可以编译成移位指令而不是乘法指令。

当操作数是无符号数时，右移运算的规则和左移类似，例如 0xfffffff3>>2 得到 0x3ffffffc，如图 15.3 所示。

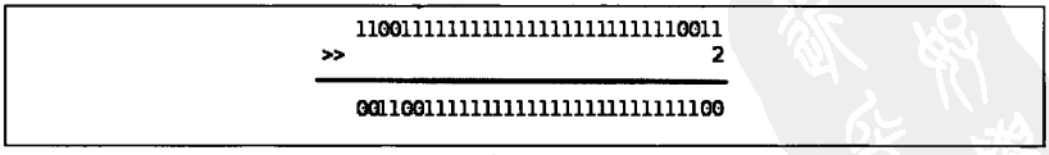


图 15.3　右移运算

最低两位的 11 被移出去了，最高两位又补了两个 0，其他位依次右移两位。和左移类似，移动的位数也必须小于左操作数的总位数，否则结果是 Undefined。在一定的取值范围内，将一个整数右移 1 位相当于除以 2，小数部分截掉。

当操作数是有符号数时，右移运算的规则比较复杂：

- 如果是正数，那么高位移入0。
- 如果是负数，那么高位移入1还是0不一定，这是Implementation-defined的。对于x86平台的gcc编译器，最高位移入1，也就是仍保持负数的符号位，这种处理方式对负数仍然保持了“右移1位相当于除以2”的性质。

综上所述，由于类型转换和移位等问题，用有符号数做位运算是很不方便的，所以，**建议只对无符号数做位运算，以减少出错的可能。**

习题

1．下面两行printf打印的结果有何不同？请读者比较分析一下。%x转换说明的含义详见第24.2.9节。

```
int i = 0xfffffff3;
printf("%x\n", 0xfffffff3>>2);
printf("%x\n", i>>2);
```

15.1.3 掩码

如果要对一个整数中的某些位进行操作，怎样表示这些位在整数中的位置呢？可以用掩码（Mask）来表示。比如掩码0x0000ff00表示对一个32位整数的8～15位进行操作，举例如下。

1．取出8～15位。

```
unsigned int a, b, mask = 0x0000ff00;
a = 0x12345678;
b = (a & mask) >> 8; /* 0x00000056 */
```

这样也可以达到同样的效果：

```
b = (a >> 8) & ~(~0U << 8);
```

2．将8～15位清0。

```
unsigned int a, b, mask = 0x0000ff00;
a = 0x12345678;
b = a & ~mask; /* 0x12340078 */
```

3．将8～15位置1。

```
unsigned int a, b, mask = 0x0000ff00;
a = 0x12345678;
b = a | mask; /* 0x1234ff78 */
```

习题

1.统计一个无符号整数的二进制表示中1的个数，函数原型是int countbit(unsigned

int x);。

2．用位操作实现无符号整数的乘法运算，函数原型是 unsigned int multiply (unsigned int x, unsigned int y);。例如：$(11011)_2 \times (10010)_2 = ((11011)_2 << 1) + ((11011)_2 << 4)$。

3．对一个 32 位无符号整数做循环右移，函数原型是 unsigned int rotate_right(unsigned int x, int n);。所谓循环右移就是把低位移出去的部分再补到高位上去，例如 rotate_right(0xdeadbeef, 8)的值应该是 0xefdeadbe。

15.1.4　异或运算的一些特性

1．一个数和自己做异或的结果是 0。如果需要一个常数 0，x86 平台的编译器可能会生成这样的指令：xorl %eax, %eax。不管 eax 寄存器里的值原来是多少，做异或运算都能得到 0，这条指令比同样效果的 movl $0, %eax 指令快，直接对寄存器做位运算比生成一个立即数再传送到寄存器要快一些，x86 指令将在第 17 章详细介绍。

2．从异或的真值表可以看出，不管是 0 还是 1，和 0 做异或保持原值不变，和 1 做异或得到原值的相反值。可以利用这个特性配合掩码实现某些位的翻转，例如：

```
unsigned int a, b, mask = 1U << 6;
a = 0x12345678;
b = a ^ mask; /* flip the 6th bit */
```

3．如果 $a_1 \wedge a_2 \wedge a_3 \wedge \ldots \wedge a_n$ 的结果是 1，则表示 a_1、a_2、a_3...a_n 之中 1 的个数为奇数个，否则为偶数个。这条性质可用于奇偶校验（Parity Check），比如在串口通信过程中，每个字节的数据都计算一个校验位，数据和校验位一起发送出去，这样接收方可以根据校验位粗略地判断接收到的数据是否有误。

4、$x \wedge x \wedge y = y$，因为 $x \wedge x = 0$，$0 \wedge y = y$。这个性质有什么用呢？我们来看这样一个问题：交换两个变量的值，不得借助额外的存储空间，所以就不能采用 temp = a; a = b; b = temp;的办法了。利用位运算可以这样做交换：

```
a = a ^ b;
b = b ^ a;
a = a ^ b;
```

分析一下这个过程。为了避免混淆，把 a 和 b 的初值分别记为 a_0 和 b_0。第一行，$a = a_0 \wedge b_0$；第二行，把 a 的新值代入，得到 $b = b_0 \wedge a_0 \wedge b_0$，等号右边的 b_0 相当于上面公式中的 x，a_0 相当于 y，所以结果为 a_0；第三行，把 a 和 b 的新值代入，得到 $a = a_0 \wedge b_0 \wedge a_0$，结果为 b_0。注意这个过程不能把同一个变量自己跟自己交换，而利用中间变量 temp 则可以交换。

习题

1．请在网上查找有关 RAID（Redundant Array of Independent Disks，独立磁盘冗

余阵列）的资料，理解其实现原理，其实就是利用了本节的性质3和性质4。

2．交换两个变量的值，不得借助额外的存储空间，除了本节讲的方法之外你还能想出什么方法？本节讲的方法不能把同一个变量自己跟自己交换，你的方法有没有什么局限性？

15.2 其他运算符

15.2.1 复合赋值运算符

复合赋值运算符（Compound Assignment Operator）包括*= /= %= += -= <<= >>= &= ^= |=，一边做运算一边赋值。例如a += 1相当于a = a + 1。但有一点细微的差别，前者对表达式a只求值一次，而后者求值两次，如果a是一个复杂的表达式，求值一次和求值两次的效率是不同的，例如a[i+j] += 1和a[i+j] = a[i+j] + 1。那么仅仅是效率上的差别吗？对于没有Side Effect的表达式，求值一次和求值两次的结果是一样的，但对于有Side Effect的表达式则不一定，例如a[foo()] += 1和a[foo()] = a[foo()] + 1，如果foo()函数调用有Side Effect，比如会打印一条消息，那么前者只打印一次，而后者打印两次。

在第6.3节讲自增、自减运算符时说++i相当于i = i + 1，其实更准确地说应该是等价于i += 1，表达式i只求值一次，而--i等价于i -= 1。

15.2.2 条件运算符

条件运算符（Conditional Operator）是C语言中唯一一个三目运算符（Ternary Operator），带三个操作数，它的形式是表达式1？表达式2：表达式3，这个运算符所组成的整个表达式的值等于表达式2或表达式3的值，取决于表达式1的值是否为真，可以把它想象成这样的函数：

```
if (表达式1)
        return 表达式2;
else
        return 表达式3;
```

表达式1相当于if语句的控制表达式，因此它的值必须是标量类型，而表达式2和表达式3相当于同一个函数在不同情况下的返回值，因此它们的类型要求一致，也要做Usual Arithmetic Conversion。

下面举个例子，定义一个函数求两个参数中较大的一个。

```
int max(int a, int b)
{
        return (a > b) ? a : b;
}
```

15.2.3　逗号运算符

逗号运算符（Comma Operator）也是一种双目运算符，它的形式是表达式 1，表达式 2，两个表达式不要求类型一致，左边的表达式 1 先求值，求完了直接把值丢掉，再求右边表达式 2 的值作为整个表达式的值。逗号运算符是左结合的，类似于+ - * /运算符，根据组合规则可以写出表达式 1，表达式 2，表达式 3，…，表达式 n 这种形式，表达式 1，表达式 2 可以看作一个子表达式，先求表达式 1 的值，然后求表达式 2 的值作为这个子表达式的值，然后这个值再和表达式 3 组成一个更大的表达式，求表达式 3 的值作为这个更大的表达式的值，依此类推，整个计算过程就是从左到右依次求值，最后一个表达式的值成为整个表达式的值。

注意，函数调用时各实参之间也是用逗号隔开，这种逗号是分隔符而不是逗号运算符。但可以这样使用逗号运算符：

```
f(a, (t=3, t+2), c)
```

传给函数 f 的参数有三个，其中第二个参数的值是表达式 t+2 的值。

15.2.4　sizeof 运算符与 typedef 类型声明

sizeof 是一个很特殊的运算符，它有两种形式：“sizeof 表达式”和“sizeof(类型名)”。这个运算符很特殊，“sizeof 表达式”中的子表达式并不求值，而只是根据类型转换规则求得子表达式的类型，然后把这种类型所占的字节数作为整个表达式的值。有些人喜欢写成“sizeof(表达式)”的形式也可以，这里的括号和 return(1); 的括号一样，不起任何作用。但另外一种形式“sizeof(类型名)”的括号则是必须写的，整个表达式的值也是这种类型所占的字节数。

比如用 sizeof 运算符求一个数组的长度：

```
int a[12];
printf("%u\n", sizeof a/sizeof a[0]);
```

注意 sizeof a 中的 a 做左值，表示整个数组，而不是做右值转换成指向首元素的指针。表达式 a 的类型在编译时就可以确定，所以 sizeof a 的值在编译时计算[①]，12×4=48，sizeof a[0]的值是 4，所以 sizeof a/sizeof a[0]是常量表达式，在编译时就被替换成 12 了。事实上，即使写成 sizeof i++，表达式 i++也不求值，这是 sizeof 运算符的特殊之处，求 sizeof i++就是求表达式 i++的类型所占的字节数，而 i++的类型在编译时就知道了，不需要到运行时求值之后才知道。

在第 8.4 节提到，字符串字面值和数组类型相似，"Hello"可以看作是由'H'、'e'、'l'、'l'、'o'加一个 Null 字符组成的数组，因此 sizeof("Hello")的值是 6。

sizeof 运算符的结果是 size_t 类型的，这个类型定义在 stddef.h 头文件中，不过你的代码中只要不出现 size_t 这个类型名就不用包含这个头文件，比如像上面的例

① VLA 是一个例外，对 VLA 取 sizeof 是在运行时计算的，因为数组的长度在运行时才知道。

子就不用包含这个头文件。C 标准规定 size_t 是一种无符号整型，编译器可以用 typedef 做一个类型声明：

```
typedef unsigned long size_t;
```

那么 size_t 就代表 unsigned long 型。不同平台的编译器可能会根据自己平台的具体情况定义 size_t 所代表的类型，比如有的平台定义为 unsigned long 型，有的平台定义为 unsigned long long 型，C 标准规定 size_t 这个名字就是为了隐藏这些细节，使代码具有可移植性。所以注意不要把 size_t 类型和它所代表的真实类型混用，例如：

```
unsigned long x;
size_t y;
x = y;
```

如果在一种 ILP32 平台上定义 size_t 代表 unsigned long long 型，这段代码把 y 赋给 x 时就把高位截掉了，结果可能是错的。注意上面的 printf 中使用了%u 转换说明，表示后面的参数是无符号整型，这是一个小细节，用%d 通常也可以，但在一些极限情况下可能会出问题，比如后面的参数值很大，结果打印出来是负数。

typedef 这个关键字用于给某种类型起个新名字，比如上面的 typedef 声明可以这么看：去掉 typedef 就成了一个变量声明 unsigned long size_t;，size_t 是一个变量名，类型是 unsigned long，那么加上 typedef 之后，size_t 就是一个类型名，就代表 unsigned long 类型。再举个例子：

```
typedef char array_t[10];
array_t a;
```

这相当于声明 char a[10];。类型名也遵循标识符的命名规则，并且通常加个_t 后缀表示 Type。C 标准库的头文件 stdint.h 中定义了很多这样的类型名，如表 15.1 所示，使用这些类型名写代码就可以屏蔽 ILP32 和 LP64 之间的差异了：

表 15.1 stdint.h 中定义的部分类型名

类型名	说明
int8_t int16_t int32_t int64_t	8 位、16 位、32 位、64 位的有符号整数
uint8_t uint16_t uint32_t uint64_t	8 位、16 位、32 位、64 位的无符号整数
intptr_t	一种有符号整数类型，指针类型可以转换成这种类型而不丢失信息
uintptr_t	一种无符号整数类型，指针类型可以转换成这种类型而不丢失信息

习题

1. 有这样一段代码：

```
#include <stdio.h>
```

```
int array[] = { 123, 43, 21, 171, 42, 99, 216 };
#define LEN (sizeof(array) / sizeof(array[0]))

int main(void)
{
        int i, sum = 0;

        for (i = 0; i < LEN; i++)
                sum += array[i];
        printf("%d\n", sum);

        return 0;
}
```

如果把 for 循环改写成这样：

```
for (i = -1; i < LEN - 1; i++)
        sum += array[i+1];
```

结果和原来是否相同？

15.3　Side Effect 与 Sequence Point

如果你只想规规矩矩地写代码，那么基本用不着看这一节。本节的内容基本上是钻牛角尖儿的，除了 Short-circuit 比较实用，其他写法都应该避免使用。但没办法，有时候不是你想钻牛角尖儿，而是有人逼你去钻牛角尖儿。比如有的公司招聘喜欢出这样的笔试题：

```
int a = 0;
a = a++;
```

答案应该是 Unspecified，我甚至怀疑有些出题人是否真的知道答案。下面我们来看看到底哪些情况是 Specified，哪些情况是 Unspecified。

我们知道，调用一个函数可能产生 Side Effect，使用某些运算符（++ -- = 复合赋值）也会产生 Side Effect，如果一个表达式中隐含着多个 Side Effect，究竟哪个先发生哪个后发生呢？C 标准规定代码中的某些点是 Sequence Point，当执行到一个 Sequence Point 时，在此之前的 Side Effect 必须全部作用完毕，在此之后的 Side Effect 必须一个都没发生，这是 Specified。至于两个 Sequence Point 之间的多个 Side Effect 哪个先发生哪个后发生则没有规定，编译器可以任意选择各 Side Effect 的作用顺序，这是 Unspecified。下面详细解释各种 Sequence Point。

1．调用一个函数时，在所有准备工作做完之后、函数调用开始之前是 Sequence Point。比如调用 foo(f(), g())时，函数调用运算符()的操作数 foo、参数 f()、参数 g()这三个表达式哪个先求值哪个后求值是 Unspecified，但是必须都求值完了才能做最后的函数调用，所以 f()和 g()的 Side Effect 按什么顺序发生不一定，但必定在这些 Side Effect 全部作用完之后才开始调用 foo(f(), g())。

2．条件运算符?:、逗号运算符、逻辑与&&、逻辑或||的第一个操作数求值之后是Sequence Point。我们刚讲过条件运算符和逗号运算符，条件运算符要根据表达式1的值是否为真决定下一步求表达式2还是表达式3的值，如果决定求表达式2的值，表达式3就不会被求值了，反之也一样；逗号运算符也是这样，表达式1求值结束才继续求表达式2的值。

逻辑与和逻辑或早在第4.3节就讲了，但在初学阶段我一直回避它们的操作数求值顺序问题。这两个运算符和条件运算符类似，先求左操作数的值，然后根据这个值是否为真，右操作数可能被求值，也可能不被求值。比如例8.5这个程序中的这几句：

```
    ret = scanf("%d", &man);
    if (ret != 1 || man < 0 || man > 2) {
        printf("Invalid input!\n");
        return 1;
    }
```

其实可以写得更简单（类似于参考文献[3]的简洁风格）：

```
    if (scanf("%d", &man) != 1 || man < 0 || man > 2) {
        printf("Invalid input!\n");
        return 1;
    }
```

其中控制表达式scanf("%d", &man) != 1 || man < 0 || man > 2的求值顺序是这样的：

1．||运算符是左结合的，即 man < 0 和左边的||运算符结合，所以先求子表达式scanf("%d", &man) != 1 || man < 0的值作为第二个||运算符的左操作数。

2．在这个子表达式中先求左操作数scanf("%d", &man) != 1的值。

如果scanf调用失败，则“返回值不等于1”成立（即左操作数的值为1），可以立刻得到子表达式的值为1，而其右操作数man < 0不会被求值，既然子表达式（即第二个||运算符的左操作数）的值是1，那么整个控制表达式的值就是1，其右操作数man > 2也不会被求值，接下来执行下一句printf("Invalid input!\n");。

如果scanf调用成功，则读入的数保存在变量man中，并且返回值等于1，那么说“返回值不等于1”就不成立了，子表达式的左操作数为0，就会去求右操作数man < 0的值从而得出子表达式的值，这时变量man的值正是scanf读上来的，我们判断它是否在[0, 2]之间，如果man < 0不成立，则子表达式的值为0，即整个控制表达式的左操作数为0，就会去求右操作数man > 2的值，从而得出整个控制表达式的值，如果man > 2也不成立，则整个控制表达式的值为0，控制流程将跳过{}中的语句块。

如果scanf调用成功，但保存在变量man中的值不在[0, 2]之间，则整个控制表达式的值为0，也会执行printf("Invalid input!\n");。

&&运算与||运算类似，a && b 的计算过程是：首先求表达式 a 的值，如果 a 的值是假（即 0）则整个表达式的值是 0，不会再去求 b 的值；如果 a 的值是真（非零），则下一步求 b 的值从而得出整个表达式的值。所以，a && b 相当于“if a then b”，而 a || b 相当于“if not a then b”。这种特性称为 Short-circuit，很多人喜欢利用 Short-circuit 特性简化代码。

3．在一个完整的声明末尾是 Sequence Point，所谓完整的声明是指这个声明不是另外一个声明的一部分。比如声明 int a[10], b[20];，在 a[10]末尾是 Sequence Point，在 b[20]末尾也是。

4．在一个完整的表达式末尾是 Sequence Point，所谓完整的表达式是指这个表达式不是另外一个表达式的一部分。所以如果有 f(); g();这样两条语句，f()和 g()是两个完整的表达式，f()的 Side Effect 必定在 g()之前发生。

5．在库函数即将返回时是 Sequence Point。这条规则似乎可以包含在上一条规则里面，因为函数返回时必然会结束掉一个完整的表达式。而事实上很多库函数是以宏定义的形式实现的（函数式宏定义在第 20.2.1 节介绍），并不是真正的函数，所以才需要有这条规则。

还有两种 Sequence Point 和某些 C 标准库函数的执行过程相关，此处从略，有兴趣的读者可查阅参考文献[8]的 Annex C。

现在分析一下本节开头的例子：int a = 0; a = a++;。a = a++这个表达式按照运算符优先级应该先算 a++，再把表达式 a++的值赋给 a。已知 a 的初值是 0，则表达式 a++的值是 0。现在有两个 Side Effect，一个是在计算表达式 a++之后应该把 a 改成 1，另一个是把表达式 a++的值 0 赋给等号左边的 a，哪个先发生不一定，只知道在整个表达式求值结束时这两个 Side Effect 一定都发生了，最后 a 的值可能是 0 也可能是 1，所以是 Unspecified。这行代码用不同平台的不同编译器来编译结果可能是不同的，甚至在同一平台上用同一编译器的不同版本来编译也可能不同。

由于两个 Sequence Point 之间的多个 Side Effect 可以按任意顺序发生，所以在写代码时要注意，**在两个 Sequence Point 之间，同一个变量的值最多只允许改变一次**。但是做到这一点还不足以保证代码的执行结果是确定的，一个变量被改变了一次，就有改变之前和改变之后两个不同的值，如果这个变量在一个表达式中出现多次，它应该代表哪个值呢？比如在 a[i++] = i;中变量 i 只改变了一次，但结果仍是 Unspecified，我们分析一下：设 i 的初值是 0，则表达式 i++的值是 0，有一个 Side Effect 是把 i 改成 1，但这个 Side Effect 什么时候发生不一定，如果在这个 Side Effect 发生之后才取等号右边 i 的值，则把 a[0]赋值为 1，如果在这个 Side Effect 发生之前就取等号右边 i 的值，则把 a[0]赋值为 0。再比如 i = i + 1;，它的执行结果是确定的，它会读取 i 的值，也会产生 Side Effect 改写 i 的值，但必须先读再改写，所以读取到的 i 值一定是初值而不是改写之后的值。C99 的 6.5 节条款 2 规定：“Between the previous and next sequence point an object shall have its stored value modified at most once by the evaluation of an expression. Furthermore, the prior value shall be read only to determine the value to be stored.”

15.4 运算符总结

到此为止，除了和指针相关的运算符还没讲之外，其他运算符都讲过了，是时候做一个总结了。

运算符+ - * / % ><>= <= == != & | ^ 以及各种复合赋值运算符要求两边的操作数类型一致，条件运算符?:要求后两个操作数类型一致，这些运算符在计算之前都需要做 Usual Arithmetic Conversion。

下面按优先级从高到低的顺序总结一下C语言的运算符，每一条所列的各运算符具有相同的优先级，对于同一优先级的多个运算符按什么顺序计算也有说明，双目运算符就简单地用“左结合”或“右结合”来说明了。和指针有关的运算符* & ->也在这里列出来了，到第22章再详细解释。

1. 标识符、常量、字符串和用()括号套起来的表达式是组成表达式的最基本单元，在运算中做操作数，优先级最高。

2. 后缀运算符，包括数组取下标[]、函数调用()、结构体取成员“.”、指向结构体的指针取成员->、后缀自增++、后缀自减--。如果一个操作数后面有多个后缀，按照离操作数从近到远的顺序（也就是从左到右）依次计算，比如 a.name++，先算a.name，再++，这里的.name应该看成a的一个后缀，而不是把.看成双目运算符。

3. 单目运算符，包括前缀自增++、前缀自减--、sizeof、类型转换()、取地址运算&、指针间接寻址*、正号+、负号-、按位取反~、逻辑非!。如果一个操作数前面有多个前缀，按照离操作数从近到远的顺序（也就是从右到左）依次计算，比如!~a，先算~a，再求!。

4. 乘*、除/、模%运算符。这三个运算符是左结合的。

5. 加+、减-运算符。左结合。

6. 移位运算符<<和>>。左结合。

7. 关系运算符<><= >=。左结合。

8. 相等性运算符==和!=。左结合。

9. 按位与&。左结合。

10. 按位异或^。左结合。

11. 按位或|。左结合。

12. 逻辑与&&。左结合。

13. 逻辑或||。左结合。

14．条件运算符:?。在第 4.2 节讲过 Dangling-else 问题，条件运算符也有类似的问题。例如 a ? b : c ? d : e 是看成(a ? b : c) ? d : e 还是 a ? b : (c ? d : e)呢？C 语言规定是后者。

15．赋值=和各种复合赋值（*= /= %= += -= <<= >>= &= ^= |=）。在双目运算符中只有赋值和复合赋值是右结合的。

16．逗号运算符。左结合。

参考文献[3]第 2 章也有这样一个列表，但是对于结合性解释得不够清楚。左结合和右结合这两个概念只对双目运算符有意义，对于前缀、后缀和三目运算符我单独做了说明。C 语言表达式的详细语法规则可以查阅参考文献[8]的 Annex A.2，其实语法规则并不是用优先级和结合性这两个概念来表述的，有一些细节用优先级和结合性是表达不了的，只有看 C99 才能了解完整的语法规则。

习题

1．以下代码得到的 sum 是 0xffff，对吗？

```
int i = 0;
unsigned int sum = 0;
for (; i < 16; i++)
        sum = sum + 1U<<i;
```

2．以下代码定义了指针变量 p 并把它放在一个表达式中参与运算：

```
char *p = "Hello";
size_t n = sizeof (int) * p;
```

虽然还没学到指针类型，但现在我们不关心运算结果，而是关心表达式的语法解析。sizeof (int) * p 这个表达式应该怎么理解呢？一种理解是 sizeof(int)乘以 p，另一种理解是，*号、(int)和 sizeof 是依次作用于 p 的前缀运算符。动手试验一下看编译器会是怎么理解的，想想为什么编译器会这样理解。

第 16 章

计算机体系结构基础

现代计算机都是基于 Von Neumann 体系结构的，不管是嵌入式系统、PC 还是服务器。这种体系结构的主要特点是：CPU（Central Processing Unit，中央处理器，或简称处理器 Processor）和内存（Memory）是计算机的两个主要组成部分，内存中保存着数据和指令，CPU 从内存中取指令（Fetch）执行，其中有些指令让 CPU 做运算，有些指令让 CPU 读写内存中的数据。本章简要介绍组成计算机的 CPU、内存和设备以及它们之间的关系，为后续章节的学习打下基础。

16.1 内存与地址

我们都见过如图 16.1 所示挂在墙上的很多个邮箱，每个邮箱有一个房间编号，根据房间编号找到相应的邮箱投入信件或取出信件。内存与此类似，每个内存单元有一个地址（Address），内存地址是从 0 开始编号的整数，CPU 通过地址找到相应的内存单元，取其中的指令或者读写其中的数据。与邮箱不同的是，一个地址所对应的内存单元不能存很多东西，只能存一个字节，以前讲过的 int、float 等多字节的数据类型保存在内存中要占用连续的多个地址，这种情况下数据的地址是它所占内存单元的起始地址。

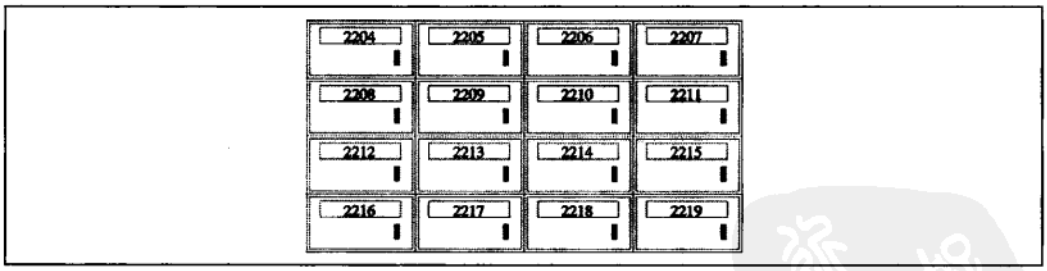


图 16.1 邮箱的地址

16.2 CPU

CPU 总是周而复始地做同一件事：从内存取指令，然后解释执行它，然后再取下一条指令，再解释执行。CPU 最核心的功能单元包括：

- 寄存器（Register）。是 CPU 内部的高速存储器，像内存一样可以存取数据，但比访问内存快得多。随后的几章我们会详细介绍 x86 的寄存器 eax、

esp、eip 等，有些寄存器只能用于某种特定的用途，比如 eip 用作程序计数器，这称为特殊寄存器（Special-purpose Register），而另外一些寄存器可以用在各种运算和读写内存的指令中，比如 eax 寄存器，这称为通用寄存器（General-purpose Register）。

- 程序计数器（Program Counter，PC）。是一种特殊寄存器，保存着 CPU 取下一条指令的地址，CPU 按程序计数器保存的地址去内存中取指令然后解释执行，这时程序计数器保存的地址会自动加上该指令的长度，指向内存中的下一条指令。
- 指令译码器（Instruction Decoder）。CPU 取上来的指令由若干个字节组成，这些字节中有些位表示内存地址，有些位表示寄存器编号，有些位表示这种指令做什么操作，是加减乘除还是读写内存，指令译码器负责解释这条指令的含义，然后调动相应的执行单元去执行它。
- 算术逻辑单元（Arithmetic and Logic Unit，ALU）。如果译码器将一条指令解释为运算指令，就调动算术逻辑单元去做运算，比如加减乘除、位运算、逻辑运算。指令中会指示运算结果保存到哪里，可能保存到寄存器中，也可能保存到内存中。
- 地址和数据总线（Bus）。CPU 和内存之间用地址总线、数据总线和控制线连接起来，每条线上有 1 和 0 两种状态。如果在执行指令过程中需要访问内存，比如从内存读一个数到寄存器，执行过程可以想象成这样（如图 16.2 所示）。

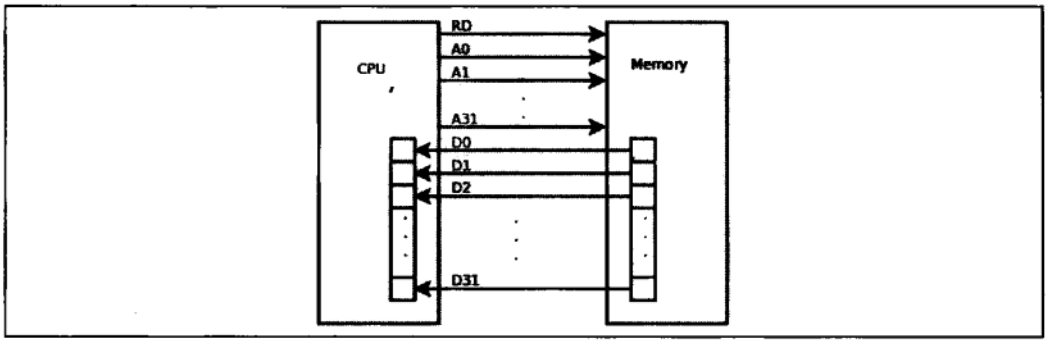


图 16.2　访问内存读数据的过程

1．CPU 内部将寄存器对接到数据总线上，使寄存器的每一位对接到一条数据线，等待接收数据。

2．CPU 通过控制线发一个读请求，并且将内存地址通过地址线发给内存。

3．内存收到地址和读请求之后，将相应的内存单元对接到数据总线的另一端，这样，内存单元每一位的 1 或 0 状态通过一条数据线到达 CPU 寄存器中相应的位，就完成了数据传送。

4．往内存里写数据的过程与此类似，只是数据线上的传输方向相反。

图 16.2 中画了 32 条地址线和 32 条数据线，CPU 寄存器也是 32 位，可以说这种

体系结构是 32 位的，比如 x86 就是这样的体系结构，目前主流的处理器是 32 位或 64 位的。地址线、数据线和 CPU 寄存器的位数通常是一致的，从图 16.2 可以看出数据线和 CPU 寄存器的位数应该一致，另外有些寄存器（比如程序计数器）需要保存一个内存地址，因而地址线和 CPU 寄存器的位数也应该一致。处理器的位数也称为字长，字（Word）这个概念用得比较混乱，在有些上下文中指 16 位，在有些上下文中指 32 位（这种情况下 16 位被称为半字 Half Word），在有些上下文中指处理器的字长，如果处理器是 32 位那么一个字就是 32 位，如果处理器是 64 位那么一个字就是 64 位。32 位计算机有 32 条地址线，地址空间（Address Space）是 0x00000000～0xffffffff，共 4GB，而 64 位计算机有更大的地址空间。

最后还要说明一点，本节所说的地址线、数据线是指 CPU 的内总线，是直接和 CPU 的执行单元相连的，内总线经过 MMU 和总线接口的转换之后引出到芯片引脚才是外总线，外地址线和外数据线的位数都有可能和内总线不同，例如 32 位处理器的外地址总线可寻址的空间可以大于 4GB，到第 16.4 节再详细解释。

我们结合表 1.1 看一下 CPU 取指执行的过程，如图 16.3 所示。

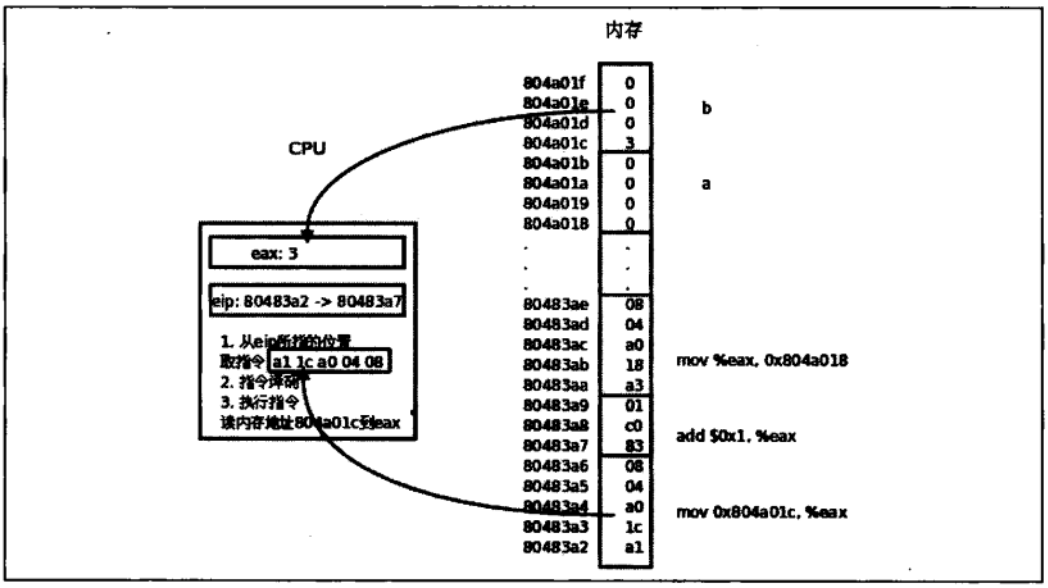


图 16.3 CPU 的取指执行过程

1．eip 寄存器指向地址 0x80483a2，CPU 从这里开始取一条 5 个字节的指令，然后 eip 寄存器指向下一条指令的起始地址 0x80483a7。

2．CPU 对这 5 个字节译码，得知这条指令要求从地址 0x804a01c 开始取 4 个字节保存到 eax 寄存器。

3．执行指令，读内存，取上来的数是 3，保存到 eax 寄存器。注意，地址 0x804a01c～0x804a01f 里存储的四个字节不能按地址从低到高的顺序看成 0x03000000，而要按地址从高到低的顺序看成 0x00000003。也就是说，对于多字节的整数类型，低地址保存的是整数的低位，这称为小端（Little Endian）字节序（Byte Order）。x86 平台是小端字节序的，而另外一些平台规定低地址保存整数的高位，称为大端（Big Endian）字节序。注意上图只画了前三步，剩下的步骤读者可以自己画图理解。

4．CPU 从 eip 寄存器指向的地址取一条 3 个字节的指令，然后 eip 寄存器指向下一条指令的起始地址 0x80483aa。

5．CPU 对这 3 个字节译码，得知这条指令要求把 eax 寄存器的值加 1，结果仍保存到 eax 寄存器。

6．执行指令，现在 eax 寄存器中的数是 4。

7．CPU 从 eip 寄存器指向的地址取一条 5 个字节的指令，然后 eip 寄存器指向下一条指令的起始地址 0x80483af。

8．CPU 对这 5 个字节译码，得知这条指令要求把 eax 寄存器的值保存到从地址 0x804a018 开始的 4 个字节。

10．执行指令，把 4 这个值保存到从地址 0x804a018 开始的 4 个字节（按小端字节序保存）。

16.3　设备

CPU 执行指令除了访问内存之外还要访问很多设备（Device），如键盘、鼠标、硬盘、显示器等，那么它们和 CPU 之间如何连接呢？如图 16.4 所示。

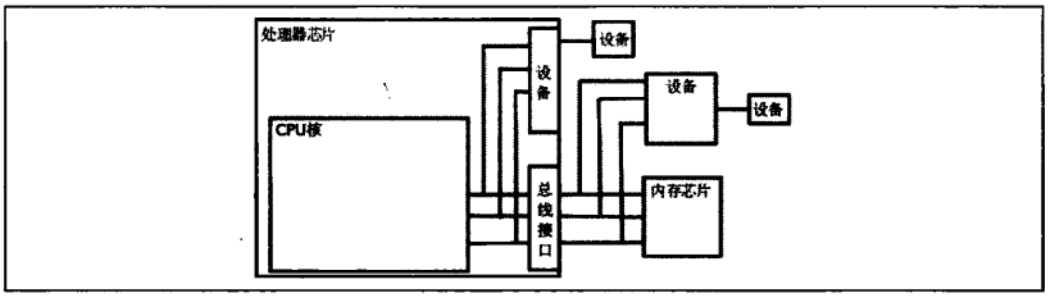


图 16.4　设备

有些设备像内存芯片一样连接到处理器的地址总线和数据总线，正因为地址线和数据线上可以挂多个设备和内存芯片所以才叫“总线”，但不同的设备和内存芯片应该占不同的地址范围。访问这种设备就像访问内存一样，按地址读写即可，但和访问内存不同的是，往一个地址写数据只是给设备发一个命令，数据不一定要保存，而从一个地址读数据也不一定是读先前保存在这个地址的数据，而是得到设备的当前状态。设备中可供读写访问的单元通常称为设备寄存器（注意和 CPU 寄存器不是一回事），操作设备的过程就是读写这些设备寄存器的过程，比如向串口发送寄存器里写数据，串口设备就会把数据发送出去，读串口接收寄存器的值，就可以读取串口设备接收到的数据。

还有一些设备集成在处理器芯片中。在图 16.4 中，从 CPU 核引出的地址和数据总线有一端经总线接口引出到芯片引脚上了，还有一端没有引出，而是接到芯片内部集成的设备上，无论是在 CPU 外部接总线的设备还是在 CPU 内部接总线的设备都有各自的地址范围，都可以像访问内存一样访问，很多体系结构（比如 ARM）采用这种方式操作设备，称为内存映射 I/O（Memory-mapped I/O）。但是

x86 比较特殊，x86 对于设备有独立的端口地址空间，CPU 核需要引出额外的地址线来连接片内设备（和访问内存所用的地址线不同），访问设备寄存器时用特殊的 in/out 指令，而不是和访问内存用同样的指令，这种方式称为端口 I/O（Port I/O）。

从 CPU 的角度来看，访问设备只有内存映射 I/O 和端口 I/O 两种，要么像内存一样访问，要么用一种专用的指令访问。其实访问设备是相当复杂的，计算机的设备五花八门，各种设备的性能要求都不一样，有的要求带宽大，有的要求响应快，有的要求热插拔，于是出现了各种适应不同要求的设备总线，比如 PCI、AGP、USB、1394、SATA 等，这些设备总线并不直接和 CPU 相连，CPU 通过内存映射 I/O 或端口 I/O 访问相应的总线控制器，通过总线控制器再去访问挂在总线上的设备。所以上图中标有“设备”的框可能是实际的设备，也可能是设备总线的控制器。

在 x86 平台上，硬盘是挂在 IDE、SATA 或 SCSI 总线上的设备，保存在硬盘上的程序是不能被 CPU 直接取指令执行的，操作系统在执行程序时会把它从硬盘拷贝到内存，这样 CPU 才能取指令执行，这个过程称为加载（Load）。程序加载到内存之后，成为操作系统调度执行的一个任务，就称为进程（Process）。进程和程序不是一一对应的。一个程序可以多次加载到内存，成为同时运行的多个进程，例如可以同时开多个终端窗口，每个窗口都运行一个 Shell 进程，而它们对应的程序都是磁盘上的/bin/bash 文件。

操作系统（Operating System）本身也是一段保存在磁盘上的程序，计算机在启动时执行一段固定的启动代码（称为 Bootloader）首先把操作系统从磁盘加载到内存，然后执行操作系统中的代码把用户需要的其他程序加载到内存。操作系统和其他用户程序的不同之处在于：操作系统是常驻内存的，而其他用户程序则不一定，用户需要运行哪个程序，操作系统就把它加载到内存，用户不需要哪个程序，操作系统就把它终止掉，释放它所占的内存。操作系统最核心的功能是管理进程调度、管理内存的分配使用和管理各种设备，做这些工作的程序称为内核（Kernel），在我的系统上内核程序是/boot/vmlinuz-2.6.28-13-generic 文件，它在计算机启动时加载到内存并常驻内存。广义上操作系统的概念还包括一些必不可少的用户程序，比如 Shell 是每个 Linux 系统必不可少的，而 Office 办公套件则是可有可无的，所以前者也属于广义上操作系统的范畴，而后者属于应用软件。

访问设备还有一点和访问内存不同。内存只是保存数据而不会产生新的数据，如果 CPU 不去读它，它也不需要主动提供数据给 CPU，所以内存总是被动地等待被读或者被写。而设备往往会自己产生数据，并且需要主动通知 CPU 来读这些数据，例如敲键盘产生一个输入字符，用户希望计算机马上响应自己的输入，这就要求键盘设备主动通知 CPU 来读这个字符并做相应处理，给用户响应。这是由中断（Interrupt）机制实现的，每个设备都有一条中断线，通过中断控制器连接到 CPU，当设备需要主动通知 CPU 时就引发一个中断信号，CPU 正在执行的指令将被打断，程序计数器会指向某个固定的地址（这个地址由体系结构定义），于是 CPU 从这个地址开始取指令（或者说跳转到这个地址），执行中断服务程序（Interrupt Service Routine，ISR），完成中断处理之后再返回先前被打断的地方执行后续指令。比如某种体系结构规定发生中断时跳转到地址 0x00000010 执行，那么就要事先把一段 ISR 程序加载到这个地址，ISR 程序是内核代码的一部分，在这段代码中首先判断是哪个设备引发了中断，然后调用该设备的中断处理函数做进一步处理。

由于各种设备的操作方法各不相同，每种设备都需要专门的设备驱动程序（Device Driver），一个操作系统为了支持广泛的设备就需要有大量的设备驱动程序，事实上 Linux 内核源代码中绝大部分是设备驱动程序。设备驱动程序通常是内核里的一组函数，通过读写设备寄存器实现对设备的初始化、读、写等操作，有些设备还要提供一个中断处理函数供 ISR 调用。

16.4　MMU

现代操作系统普遍采用虚拟内存管理（Virtual Memory Management）机制，这需要处理器中的 MMU（Memory Management Unit，内存管理单元）提供支持，本节简要介绍 MMU 的作用。

首先引入两个概念，虚拟地址和物理地址。如果处理器没有 MMU，或者有 MMU 但没有启用，CPU 执行单元发出的内存地址将直接传到芯片引脚上，被内存芯片（以下称为物理内存，以便与虚拟内存区分）接收，这称为物理地址（Physical Address，以下简称 PA），如图 16.5 所示。

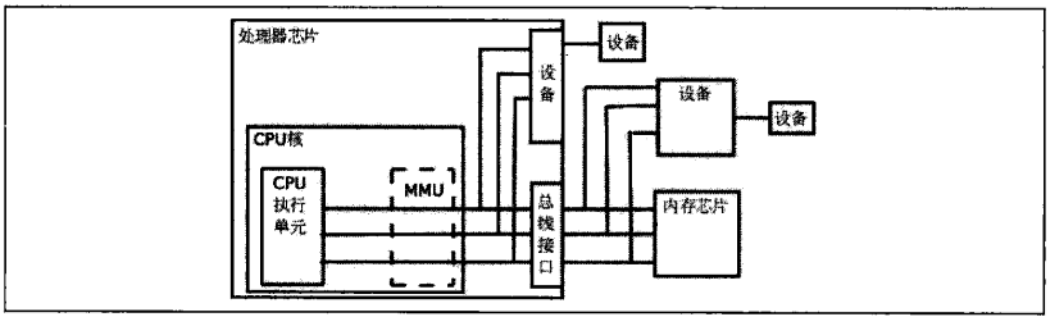


图 16.5　物理地址

如果处理器启用了 MMU，CPU 执行单元发出的内存地址将被 MMU 截获，从 CPU 到 MMU 的地址称为虚拟地址（Virtual Address，以下简称 VA），而 MMU 将这个地址翻译成另一个地址发到 CPU 芯片的外部地址引脚上，也就是将 VA 映射成 PA，如图 16.6 所示。

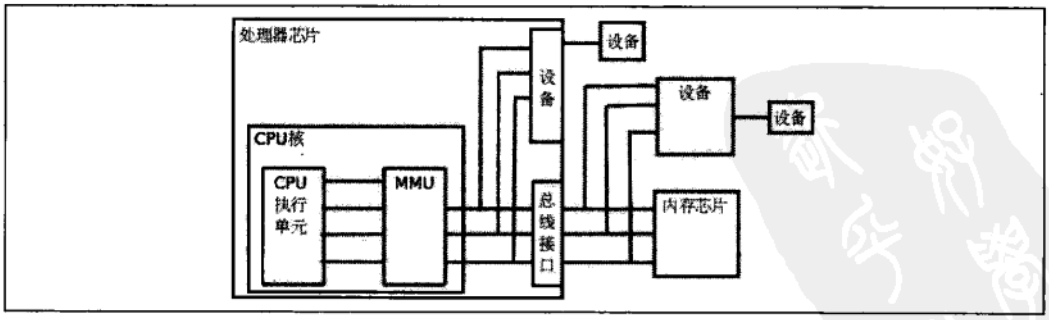


图 16.6　虚拟地址

如果是 32 位处理器，则内地址总线是 32 位的，与 CPU 执行单元相连（图中只是示意性地画了 4 条地址线），而经过 MMU 转换之后的外地址总线则不一定是 32 位的。也就是说，虚拟地址空间和物理地址空间是独立的，32 位处理器的虚拟地

址空间是4GB，而物理地址空间既可以大于也可以小于4GB。

MMU将VA映射到PA是以页（Page）为单位的，32位处理器的页尺寸通常是4KB。例如，MMU可以通过一个映射项将VA的一页0xb7001000～0xb7001fff映射到PA的一页0x2000～0x2fff，如果CPU执行单元要访问虚拟地址0xb7001008，则实际访问到的物理地址是0x2008。物理内存中的页称为物理页面或者页帧（Page Frame）。虚拟内存的哪个页面映射到物理内存的哪个页帧是通过页表（Page Table）来描述的，页表保存在物理内存中，MMU会查找页表来确定一个VA应该映射到什么PA。

操作系统和MMU是这样配合的：

1．操作系统在初始化或分配、释放内存时会执行一些指令在物理内存中填写页表，然后用指令设置MMU，告诉MMU页表在物理内存中的什么位置。

2．设置好之后，CPU每次执行访问内存的指令都会自动引发MMU做查表和地址转换操作，地址转换操作由硬件自动完成，不需要用指令控制MMU去做。

我们在程序中使用的变量和函数都有各自的地址，程序被编译后，这些地址就成了指令中的地址，指令中的地址被CPU解释执行，就成了CPU执行单元发出的内存地址，所以在启用MMU的情况下，程序中使用的地址都是虚拟地址，都会引发MMU做查表和地址转换操作。那为什么要设计这么复杂的内存管理机制呢？多了一层VA到PA的转换到底换来了什么好处？All problems in computer science can be solved by another level of indirection.还记得这句话吗？多了一层间接必然是为了解决什么问题的，等讲完了必要的预备知识之后，将在第19.5节讨论虚拟内存管理机制的作用。

MMU除了做地址转换之外，还提供内存保护机制。各种体系结构都有用户模式（User Mode）和特权模式（Privileged Mode）之分，如图16.7所示，操作系统可以在页表中设置每个内存页面的访问权限，有些页面不允许访问，有些页面只有在CPU处于特权模式时才允许访问，有些页面在用户模式和特权模式都可以访问，访问权限又分为可读、可写和可执行三种。这样设定好之后，当CPU要访问一个VA时，MMU会检查CPU当前处于用户模式还是特权模式，访问内存的目的是读数据、写数据还是取指令，如果和操作系统设定的页面权限相符，就允许访问，把它转换成PA，否则不允许访问，产生一个异常（Exception）。异常的处理过程和中断类似，不同的是中断由外部设备产生而异常由CPU内部产生，中断产生的原因和CPU当前执行的指令无关，而异常的产生就是由于CPU当前执行的指令出了问题，例如访问内存的指令被MMU检查出权限错误，除法指令的除数为0等都会产生异常。

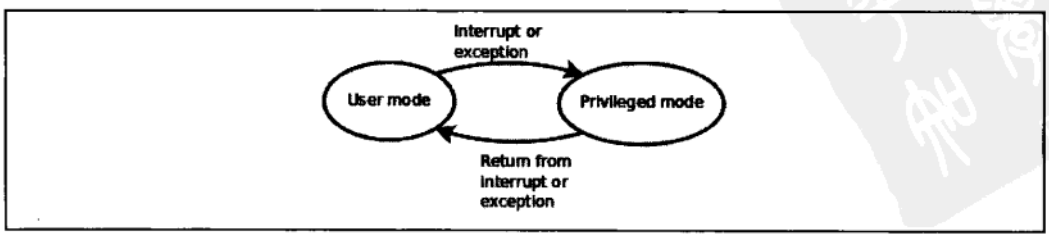


图16.7 处理器模式

通常操作系统把虚拟地址空间划分为用户空间和内核空间，例如 x86 平台的 Linux 系统虚拟地址空间是 0x00000000～0xffffffff，前 3GB（0x00000000～0xbfffffff）是用户空间，后 1GB（0xc0000000～0xffffffff）是内核空间。用户程序加载到用户空间，在用户模式下执行，不能访问内核中的数据，也不能跳转到内核代码中执行。这样可以保护内核，如果一个进程访问了非法地址，顶多这一个进程崩溃，而不会影响到内核和整个系统的稳定性。CPU 在产生中断或异常时不仅会跳转到中断或异常服务程序，还会自动切换模式，从用户模式切换到特权模式，因此从中断或异常服务程序可以跳转到内核代码中执行。事实上，整个内核就是由各种中断和异常处理程序组成的。总结一下：**在正常情况下处理器在用户模式执行用户程序，在中断或异常情况下处理器切换到特权模式执行内核程序，处理完中断或异常之后再返回用户模式继续执行用户程序。**

段错误我们已经遇到过很多次了，它是这样产生的：

1．用户程序要访问的一个 VA，经 MMU 检查无权访问。

2．MMU 产生一个异常，CPU 从用户模式切换到特权模式，跳转到内核代码中执行异常服务程序。

3．内核把这个异常解释为段错误，把引发异常的进程终止掉。

16.5　Memory Hierarchy

硬盘、内存、CPU 寄存器，还有本节要讲的 Cache，这些都是存储器，计算机为什么要有这么多种存储器呢？这些存储器各自有什么特点？这是本节要讨论的问题。

由于硬件技术的限制，我们可以制造出容量很小但访问速度很快的存储器，也可以制造出容量很大但访问速度很慢的存储器，但不可能两边的好处都占着，不可能制造出访问速度又快容量又大的存储器。因此，现代计算机都把存储器分成若干级，称为 Memory Hierarchy，按照离 CPU 由近到远的顺序依次是 CPU 寄存器、Cache、内存、硬盘，越靠近 CPU 的存储器容量越小但访问速度越快，图 16.8 给出了各种存储器的容量和访问速度的典型值，具体说明如表 16.1 所示。

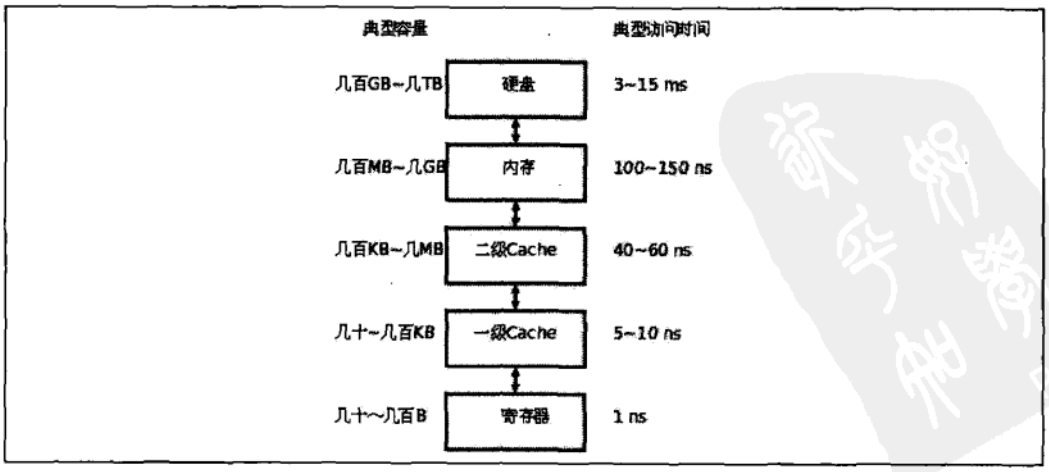


图 16.8　Memory Hierarchy

表 16.1 Memory Hierarchy

存储器类型	位于哪里	存储容量	半导体工艺	访问时间	如何访问
CPU 寄存器	位于 CPU 执行单元中	CPU 寄存器通常只有几个到几十个，每个寄存器的容量取决于 CPU 的字长，所以一共只有几十到几百字节	“寄存器”这个名字就是一种数字电路的名字，它由一组触发器（Flip-flop）组成，每个触发器保存一个 bit 的数据，可以做存取和移位等操作。计算机掉电时寄存器中保存的数据会丢失	寄存器是访问速度最快的存储器，典型的访问时间是几纳秒	使用哪个寄存器，如何使用寄存器，这些都是由指令决定的
Cache	和 MMU 一样位于 CPU 核中	Cache 通常分为几级，最典型的是如图 16.8 所示的两级 Cache，一级 Cache 更靠近 CPU 执行单元，二级 Cache 更靠近物理内存，通常一级 Cache 有几十到几百 KB，二级 Cache 有几百 KB 到几 MB	Cache 和内存都是由 RAM（Random Access Memory）组成的，可以根据地址随机访问，计算机掉电时 RAM 中保存的数据会丢失。不同的是，Cache 通常由 SRAM（Static RAM，静态 RAM）组成，而内存通常由 DRAM（Dynamic RAM，动态 RAM）组成。DRAM 电路比 SRAM 简单，存储容量可以做得更大，但 DRAM 的访问速度比 SRAM 慢	典型的访问时间是几十纳秒	Cache 缓存最近访问过的内存数据，由于 Cache 的访问速度是内存的几十倍，所以有效利用 Cache 可以大大提高计算机的整体性能。一级 Cache 是这样工作的：CPU 执行单元要访问内存时首先发出 VA，Cache 利用 VA 查找相应的数据有没有被缓存，如果 Cache 中有就不需要访问物理内存了，如果是读操作就直接将 Cache 中的数据传给 CPU 寄存器，如果是写操作就直接改写到 Cache 中；如果 Cache 没有缓存该数据，就去物理内存中取数据，但并不是要哪个字节就取哪个字节，而是把相邻的几十个字节都取上来缓存着，以备下次用到，这称为一个 Cache Line，典型的 Cache Line 大小是 32~256 字节。如果计算机还配

续表

存储器类型	位于哪里	存储容量	半导体工艺	访问时间	如何访问
					置了二级缓存，则在访问物理内存之前先用 PA 去二级缓存中查找。一级缓存是用 VA 寻址的，二级缓存是用 PA 寻址的，这是它们的区别。Cache 所做的工作是由硬件自动完成的，而不是像寄存器一样由指令决定先做什么后做什么
内存	位于 CPU 外的芯片，与 CPU 通过地址和数据总线相连	典型的存储容量是几百 MB 到几 GB	由 DRAM 组成，详见上面关于 Cache 的说明	典型的访问时间是几百纳秒	内存是通过地址来访问的，在启用 MMU 的情况下，程序指令中的地址是 VA，而访问内存用的是 PA，它们之间的映射关系由操作系统维护
硬盘	位于设备总线上，并不直接和 CPU 相连，CPU 通过设备总线的控制器访问硬盘	典型的存储容量是几百 GB 到几 TB	硬盘由磁性介质和磁头组成，访问硬盘时存在机械运动，磁头要移动，磁性介质要旋转，机械运动的速度很难提高到电子的速度，所以访问速度很受限制。保存在硬盘上的数据掉电后不会丢失	典型的访问时间是几毫秒，是寄存器访问时间的 10^6 倍	由驱动程序操作设备总线控制器去访问。由于硬盘的访问速度较慢，操作系统通常一次从硬盘上读几个页面到内存中缓存起来，如果这几个页面后来都被程序访问到了，那么这一次读硬盘的时间就可以分摊（Amortize）给程序的多次访问了

对表 16.1 总结如下。

- 寄存器、Cache 和内存中的数据都是掉电丢失的，这称为易失性存储器（Volatile Memory），与之相对的，硬盘是一种非易失性存储器（Non-volatile Memory）。
- 除了访问寄存器由程序指令直接控制之外，访问其他存储器都不是由指令直接控制的，有些是硬件自动完成的，有些是操作系统配合硬件完成的。
- Cache 从内存取数据时会预取一个 Cache Line 缓存起来，操作系统从硬盘读数据时会预读几个页面缓存起来，都是希望这些数据以后会被程序访问到。大多数程序的行为都具有局部性（Locality）的特点：它们会花费大量的时间反复执行一小段代码（例如循环），或者反复访问一个很小的地址范围中的数据（例如访问一个数组）。所以预读+缓存的办法是很有效

的：CPU 取一条指令，我把和它相邻的指令也都缓存起来，CPU 很可能马上就会取到；CPU 访问一个数据，我把和它相邻的数据也都缓存起来，CPU 很可能马上就会访问到。设想有两台计算机，一台有 256KB 的 Cache，另一台没有 Cache，两台计算机的内存都是 512MB 的，硬盘都是 100GB 的，虽然多出来 256KB 的 Cache 与内存、硬盘的容量相比微不足道，但访问 Cache 比访问内存、硬盘快几个数量级，由于局部性原理，CPU 大部分时间在和 Cache 打交道，有 Cache 的计算机明显会快很多。高速存储器的容量只能做得很小，却能显著提升计算机的性能，这就是 Memory Hierarchy 的意义所在。

第 17 章

x86 汇编程序基础

要彻底搞清楚 C 语言的原理，就必须深入到指令一层去理解。你写一行 C 代码，编译器会生成什么样的指令，要做到心中有数。本章介绍汇编程序的一些基础知识。汇编不是本书的重点，本书要求读者能看懂基本的汇编程序而不要求会写汇编程序，下一章将在汇编的基础上讨论 C 语言的原理。

17.1 最简单的汇编程序

例 17.1 最简单的汇编程序

```
#PURPOSE: Simple program that exits and returns a
#         status code back to the Linux kernel
#
#INPUT:   none
#
#OUTPUT:  returns a status code. This can be viewed
#         by typing
#
#         echo $?
#
#         after running the program
#
#VARIABLES:
#         %eax holds the system call number
#         %ebx holds the return status
#
 .section .data

 .section .text
 .globl _start
_start:
 movl $1, %eax  # this is the linux kernel command
                # number (system call) for exiting
                # a program

 movl $4, %ebx  # this is the status number we will
                # return to the operating system.
                # Change this around and it will
                # return different things to
                # echo $?

 int $0x80      # this wakes up the kernel to run
                # the exit command
```

把这个程序保存成文件 hello.s（汇编程序通常以.s 作为文件名后缀），用汇编器 as 把汇编程序中的助记符翻译成机器指令，生成目标文件 hello.o：

```
$ as hello.s -o hello.o
```

然后用链接器（Linker，或 Link Editor）ld 把目标文件 hello.o 链接成可执行文件 hello：

```
$ ld hello.o -o hello
```

为什么用汇编器翻译成机器指令了还不行，还要有一个链接的步骤呢？链接主要有两个作用，一是修改目标文件中的信息，对地址做重定位，在第 17.5.2 节详细解释，二是把多个目标文件合并成一个可执行文件，在第 18.2 节详细解释。我们这个例子虽然只有一个目标文件，但也需要经过链接才能成为可执行文件。

现在执行这个程序，它只做了一件事就是退出，退出状态是 4，第 3.2 节讲过在 Shell 中可以用特殊变量$?得到上一条命令的退出状态：

```
$ ./hello
$ echo $?
4
```

所以这段汇编代码相当于在 C 程序的 main 函数中 return 4;。为什么呢？我们在第 18.2 节详细解释。

下面逐行分析这个汇编程序。首先，#号表示单行注释，类似于 C 语言的//注释。

```
.section .data
```

汇编程序中以.开头的名称并不是指令的助记符，不会被翻译成机器指令，而是给汇编器一些特殊指示，称为汇编指示（Assembler Directive）或伪指令（Pseudo-operation），由于它不是真正的指令所以加个“伪”字。.section 指示把代码划分成若干个段（Section），程序被操作系统加载执行时，每个段被加载到不同的地址，操作系统对不同的页面设置不同的读、写、执行权限。.data 段保存程序的数据，是可读可写的，相当于 C 程序的全局变量。本程序中没有定义数据，所以.data 段是空的。

```
.section .text
```

.text 段保存代码，是只读和可执行的，后面那些指令都属于.text 段。

```
.globl _start
```

_start 是一个符号（Symbol），符号在汇编程序中代表一个地址，可以用在指令中，汇编程序经过汇编器的处理之后，所有的符号都被替换成它所代表的地址值。在 C 语言中我们通过变量名访问一个变量，其实就是读写从某个地址开始的内存单元，我们通过函数名调用一个函数，其实就是跳转到该函数第一条指令所在的地址，所以变量名和函数名都是符号，本质上是代表内存地址的。

.globl 指示告诉汇编器，_start 这个符号要被链接器用到，所以要在目标文件的符号表中标记它是一个全局符号（在第 17.5.1 节详细解释）。_start 就像 C 程序的 main 函数一样特殊，是整个程序的入口，链接器在链接时会查找目标文件中的 _start 符号代表的地址，把它设置为整个程序的入口地址，所以每个汇编程序都要提供一个_start 符号并且用.globl 声明。如果一个符号没有用.globl 声明，就表示这个符号不会被链接器用到。

```
    _start:
```

这里定义了_start 符号，汇编器在翻译汇编程序时会计算每个数据对象和每条指令的地址，当看到这样一个符号定义时，就把它后面一条指令的地址作为这个符号所代表的地址。而_start 这个符号又比较特殊，它所代表的地址是整个程序的入口地址，所以下一条指令 movl $1, %eax 就成了程序中第一条被执行的指令。

```
    movl $1, %eax
```

这是一条数据传送指令，这条指令要求 CPU 内部产生一个数字 1 并保存到 eax 寄存器中。mov 的后缀 l 表示 long，说明是 32 位的传送指令。这条指令不要求 CPU 读内存，1 这个数是在 CPU 内部产生的，称为立即数（Immediate）。在汇编程序中，立即数前面要加$，寄存器名前面要加%，以便跟符号名区分开。以后我们会看到 mov 指令还有另外几种形式，但数据传送方向都是一样的，第一个操作数总是源操作数，第二个操作数总是目标操作数。

```
    movl $4, %ebx
```

和上一条指令类似，生成一个立即数 4 并保存到 ebx 寄存器中。

```
    int $0x80
```

前两条指令都是为这条指令做准备的，执行这条指令时发生以下动作：

1．int 指令称为软中断指令，可以用这条指令故意产生一个异常，上一章讲过，异常的处理和中断类似，CPU 从用户模式切换到特权模式，然后跳转到内核代码中执行异常处理程序。

2．int 指令中的立即数 0x80 是一个参数，在异常处理程序中要根据这个参数决定如何处理，在 Linux 内核中 int $0x80 这种异常称为系统调用（System Call）。内核提供了很多系统服务供用户程序使用，但这些系统服务不能像库函数（比如 printf）那样调用，因为在执行用户程序时 CPU 处于用户模式，不能直接调用内核函数，所以需要通过系统调用切换 CPU 模式，经由异常处理程序进入内核，用户程序只能通过寄存器传几个参数，之后就要按内核设计好的代码路线走，而不能由用户程序随心所欲，想调哪个内核函数就调哪个内核函数，这样可以保证系统服务被安全地调用。在调用结束之后，CPU 再切换回用户模式，继续执行 int $0x80 的下一条指令，在用户程序看来就像函数调用和返回一样。

3．eax 和 ebx 的值是传递给系统调用的两个参数。eax 的值是系统调用号，Linux

的各种系统调用都是由 int $0x80 指令引发的，内核需要通过 eax 判断用户要调哪个系统调用，_exit 的系统调用号是 1。ebx 的值是传给_exit 的参数，表示退出状态。大多数系统调用完成之后会返回用户空间继续执行后面的指令，而_exit 系统调用比较特殊，它会终止掉当前进程，而不是返回用户空间继续执行。

> **x86 汇编的两种语法：intel 语法和 AT&T 语法**
>
> x86 汇编一直存在两种不同的语法，在 intel 的官方文档中使用 intel 语法，Windows 也使用 intel 语法，而 UNIX 平台的汇编器一直使用 AT&T 语法，所以本书使用 AT&T 语法。movl %edx,%eax 这条指令如果用 intel 语法来写，就是 mov eax,edx，寄存器名不加%号，源操作数和目标操作数的位置互换，字长也不是用指令的后缀 l 表示而是用另外的方式表示。本书不详细讨论这两种语法之间的区别，读者可以查阅参考文献[22]。
>
> 介绍 x86 汇编的书很多，UNIX 平台的书都采用 AT&T 语法，例如参考文献[2]，其他书一般采用 intel 语法，例如参考文献[23]。

习题

1. 把本节例子中的 int $0x80 指令去掉，汇编、链接也能通过，但是执行的时候出现段错误，你能解释其原因吗？

17.2 x86 的寄存器

x86 的通用寄存器有 eax、ebx、ecx、edx、edi、esi。这些寄存器在大多数指令中是可以任意选用的，比如 movl 指令可以把一个立即数传送到 eax 中，也可传送到 ebx 中。但也有一些指令规定只能用其中某个寄存器做某种用途，例如除法指令 idivl 要求被除数在 eax 寄存器中，edx 寄存器必须是 0，而除数可以在任意寄存器中，计算结果的商数保存在 eax 寄存器中（覆盖原来的被除数），余数保存在 edx 寄存器中。也就是说，通用寄存器对于某些特殊指令来说也不是通用的。

x86 的特殊寄存器有 ebp、esp、eip、eflags。eip 是程序计数器，eflags 保存着计算过程中产生的标志位，其中包括第 13.3 节讲过的进位标志、溢出标志、零标志和负数标志，在 intel 的手册中这几个标志位分别称为 CF、OF、ZF、SF。ebp 和 esp 用于维护函数调用的栈帧，在第 18.1 节详细讨论。

17.3 第二个汇编程序

例 17.2 求一组数中最大值的汇编程序

```
#PURPOSE: This program finds the maximum number of a
#         set of data items.
#
#VARIABLES: The registers have the following uses:
#
```

```
# %edi - Holds the index of the data item being examined
# %ebx - Largest data item found
# %eax - Current data item
#
# The following memory locations are used:
#
# data_items - contains the item data. A 0 is used
# to terminate the data
#
 .section .data
data_items:             # These are the data items
 .long 3,67,34,222,45,75,54,34,44,33,22,11,66,0

 .section .text
 .globl _start
_start:
 movl $0, %edi  # move 0 into the index register
 movl data_items(,%edi,4), %eax # load the first item
 movl %eax, %ebx        # since %eax is the first and only item, it's
                        the biggest

start_loop:             # start loop
 cmpl $0, %eax          # check to see if we've hit the end
 je loop_exit
 incl %edi              # load the next item
 movl data_items(,%edi,4), %eax
 cmpl %ebx, %eax        # compare values
 jle start_loop         # jump to loop beginning if the new one isn't
                        bigger
 movl %eax, %ebx        # the new one is the biggest
 jmp start_loop         # jump to loop beginning

loop_exit:
 # %ebx is the status code for the _exit system call
 # and it already has the maximum number
 movl $1, %eax          #1 is the _exit() syscall
 int $0x80
```

汇编、链接、运行：

```
$ as max.s -o max.o
$ ld max.o -o max
$ ./max
$ echo $?
222
```

这个程序在一组数中找到一个最大的数，并把它作为程序的退出状态。这组数在.data 段给出：

```
data_items:
 .long 3,67,34,222,45,75,54,34,44,33,22,11,66,0
```

.long 指示声明一组数，每个数占 32 位，相当于 C 语言中的数组。这个数组开头定义了一个符号 data_items，汇编器会把数组的首地址作为 data_items 符号所代表的地址，data_items 类似于 C 语言中的数组名。data_items 这个标号没有用.globl 声明，因为它只在这个汇编程序内部使用，链接器不需要用到这个名字。除了.long 之外，常用的数据声明还有：

- .byte，也是声明一组数，每个数占 8 位
- .ascii，例如.ascii "Hello world"，声明 11 个数，取值为相应字符的 ASCII 码。注意，和 C 语言不同，这样声明的字符串末尾是没有'\0'字符的，如果需要以'\0'结尾可以声明为.ascii "Hello world\0"。

data_items 数组的最后一个数是 0，我们在一个循环中依次比较每个数，碰到 0 的时候让循环终止。在这个循环中：

- edi 寄存器保存数组中的当前位置，每次比较完一个数就把 edi 的值加 1，指向数组中的下一个数。
- ebx 寄存器保存到目前为止找到的最大值，如果发现有更大的数就更新 ebx 的值。
- eax 寄存器保存当前要比较的数，每次更新 edi 之后，就把下一个数读到 eax 中。

```
_start:
 movl $0, %edi
```

初始化 edi，指向数组的第 0 个元素。

```
 movl data_items(,%edi,4), %eax
```

这条指令把数组的第 0 个元素传送到 eax 寄存器中。data_items 是数组的首地址，edi 的值是数组的下标，4 表示数组的每个元素占 4 字节，那么数组中第 edi 个元素的地址应该是 data_items + edi * 4，写在指令中就是 data_items(,%edi,4)，这种地址表示方式在下一节还会详细解释。

```
movl %eax, %ebx
```

ebx 的初始值也是数组的第 0 个元素。下面我们进入一个循环，循环的开头定义一个符号 start_loop，循环的末尾之后定义一个符号 loop_exit。

```
start_loop:
cmpl $0, %eax
je loop_exit
```

比较 eax 的值是不是 0，如果是 0 就说明到达数组末尾了，就要跳出循环。cmpl 指令将两个操作数相减，但计算结果并不保存，只是根据计算结果改变 eflags 寄存器中的标志位。如果两个操作数相等，则计算结果为 0，eflags 中的 ZF 位置 1。je 是一个条件跳转指令，它检查 eflags 中的 ZF 位，ZF 位为 1 则发生跳转，ZF 位为 0 则不跳转，继续执行下一条指令。可见比较指令和条件跳转指令是配合使用的，前者改变标志位，后者根据标志位决定是否跳转。je 可以理解成“jump if equal”，如果参与比较的两数相等则跳转。

```
incl %edi
movl data_items(,%edi,4), %eax
```

将 edi 的值加 1，把数组中的下一个数传送到 eax 寄存器中。

```
    cmpl %ebx, %eax
    jle start_loop
```

把当前数组元素 eax 和目前为止找到的最大值 ebx 做比较，如果前者小于等于后者，则最大值没有变，跳转到循环开头比较下一个数，否则继续执行下一条指令。jle 表示“jump if less than or equal”。

```
    movl %eax, %ebx
    jmp start_loop
```

更新了最大值 ebx 然后跳转到循环开头比较下一个数。jmp 是一个无条件跳转指令，什么条件也不判断，直接跳转。loop_exit 符号后面的指令调_exit 系统调用退出程序。

17.4　寻址方式

通过上一节的例子我们了解到，访问内存时在指令中可以用多种方式表示内存地址，比如可以用数组基地址、元素长度和下标三个量来表示，增加了寻址的灵活性。本节介绍 x86 常用的几种寻址方式（Addressing Mode）。内存寻址在指令中可以表示成如下的通用格式：

```
    ADDRESS_OR_OFFSET(%BASE_OR_OFFSET,%INDEX,MULTIPLIER)
```

它所表示的地址可以这样计算出来：

```
    FINAL ADDRESS = ADDRESS_OR_OFFSET + BASE_OR_OFFSET + INDEX * MU
LTIPLIER
```

其中 ADDRESS_OR_OFFSET 和 MULTIPLIER 必须是常数，BASE_OR_OFFSET 和 INDEX 必须是寄存器。在有些寻址方式会省略这 4 项中的某些项，相当于这些项是 0。

- 直接寻址（Direct Addressing Mode）。只使用 ADDRESS_OR_OFFSET 寻址，例如 movl ADDRESS, %eax 把 ADDRESS 地址处的 32 位数传送到 eax 寄存器。
- 变址寻址（Indexed Addressing Mode）。上一节的 movl data_items(,%edi,4), %eax 就属于这种寻址方式，用于访问数组元素比较方便。
- 间接寻址（Indirect Addressing Mode）。只使用 BASE_OR_OFFSET 寻址，例如 movl (%eax), %ebx，把 eax 寄存器的值看作地址，把内存中这个地址处的 32 位数传送到 ebx 寄存器。注意和 movl %eax, %ebx 区分开。
- 基址寻址（Base Pointer Addressing Mode）。只使用 ADDRESS_OR_OFFSET 和 BASE_OR_OFFSET 寻址，例如 movl 4(%eax), %ebx，用于访问结构体成员比较方便，例如一个结构体的基地址保存在 eax 寄存器中，其中一个成员在结构体内的偏移量是 4 字节，要把这个成员读上来就可以用这

条指令。

- 立即数寻址（Immediate Mode）。就是指令中有一个操作数是立即数，例如 movl $12, %eax 中的$12，这其实跟寻址没什么关系，但也算做一种寻址方式。
- 寄存器寻址（Register Addressing Mode）。就是指令中有一个操作数是寄存器，例如 movl $12, %eax 中的%eax，这跟内存寻址没什么关系，但也算作一种寻址方式。在汇编程序中寄存器用助记符来表示，在机器指令中则要用几个 bit 表示寄存器的编号，这几个 bit 也可以看作寄存器的地址，但是和内存地址不在一个地址空间。

17.5 ELF 文件

ELF（Executable and Linking Format）是一个开放标准，各种 UNIX 系统都支持 ELF 格式的可执行文件，它有三种不同的类型：

- 可重定位的目标文件（Relocatable，或者 Object File）
- 可执行文件（Executable）
- 共享库（Shared Object，或者 Shared Library）

共享库留到第 19.4 节再详细介绍，本节我们以例 17.2 为例讨论目标文件和可执行文件的格式。现在详细解释一下这个程序的汇编、链接、运行过程：

1．写一个汇编程序保存成文本文件 max.s。

2．汇编器读取这个文本文件转换成目标文件 max.o，目标文件由若干个 Section 组成，我们在汇编程序中声明的.section 会成为目标文件中的 Section，此外汇编器还会自动添加一些 Section（比如符号表）。

3．然后链接器把目标文件中的 Section 合并成几个 Segment[①]，生成可执行文件 max。

4．最后加载器（Loader）根据可执行文件中的 Segment 信息加载运行这个程序。

ELF 格式提供了两种不同的视角，链接器把 ELF 文件看成是 Section 的集合，而加载器把 ELF 文件看成是 Segment 的集合，如图 17.1 所示。

左边是从链接器的视角来看 ELF 文件，开头的 ELF Header 描述了体系结构和操作系统等基本信息，并指出 Section Header Table 和 Program Header Table 在文件中的什么位置，Program Header Table 在链接过程中用不到，所以是可有可无的，Section Header Table 中保存了所有 Section 的描述信息，通过 Section Header Table 可以找到每个 Section 在文件中的位置。右边是从加载器的视角来看 ELF 文件，

① Segment 也可以翻译成“段”，为了避免混淆，在本书中只把 Section 称为段，而 Segment 直接用英文。

开头是 ELF Header，Program Header Table 中保存了所有 Segment 的描述信息，Section Header Table 在加载过程中用不到，所以是可有可无的。从图 17.1 可以看出，一个 Segment 由一个或多个 Section 组成，这些 Section 加载到内存时具有相同的访问权限。有些 Section 只对链接器有意义，在运行时用不到，也不需要加载到内存，那么就不属于任何 Segment。注意 Section Header Table 和 Program Header Table 并不是一定要位于文件的开头和结尾，其位置由 ELF Header 指出，图 17.1 这么画只是为了清晰。

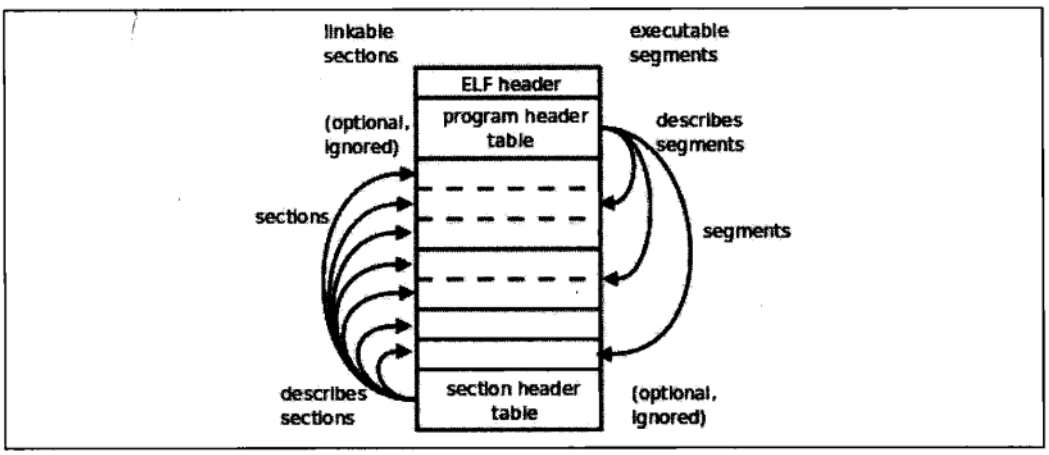


图 17.1　ELF 文件

目标文件需要链接器做进一步处理，所以一定有 Section Header Table；可执行文件需要加载运行，所以一定有 Program Header Table；而共享库既要加载运行，又要在加载时做动态链接，所以既有 Section Header Table 又有 Program Header Table。

17.5.1　目标文件

下面用 readelf 工具读出目标文件 max.o 的 ELF Header 和 Section Header Table，然后我们逐段分析。

```
$ readelf -a max.o
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              REL (Relocatable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x0
  Start of program headers:          0 (bytes into file)
  Start of section headers:          200 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           0 (bytes)
```

```
  Number of program headers:    0
  Size of section headers:              40 (bytes)
  Number of section headers:            8
  Section header string table index:    5
...
```

ELF Header 中描述了这样一些信息：

- 每个 ELF 文件开头四个字节都是 0x7f、0x45、0x4c、0x46，其中后三个字节就是“ELF”的 ASCII 码，在文件开头用几个特殊字符来标识文件类型是 UNIX 系统的惯用伎俩，这称为 Magic Number，file(1)命令就是通过分析 Magic Number 和其它一些特征来确定文件类型的；
- 文件格式是 ELF32；
- 字节序是小端；
- 操作系统是 UNIX；
- 文件类型是 Relocatable file（即可重定位的目标文件）；
- 体系结构是 Intel 80386；
- 程序的入口地址是 0x0，因为目标文件的入口地址还没确定，链接成可执行文件时才能确定入口地址；
- Program Header Table 在文件中的开始位置是 0，因为目标文件没有 Program Header Table，链接成可执行文件时才会有 Program Header Table；
- Section Header Table 在文件中的开始位置是 200，以下我们称其为文件地址，规定文件开头第一个字节的地址是 0，然后每个字节占一个地址，所以 200 是文件中的第 201 个字节；
- 此 ELF Header 的大小是 52 字节；
- Section Header Table 中有 8 个 Section Header 表项，每个表项占 40 字节，共 320 字节，所以 Section Header Table 在文件中的地址范围是 200～519（0xc8～0x207）；
- Section Header String Table 也是一个 Section，它由 Section Header Table 中的第 5 个表项来描述。

```
...
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf
Al
  [ 0]                   NULL            00000000 000000 000000 00   0  0  0
  [ 1] .text             PROGBITS        00000000 000034 00002a 00  AX  0  0 4
  [ 2] .rel.text         REL             00000000 0002b0 000010 08   6  1  4
  [ 3] .data             PROGBITS        00000000 000060 000038 00  WA  0  0 4
  [ 4] .bss              NOBITS          00000000 000098 000000 00  WA  0  0 4
  [ 5] .shstrtab         STRTAB          00000000 000098 000030 00      0  0 1
  [ 6] .symtab           SYMTAB          00000000 000208 000080 10      7  7 4
  [ 7] .strtab           STRTAB          00000000 000288 000028 00      0  0 1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)

There are no section groups in this file.
```

```
There are no program headers in this file.
...
```

从 Section Header 中读出各 Section 的描述信息，其中.text 和.data 是我们在汇编程序中声明的 Section，而其他 Section 是汇编器自动添加的。Addr 列指出这些 Section 加载到内存中的地址（虚拟地址），目标文件中各 Section 的加载地址是待定的，所以是 00000000，到链接时再确定这些地址。Off 和 Size 列指出各 Section 的起始文件地址和长度。比如.data 段从文件地址 0x60 开始，一共 0x38 个字节，回去翻一下程序，.data 段定义了 14 个 4 字节的整数，一共是 56 个字节，也就是 0x38。根据以上信息可以描绘出整个目标文件的布局，如表 17.1 所示。

表 17.1　目标文件的布局

起始文件地址	Section 或 Header
0	ELF Header
0x34	.text
0x60	.data
0x98	.bss（此段为空）
0x98	.shstrtab
0xc8	Section Header Table
0x208	.symtab
0x288	.strtab
0x2b0	.rel.text

这个文件不大，我们直接用 hexdump 工具把目标文件的字节全部打印出来看。

```
$ hexdump -C max.o
00000000  7f 45 4c 46 01 01 01 00  00 00 00 00 00 00 00 00
|.ELF............|
00000010  01 00 03 00 01 00 00 00  00 00 00 00 00 00 00 00
|................|
00000020  c8 00 00 00 00 00 00 00  34 00 00 00 00 00 28 00
|........4.....(.|
00000030  08 00 05 00 bf 00 00 00  00 8b 04 bd 00 00 00 00
|................|
00000040  89 c3 83 f8 00 74 10 47  8b 04 bd 00 00 00 00 39
|.....t.G.......9|
00000050  d8 7e ef 89 c3 eb eb b8  01 00 00 00 cd 80 00 00
|.~..............|
00000060  03 00 00 00 43 00 00 00  22 00 00 00 de 00 00 00
|....C..."........|
00000070  2d 00 00 00 4b 00 00 00  36 00 00 00 22 00 00 00
|-...K...6..."...|
00000080  2c 00 00 00 21 00 00 00  16 00 00 00 0b 00 00 00
|,...!...........|
00000090  42 00 00 00 00 00 00 00  00 2e 73 79 6d 74 61 62
|B.........symtab|
000000a0  00 2e 73 74 72 74 61 62  00 2e 73 68 73 74 72 74
|..strtab..shstrt|
```

```
    000000b0  61 62 00 2e 72 65 6c 2e  74 65 78 74 00 2e 64 61
|ab..rel.text..da|
    000000c0  74 61 00 2e 62 73 73 00  00 00 00 00 00 00 00 00
|ta..bss.........|
    000000d0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
|................|
    *
    000000f0  1f 00 00 00 01 00 00 00  06 00 00 00 00 00 00 00
|................|
    00000100  34 00 00 00 2a 00 00 00  00 00 00 00 00 00 00 00
|4...*...........|
    00000110  04 00 00 00 00 00 00 00  1b 00 00 00 09 00 00 00
|................|
    00000120  00 00 00 00 00 00 00 00  b0 02 00 00 10 00 00 00
|................|
    00000130  06 00 00 00 01 00 00 00  04 00 00 00 08 00 00 00
|................|
    00000140  25 00 00 00 01 00 00 00  03 00 00 00 00 00 00 00
|%...............|
    00000150  60 00 00 00 38 00 00 00  00 00 00 00 00 00 00 00
|`...8...........|
    00000160  04 00 00 00 00 00 00 00  2b 00 00 00 08 00 00 00
|........+.......|
    00000170  03 00 00 00 00 00 00 00  98 00 00 00 00 00 00 00
|................|
    00000180  00 00 00 00 00 00 00 00  04 00 00 00 00 00 00 00
|................|
    00000190  11 00 00 00 03 00 00 00  00 00 00 00 00 00 00 00
|................|
    000001a0  98 00 00 00 30 00 00 00  00 00 00 00 00 00 00 00
|....0...........|
    000001b0  01 00 00 00 00 00 00 00  01 00 00 00 02 00 00 00
|................|
    000001c0  00 00 00 00 00 00 00 00  08 02 00 00 80 00 00 00
|................|
    000001d0  07 00 00 00 07 00 00 00  04 00 00 00 10 00 00 00
|................|
    000001e0  09 00 00 00 03 00 00 00  00 00 00 00 00 00 00 00
|................|
    000001f0  88 02 00 00 28 00 00 00  00 00 00 00 00 00 00 00
|....(...........|
    00000200  01 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
|................|
    00000210  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
|................|
    00000220  00 00 00 00 03 00 01 00  00 00 00 00 00 00 00 00
|................|
    00000230  00 00 00 00 03 00 03 00  00 00 00 00 00 00 00 00
|................|
    00000240  00 00 00 00 03 00 04 00  01 00 00 00 00 00 00 00
|................|
    00000250  00 00 00 00 00 00 03 00  0c 00 00 00 0e 00 00 00
|................|
    00000260  00 00 00 00 00 00 01 00  17 00 00 00 23 00 00 00
|............#...|
    00000270  00 00 00 00 00 00 01 00  21 00 00 00 00 00 00 00
|........!.......|
    00000280  00 00 00 00 10 00 01 00  00 64 61 74 61 5f 69 74
|.........data_it|
```

```
    00000290  65 6d 73 00 73 74 61 72  74 5f 6c 6f 6f 70 00 6c
|ems.start_loop.l|
    000002a0  6f 6f 70 5f 65 78 69 74  00 5f 73 74 61 72 74 00
|oop_exit._start.|
    000002b0  08 00 00 00 01 02 00 00  17 00 00 00 01 02 00 00
|................|
    000002c0
```

左边一列是文件地址，中间是每个字节的十六进制表示，右边是把这些字节解释成 ASCII 码所对应的字符。中间有一个*号表示省略的部分全是 0。.data 段对应的是这一块：

```
    ...
    00000060  03 00 00 00 43 00 00 00  22 00 00 00 de 00 00 00
|....C..."....... |
    00000070  2d 00 00 00 4b 00 00 00  36 00 00 00 22 00 00 00
|-...K...6..."...|
    00000080  2c 00 00 00 21 00 00 00  16 00 00 00 0b 00 00 00
|,...!...........|
    00000090  42 00 00 00 00 00 00 00
    ...
```

.data 段将被原封不动地加载到内存中，下一小节会看到.data 段被加载到内存地址 0x080490a0～0x080490d7。

.shstrtab 和.strtab 这两个 Section 中存放的都是 ASCII 码：

```
    ...
                                00   2e   73   79   6d   74   61   62
|B.........symtab|
    000000a0  00 2e 73 74 72 74 61 62  00 2e 73 68 73 74 72 74
|..strtab..shstrt|
    000000b0  61 62 00 2e 72 65 6c 2e  74 65 78 74 00 2e 64 61
|ab..rel.text..da|
    000000c0        74    61    00    2e    62    73    73    00
|ta..bss.........|
    ...
                                00   64   61   74   61   5f   69   74
|.........data_it|
    00000290  65 6d 73 00 73 74 61 72  74 5f 6c 6f 6f 70 00 6c
|ems.start_loop.l|
    000002a0  6f 6f 70 5f 65 78 69 74  00 5f 73 74 61 72 74 00
|oop_exit._start.|
    ...
```

可见.shstrtab 段保存着各 Section 的名字，.strtab 段保存着程序中用到的符号的名字，每个名字都是以 Null 结尾的字符串。

我们知道，C 语言的全局变量如果在代码中没有初始化，就会在程序加载时用 0 初始化。这种数据属于.bss 段，在加载时它和.data 段一样都是可读可写的数据，但是在 ELF 文件中.data 段需要占用一部分空间保存初始值，而.bss 段则不需要。也就是说，.bss 段在文件中只占一个 Section Header 而没有对应的 Section，程序加载时.bss 段占多大内存空间在 Section Header 中描述。在我们这个例子中没有用到.bss 段，在第 18.3 节会看到这样的例子。

我们继续分析 readelf 输出的最后一部分，是从.rel.text 和.symtab 这两个 Section 中读出的信息。

```
...
Relocation section '.rel.text' at offset 0x2b0 contains 2 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
00000008  00000201 R_386_32            00000000   .data
00000017  00000201 R_386_32            00000000   .data

There are no unwind sections in this file.

Symbol table '.symtab' contains 8 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000     0 SECTION LOCAL  DEFAULT    1
     2: 00000000     0 SECTION LOCAL  DEFAULT    3
     3: 00000000     0 SECTION LOCAL  DEFAULT    4
     4: 00000000     0 NOTYPE  LOCAL  DEFAULT    3 data_items
     5: 0000000e     0 NOTYPE  LOCAL  DEFAULT    1 start_loop
     6: 00000023     0 NOTYPE  LOCAL  DEFAULT    1 loop_exit
     7: 00000000     0 NOTYPE  GLOBAL DEFAULT    1 _start

No version information found in this file.
```

.rel.text 告诉链接器指令中的哪些地方需要做重定位，在下一小节详细讨论。

.symtab 是符号表。Ndx 列是每个符号所在的 Section 编号，例如符号 data_items 在第 3 个 Section 里（也就是.data 段），各 Section 的编号见 Section Header Table。Value 列是每个符号所代表的地址，在目标文件中，符号地址都是相对于该符号所在 Section 的相对地址，比如 data_items 位于.data 段的开头，所以地址是 0，_start 位于.text 段的开头，所以地址也是 0，但是 start_loop 和 loop_exit 相对于.text 段的地址就不是 0 了。从 Bind 这一列可以看出_start 这个符号是 GLOBAL 的，而其他符号是 LOCAL 的，在汇编程序中用.globl 指示声明过的符号会成为全局符号，否则成为局部符号。

现在剩下.text 段没有分析，objdump 工具可以把程序中的机器指令反汇编（Disassemble），那么反汇编的结果是否跟原来写的汇编代码一模一样呢？我们对比分析一下。

```
$ objdump -d max.o

max.o:     file format elf32-i386


Disassembly of section .text:

00000000 <_start>:
   0:   bf 00 00 00 00          mov    $0x0,%edi
   5:   8b 04 bd 00 00 00 00    mov    0x0(,%edi,4),%eax
   c:   89 c3                   mov    %eax,%ebx

0000000e <start_loop>:
   e:   83 f8 00                cmp    $0x0,%eax
  11:   74 10                   je     23 <loop_exit>
```

```
  13:   47                      inc    %edi
  14:   8b 04 bd 00 00 00 00    mov    0x0(,%edi,4),%eax
  1b:   39 d8                   cmp    %ebx,%eax
  1d:   7e ef                   jle    e <start_loop>
  1f:   89 c3                   mov    %eax,%ebx
  21:   eb eb                   jmp    e <start_loop>

00000023 <loop_exit>:
  23:   b8 01 00 00 00          mov    $0x1,%eax
  28:   cd 80                   int    $0x80
```

左边是机器指令的字节，右边是反汇编结果。显然，所有的符号都被替换成地址了，比如 je 23，注意没有加$的数表示内存地址，而不表示立即数。这条指令后面的<loop_exit>并不是指令的一部分，而是反汇编器从.symtab 和.strtab 中查到的符号名称，写在后面是为了有更好的可读性。目前所有指令中用到的符号地址都是相对地址，下一步链接器要修改这些指令，把其中的地址都改成加载时的内存地址，这些指令才能正确执行。

17.5.2　可执行文件

现在我们按上一节的步骤分析可执行文件 max，看看链接器都做了什么改动。

```
$ readelf -a max
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x8048074
  Start of program headers:          52 (bytes into file)
  Start of section headers:          256 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         2
  Size of section headers:           40 (bytes)
  Number of section headers:         6
  Section header string table index: 3

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        08048074 000074 00002a 00  AX  0   0  4
  [ 2] .data             PROGBITS        080490a0 0000a0 000038 00  WA  0   0  4
  [ 3] .shstrtab         STRTAB          00000000 0000d8 000027 00      0   0  1
  [ 4] .symtab           SYMTAB          00000000 0001f0 0000a0 10      5   6  4
  [ 5] .strtab           STRTAB          00000000 000290 000040 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
```

```
  O (extra OS processing required) o (OS specific), p (processor specific)

There are no section groups in this file.

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x000000 0x08048000 0x08048000 0x0009e 0x0009e R E 0x1000
  LOAD           0x0000a0 0x080490a0 0x080490a0 0x00038 0x00038 RW  0x1000

 Section to Segment mapping:
  Segment Sections...
   00     .text
   01     .data

There is no dynamic section in this file.

There are no relocations in this file.

There are no unwind sections in this file.

Symbol table '.symtab' contains 10 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 08048074     0 SECTION LOCAL  DEFAULT    1
     2: 080490a0     0 SECTION LOCAL  DEFAULT    2
     3: 080490a0     0 NOTYPE  LOCAL  DEFAULT    2 data_items
     4: 08048082     0 NOTYPE  LOCAL  DEFAULT    1 start_loop
     5: 08048097     0 NOTYPE  LOCAL  DEFAULT    1 loop_exit
     6: 08048074     0 NOTYPE  GLOBAL DEFAULT    1 _start
     7: 080490d8     0 NOTYPE  GLOBAL DEFAULT  ABS __bss_start
     8: 080490d8     0 NOTYPE  GLOBAL DEFAULT  ABS _edata
     9: 080490d8     0 NOTYPE  GLOBAL DEFAULT  ABS _end

No version information found in this file.
```

在 ELF Header 中，Type 改成了 EXEC，由目标文件变成可执行文件了，Entry point address 改成了 0x8048074（这是_start 符号的地址），还可以看出，多了两个 Program Header，少了两个 Section Header。

在 Section Header Table 中，.text 和.data 段的加载地址分别改成了 0x08048074 和 0x080490a0。.bss 段没有用到，所以被删掉了。.rel.text 段就是用于链接过程的，做完链接就没用了，所以也删掉了。

多出来的 Program Header Table 描述了两个 Segment 的信息。.text 段和前面的 ELF Header、Program Header Table 一起组成一个 Segment（FileSiz 指出总长度是 0x9e），.data 段组成另一个 Segment（总长度是 0x38），以后我们把这两个 Segment 分别叫做 Text Segment 和 Data Segment。VirtAddr 列指出 Text Segment 加载到虚拟地址 0x08048000（注意在 x86 平台上后面的 PhysAddr 列是没有意义的，并不代表实际的物理地址），Data Segment 加载到地址 0x080490a0。Flg 列指出 Text Segment 的访问权限是可读可执行，Data Segment 的访问权限是可读可写。最后一列 Align 的值 0x1000（4K）是 x86 平台的内存页面大小。在加载时文件也要按内存页面大小分成若干页，文件中的一页对应内存中的一页，对应关系如图 17.2 所示。

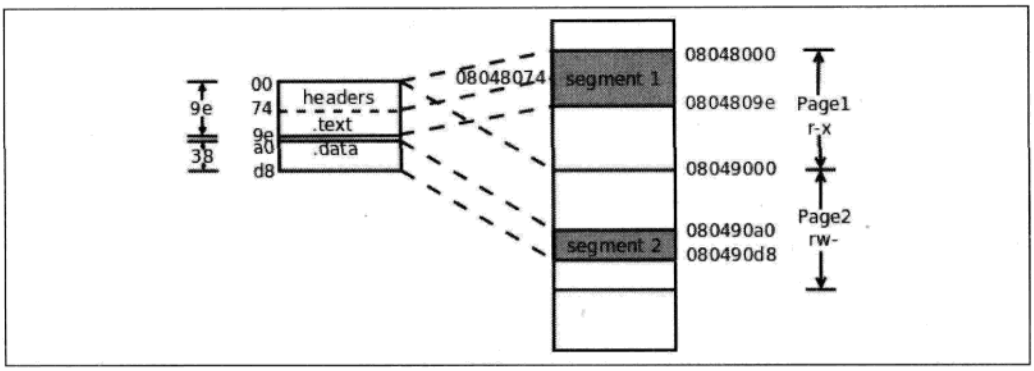


图 17.2　文件和加载地址的对应关系

这个可执行文件很小，总共也不超过一页大小，但是两个 Segment 必须加载到内存中两个不同的页面，因为 MMU 的权限保护机制是以页为单位的，一个页面只能设置一种权限。所以 Text Segment 加载到虚拟地址 0x08048000～0x08048fff 的内存页面，而 Data Segment 加载到虚拟地址 0x08049000～0x08049fff 的内存页面。此外，为了简化链接器和加载器的实现，还规定每个 Segment 在文件的页面中偏移多少加载到内存页面也要偏移多少。Text Segment 在文件的第 0 个页面开头，加载到内存页面也是从首地址 0x08048000 开始，由于 Text Segment 包含了文件名和.text 段，所以.text 段的加载地址是 0x08048074，_start 符号位于.text 段的开头，所以_start 符号的地址也是 0x08048074，从符号表中可以验证这一点。Data Segment 在文件的第 0 个页面中的偏移是 0xa0，在内存页面中的偏移也是 0xa0，所以从 0x080490a0 开始。

原来目标文件符号表中的 Value 都是相对地址，现在都改成绝对地址了。此外还多了三个符号__bss_start、_edata 和_end，这些符号在链接脚本中定义，被链接器添加到可执行文件中，链接脚本在第 19.1 节介绍。

再看一下反汇编的结果：

```
$ objdump -d max

max:     file format elf32-i386


Disassembly of section .text:

08048074 <_start>:
 8048074:   bf 00 00 00 00          mov  $0x0,%edi
 8048079:   8b 04 bd a0 90 04 08    mov  0x80490a0(,%edi,4),%eax
 8048080:   89 c3                   mov  %eax,%ebx

08048082 <start_loop>:
 8048082:   83 f8 00                cmp  $0x0,%eax
 8048085:   74 10                   je   8048097 <loop_exit>
 8048087:   47                      inc  %edi
 8048088:   8b 04 bd a0 90 04 08    mov  0x80490a0(,%edi,4),%eax
 804808f:   39 d8                   cmp  %ebx,%eax
 8048091:   7e ef                   jle  8048082 <start_loop>
 8048093:   89 c3                   mov  %eax,%ebx
```

```
 8048095:   eb eb                    jmp  8048082 <start_loop>

08048097 <loop_exit>:
 8048097:   b8 01 00 00 00           mov  $0x1,%eax
 804809c:   cd 80                    int  $0x80
```

指令中的相对地址都改成绝对地址了。我们仔细检查一下改了哪些地方。首先看跳转指令，原来目标文件的指令是这样：

```
...
  11:   74 10               je    23 <loop_exit>
...
  1d:   7e ef               jle   e <start_loop>
...
  21:   eb eb               jmp   e <start_loop>
...
```

现在改成了这样：

```
...
 8048085:   74 10           je    8048097 <loop_exit>
...
 8048091:   7e ef           jle   8048082 <start_loop>
...
 8048095:   eb eb           jmp   8048082 <start_loop>
...
```

改了吗？其实只是反汇编的结果不同了，指令的机器码根本没变。为什么不用改指令就能跳转到新的地址呢？因为跳转指令中指定的是相对于当前指令向前或向后跳多少字节，而不是指定一个完整的内存地址，内存地址有 32 位，这些跳转指令只有 16 位，显然也不可能指定一个完整的内存地址，这称为相对跳转。这种相对跳转指令只有 16 位，只能在当前指令前后的一个小范围内跳转，不可能跳得太远，也有的跳转指令指定一个完整的内存地址，可以跳到任何地方，称为绝对跳转。

再看内存访问指令，原来目标文件的指令是这样：

```
...
   5:   8b 04 bd 00 00 00 00    mov   0x0(,%edi,4),%eax
...
  14:   8b 04 bd 00 00 00 00    mov   0x0(,%edi,4),%eax
...
```

现在改成了这样：

```
...
 8048079:   8b 04 bd a0 90 04 08    mov
0x80490a0(,%edi,4),%eax
...
 8048088:   8b 04 bd a0 90 04 08    mov
0x80490a0(,%edi,4),%eax
...
```

指令中的地址原本是 0x00000000，现在改成了 0x080490a0（注意是小端字节序）。那么链接器怎么知道要改这两处呢？是根据目标文件中的.rel.text 段提供的重定

位信息来改的：

```
...
Relocation section '.rel.text' at offset 0x2b0 contains 2 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
00000008  00000201 R_386_32        00000000   .data
00000017  00000201 R_386_32        00000000   .data
...
```

第一列 Offset 的值就是.text 段需要改的地方，在.text 段中的相对地址是 8 和 0x17，正是这两条指令中 00 00 00 00 的位置。

下面试试用 strip 命令去除可执行文件中的符号信息，这样可以有效减小文件的尺寸而不影响运行：

```
$ strip max
$ readelf -a max
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x8048074
  Start of program headers:          52 (bytes into file)
  Start of section headers:          240 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         2
  Size of section headers:           40 (bytes)
  Number of section headers:         4
  Section header string table index: 3

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg
Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00
0   0  0
  [ 1] .text             PROGBITS        08048074 000074 00002a 00  AX
0   0  4
  [ 2] .data             PROGBITS        080490a0 0000a0 000038 00  WA
0   0  4
  [ 3] .shstrtab         STRTAB          00000000 0000d8 000017 00
0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor
specific)

There are no section groups in this file.

Program Headers:
```

```
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x000000 0x08048000 0x08048000 0x0009e 0x0009e R E 0x1000
  LOAD           0x0000a0 0x080490a0 0x080490a0 0x00038 0x00038 RW  0x1000

 Section to Segment mapping:
  Segment Sections...
   00     .text
   01     .data

There is no dynamic section in this file.

There are no relocations in this file.

There are no unwind sections in this file.

No version information found in this file.
```

注意不要对目标文件和共享库使用 strip 命令，因为链接器需要利用目标文件和共享库中的符号信息来做链接。

第 18 章

汇编与 C 之间的关系

上一章我们学习了汇编的一些基础知识，本章我们进一步研究 C 程序编译之后的汇编是什么样的，C 语言的各种语法分别对应什么样的指令，从而更深入地理解 C 语言。gcc 还提供了一种扩展语法可以在 C 程序中内嵌汇编指令，这在内核代码中很常见，本章也会简要介绍这种用法。

18.1 函数调用

我们用下面的代码来研究函数的调用过程。

例 18.1 研究函数的调用过程

```
int bar(int c, int d)
{
        int e = c + d;
        return e;
}

int foo(int a, int b)
{
        return bar(a, b);
}

int main(void)
{
        foo(2, 3);
        return 0;
}
```

如果在编译时加上-g 选项（在第 10 章讲过-g 选项），那么用 objdump 反汇编时可以把 C 代码和汇编代码穿插起来显示，这样 C 代码和汇编代码的对应关系看得更清楚。反汇编的结果很长，以下只列出我们关心的部分。

```
$ gcc main.c -g
$ objdump -dS a.out
...
080483b4 <bar>:
int bar(int c, int d)
{
 80483b4:   55                          push   %ebp
 80483b5:   89 e5                       mov    %esp,%ebp
 80483b7:   83 ec 10                    sub    $0x10,%esp
```

```
    int e = c + d;
 80483ba:   8b 45 0c                mov    0xc(%ebp),%eax
 80483bd:   8b 55 08                mov    0x8(%ebp),%edx
 80483c0:   8d 04 02                lea    (%edx,%eax,1),%eax
 80483c3:   89 45 fc                mov    %eax,-0x4(%ebp)
    return e;
 80483c6:   8b 45 fc                mov    -0x4(%ebp),%eax
}
 80483c9:   c9                      leave
 80483ca:   c3                      ret

080483cb <foo>:

int foo(int a, int b)
{
 80483cb:   55                      push   %ebp
 80483cc:   89 e5                   mov    %esp,%ebp
 80483ce:   83 ec 08                sub    $0x8,%esp
    return bar(a, b);
 80483d1:   8b 45 0c                mov    0xc(%ebp),%eax
 80483d4:   89 44 24 04             mov    %eax,0x4(%esp)
 80483d8:   8b 45 08                mov    0x8(%ebp),%eax
 80483db:   89 04 24                mov    %eax,(%esp)
 80483de:   e8 d1 ff ff ff          call   80483b4 <bar>
}
 80483e3:   c9                      leave
 80483e4:   c3                      ret

080483e5 <main>:

int main(void)
{
 80483e5:   55                      push   %ebp
 80483e6:   89 e5                   mov    %esp,%ebp
 80483e8:   83 ec 08                sub    $0x8,%esp
    foo(2, 3);
 80483eb:   c7 44 24 04 03 00 00    movl   $0x3,0x4(%esp)
 80483f2:   00
 80483f3:   c7 04 24 02 00 00 00    movl   $0x2,(%esp)
 80483fa:   e8 cc ff ff ff          call   80483cb <foo>
    return 0;
 80483ff:   b8 00 00 00 00          mov    $0x0,%eax
}
 8048404:   c9                      leave
 8048405:   c3                      ret
...
```

要查看编译后的汇编代码，其实还有一种办法是 gcc -S main.c，这样只生成汇编代码 main.s，而不生成二进制的目标文件。

整个程序的执行过程是 main 调用 foo，foo 调用 bar，我们用 gdb 跟踪程序的执行，直到 bar 函数中的 int e = c + d;执行完毕准备返回时，才在 gdb 中打印函数栈帧。

```
(gdb) start
...
main () at main.c:14
14              foo(2, 3);
(gdb) s
```

```
foo (a=2, b=3) at main.c:9
9               return bar(a, b);
(gdb) s
bar (c=2, d=3) at main.c:3
3               int e = c + d;
(gdb) disassemble
Dump of assembler code for function bar:
   0x080483b4 <+0>:     push   %ebp
   0x080483b5 <+1>:     mov    %esp,%ebp
   0x080483b7 <+3>:     sub    $0x10,%esp
=> 0x080483ba <+6>:     mov    0xc(%ebp),%eax
   0x080483bd <+9>:     mov    0x8(%ebp),%edx
   0x080483c0 <+12>:    lea    (%edx,%eax,1),%eax
   0x080483c3 <+15>:    mov    %eax,-0x4(%ebp)
   0x080483c6 <+18>:    mov    -0x4(%ebp),%eax
   0x080483c9 <+21>:    leave
   0x080483ca <+22>:    ret
End of assembler dump.
(gdb) si
0x080483bd      3               int e = c + d;
(gdb) si
0x080483c0      3               int e = c + d;
(gdb) si
0x080483c3      3               int e = c + d;
(gdb) si
4               return e;
(gdb) si
5   }
(gdb) bt
#0  bar (c=2, d=3) at main.c:5
#1  0x080483e3 in foo (a=2, b=3) at main.c:9
#2  0x080483ff in main () at main.c:14
(gdb) disassemble
Dump of assembler code for function bar:
   0x080483b4 <+0>:     push   %ebp
   0x080483b5 <+1>:     mov    %esp,%ebp
   0x080483b7 <+3>:     sub    $0x10,%esp
   0x080483ba <+6>:     mov    0xc(%ebp),%eax
   0x080483bd <+9>:     mov    0x8(%ebp),%edx
   0x080483c0 <+12>:    lea    (%edx,%eax,1),%eax
   0x080483c3 <+15>:    mov    %eax,-0x4(%ebp)
   0x080483c6 <+18>:    mov    -0x4(%ebp),%eax
=> 0x080483c9 <+21>:    leave
   0x080483ca <+22>:    ret
End of assembler dump.
(gdb) info registers
eax            0x5        5
ecx            0x5386962  87583074
edx            0x2        2
ebx            0x283ff4   2637812
esp            0xbffff358 0xbffff358
ebp            0xbffff368 0xbffff368
esi            0x0        0
edi            0x0        0
eip            0x80483c9  0x80483c9 <bar+21>
eflags         0x282      [ SF IF ]
cs             0x73       115
ss             0x7b       123
ds             0x7b       123
es             0x7b       123
```

```
fs             0x0        0
gs             0x33       51
(gdb) x/12x $esp
0xbffff358: 0xbffff388  0x08048439  0x00284324  0x00000005
0xbffff368: 0xbffff378  0x080483e3  0x00000002  0x00000003
0xbffff378: 0xbffff388  0x080483ff  0x00000002  0x00000003
```

这里又用到几个新的 gdb 命令：

- disassemble 可以反汇编当前函数或者指定的函数，单独用 disassemble 命令是反汇编当前函数，如果 disassemble 命令后面跟函数名或地址则反汇编指定的函数或地址。
- 以前我们讲过 step 命令可以一行代码一行代码地单步调试，而这里用到的 si 命令可以一条指令一条指令地单步调试。
- info registers 可以显示所有寄存器的当前值。
- 在 gdb 中表示寄存器名时前面要加个$，例如 p $esp 可以打印 esp 寄存器的值。在上例中用 info registers 命令看到 esp 寄存器的值是 0xbffff358，所以用 x/12x $esp 命令可以查看内存中从 0xbffff358 地址开始的 12 个 32 位数。

在执行程序时，操作系统为进程分配一块栈空间来保存函数栈帧，esp 寄存器总是指向栈顶，在 x86 平台上这个栈是从高地址向低地址增长的，我们知道每次调用一个函数都要分配一个栈帧来保存参数和局部变量，现在我们详细分析这些数据在栈空间的布局，根据 gdb 的输出结果如图 18.1 所示[①]：

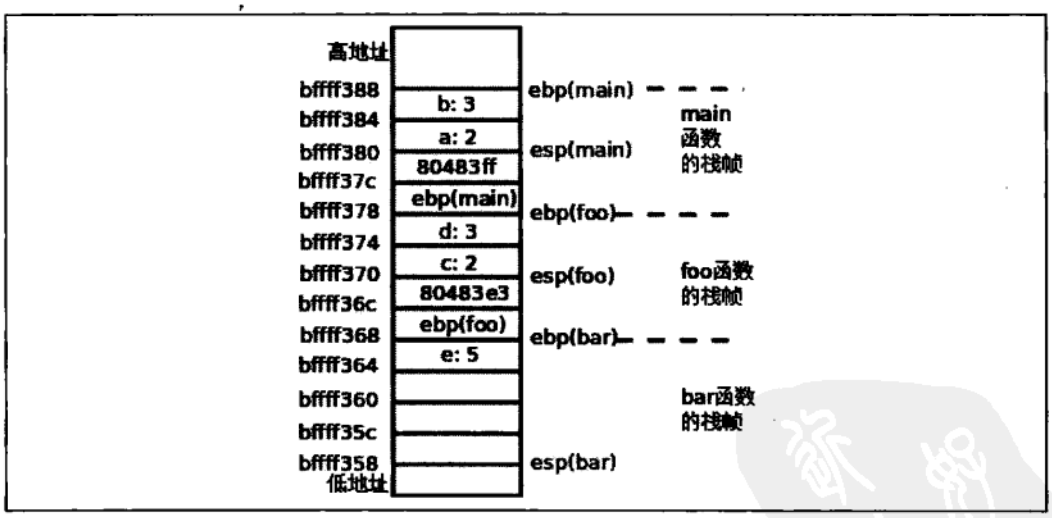


图 18.1 函数栈帧

图 18.1 中每个小方格表示 4 个字节的内存单元，例如 b: 3 这个小方格占的内存地址是 0xbffff384～0xbffff387，我把地址写在每个小方格的下边界线上，是为了强调该地址是内存单元的起始地址。我们从 main 函数的这里开始看起：

① 为了安全性，Linux 内核为每个新进程指定的栈空间起始地址是随机的，所以每次运行这个程序得到的地址都不一样，但通常都是 0xbf??????这样的一个地址。

```
    foo(2, 3);
 80483eb:   c7 44 24 04 03 00 00    movl   $0x3,0x4(%esp)
 80483f2:   00
 80483f3:   c7 04 24 02 00 00 00    movl   $0x2,(%esp)
 80483fa:   e8 cc ff ff ff          call   80483cb <foo>
    return 0;
 80483ff:   b8 00 00 00 00          mov    $0x0,%eax
```

一开始 esp 寄存器的值是 0xbffff380，在图 18.1 中标注为 esp(main)。要调用函数 foo 先要把参数准备好，第二个参数保存在 esp+4 指向的内存位置，第一个参数保存在 esp 指向的内存位置，可见参数是从右向左依次压栈的。然后执行 call 指令，这个指令有两个作用：

1．foo 函数调用完之后要返回到 call 的下一条指令继续执行，所以把 call 的下一条指令的地址 0x80483ff 压栈，同时把 esp 的值减 4，esp 的值现在是 0xbffff37c。

2．修改程序计数器 eip，跳转到 foo 函数的开头执行。

现在看 foo 函数的汇编代码：

```
int foo(int a, int b)
{
 80483cb:   55                      push   %ebp
 80483cc:   89 e5                   mov    %esp,%ebp
 80483ce:   83 ec 08                sub    $0x8,%esp
```

push %ebp 指令把 ebp 寄存器的值压栈，同时把 esp 的值减 4。esp 的值现在是 0xbffff378，下一条指令把这个值传送给 ebp 寄存器。这两条指令合起来是把原来 ebp 的值保存在栈上，然后又给 ebp 赋了新值，新值在图 18.1 中标注为 ebp(foo)。然后 esp 所指向的地址向下移动 8 个字节（在图 18.1 中标注为 esp(foo)），为即将压栈的参数 d 和 c 留出空间。

在每个函数的栈帧中，ebp 指向栈底，而 esp 指向栈顶，在函数执行过程中 esp 随着压栈和出栈操作随时变化，而 ebp 是不动的，函数的参数和局部变量都是通过 ebp 的值加上一个偏移量来访问，例如 foo 函数的参数 b 和 a 分别通过 ebp+12 和 ebp+8 来访问。所以 foo 函数接下来的指令把参数 b 和 a 取出来，作为参数 d 和 c 再次压栈，为调用 bar 函数做准备，然后把返回地址 0x80483e3 压栈，调用 bar 函数：

```
    return bar(a, b);
 80483d1:   8b 45 0c                mov    0xc(%ebp),%eax
 80483d4:   89 44 24 04             mov    %eax,0x4(%esp)
 80483d8:   8b 45 08                mov    0x8(%ebp),%eax
 80483db:   89 04 24                mov    %eax,(%esp)
 80483de:   e8 d1 ff ff ff             call   80483b4 <bar>
}
 80483e3:   c9                      leave
 80483e4:   c3                      ret
```

现在看 bar 函数的指令：

```
int bar(int c, int d)
{
```

```
80483b4:   55                  push   %ebp
80483b5:   89 e5               mov    %esp,%ebp
80483b7:   83 ec 10            sub    $0x10,%esp
   int e = c + d;
80483ba:   8b 45 0c            mov    0xc(%ebp),%eax
80483bd:   8b 55 08            mov    0x8(%ebp),%edx
80483c0:   8d 04 02            lea    (%edx,%eax,1),%eax
80483c3:   89 45 fc            mov    %eax,-0x4(%ebp)
```

这次又把 foo 函数的 ebp 压栈保存，然后给 ebp 赋了新值，指向 bar 函数栈帧的栈底（在图 18.1 中标注为 ebp(bar)），然后 esp 所指向的地址向下移动 16 个字节（在图 18.1 中标注为 esp(bar)），为局部变量 e 留出空间。

通过 ebp+12 和 ebp+8 分别可以访问参数 d 和 c，通过 ebp-4 可以访问局部变量 e。所以后面几条指令的意思是把参数 d 和 c 取出来传送到寄存器 eax 和 edx 中，然后用 lea 指令做加法，计算结果保存在 eax 寄存器中，再把 eax 的值存回局部变量 e 的内存单元。lea 指令根据第一个操作数的寻址方式计算出所代表的地址，但并不通过这个地址访问内存，而是直接把这个地址传给第二个操作数，我们知道 x86 的内存寻址方式涉及加法和乘法，lea 指令只是利用寻址电路做加法和乘法，而不是真的寻址，lea (%edx,%eax,1),%eax 这条指令的意思是 eax = edx + eax * 1。

我们知道在 gdb 中可以用 bt 命令和 frame 命令查看每层栈帧上的参数和局部变量，现在可以解释它的工作原理了：如果我当前在 bar 函数中，我可以通过 ebp 找到 bar 函数的参数和局部变量，也可以找到 foo 函数的 ebp 保存在栈上的值，有了 foo 函数的 ebp，又可以找到它的参数和局部变量，也可以找到 main 函数的 ebp 保存在栈上的值，因此各层函数栈帧通过保存在栈上的 ebp 的值串起来了。

现在看 bar 函数的返回指令：

```
   return e;
80483c6:   8b 45 fc            mov    -0x4(%ebp),%eax
}
80483c9:   c9                  leave
80483ca:   c3                  ret
```

bar 函数有一个 int 型的返回值，这个返回值是通过 eax 寄存器传递的，所以首先把 e 的值读到 eax 寄存器中。然后执行 leave 指令，这个指令是函数开头的 push %ebp 和 mov %esp,%ebp 的逆操作：

1．把 ebp 的值赋给 esp，现在 esp 的值是 0xbffff368。

2．现在 esp 所指向的栈顶保存着 foo 函数栈帧的 ebp，把这个值恢复给 ebp，同时 esp 增加 4，esp 的值变成 0xbffff36c。

最后是 ret 指令，它是 call 指令的逆操作：

1．现在 esp 所指向的栈顶保存着返回地址 0x80483e3，把这个值恢复给 eip，同时 esp 增加 4，esp 的值变成 0xbffff370。

2．由于修改了程序计数器 eip，程序跳转到返回地址处继续执行。

地址 0x80483e3 处是 foo 函数的返回指令：

```
80483e3:    c9                      leave
80483e4:    c3                      ret
```

这两条指令重复同样的过程，又返回到了 main 函数。

现在思考这样一个问题：如果这时 main 函数又调用另外一个函数，先前调用 foo 函数时压栈的参数 b 和 a 并没有出栈，再调用另外一个函数又要将新的参数压栈，那么随着 main 调用的函数越来越多，堆栈岂不是要一直增长下去？当然不是了，读者可以写一段小程序自己反汇编研究一下编译器是怎么处理的。

注意函数调用和返回过程中的这些规则：

1．参数压栈传递，并且是从右到左依次压栈，传参所使用的栈空间由调用者分配和释放。

2．ebp 总是指向当前栈帧的栈底。

3．返回值通过 eax 寄存器传递。

这些规则并不是体系结构所强加的，ebp 寄存器并不是必须这么用，函数的参数和返回值也不是必须这么传，不同的操作系统和编译器可以规定不同的方式来实现 C 代码中的函数调用，每种实现方式称为一种 Calling Convention。本章介绍的规则是 gcc 默认的 Calling Convention，称为 cdecl，其它常见的 Calling Convention 可参考 http://en.wikipedia.org/wiki/X86_calling_conventions。

习题

1．在第 3.2 节讲过，Old Style C 风格的函数声明可以不指定参数个数和类型，这样编译器不会对函数调用做检查，那么如果调用时的参数类型不对或者参数个数不对会怎么样呢？比如把本节的例子改成这样：

```
int foo();
int bar();

int main(void)
{
        foo(2, 3, 4);
        return 0;
}

int foo(int a, int b)
{
        return bar(a);
}

int bar(int c, int d)
{
        int e = c + d;
        return e;
}
```

main 函数调用 foo 时多传了一个参数，那么参数 a 和 b 分别取什么值？多的参数怎么办？foo 调用 bar 时少传了一个参数，那么参数 d 的值从哪里取得？请读者利用反汇编和 gdb 自己分析一下。我们再看一个参数类型不符的例子：

```
#include <stdio.h>

int main(void)
{
        void foo();
        char c = 60;
        foo(c);
        return 0;
}

void foo(double d)
{
        printf("%f\n", d);
}
```

打印结果是多少？如果把声明 void foo();改成 void foo(double);，打印结果又是多少？

18.2 main 函数、启动例程和退出状态

为什么汇编程序的入口是_start 而 C 程序的入口是 main 函数呢？要弄清楚这个问题，首先要理解 gcc 的编译步骤。继续用上一节的代码做实验，以前我们常用 gcc main.c -o main 命令编译一个程序，其实也可以分三步做，第一步生成汇编代码，第二步生成目标文件，第三步生成可执行文件：

```
$ gcc -S main.c
$ gcc -c main.s
$ gcc main.o
```

-S 选项生成汇编代码，-c 选项生成目标文件，此外在第 8.2 节还讲过-E 选项只做预处理而不编译，如果这些选项都不加，则 gcc 执行完整的编译步骤，直到最后链接生成可执行文件为止，如图 18.2 所示。

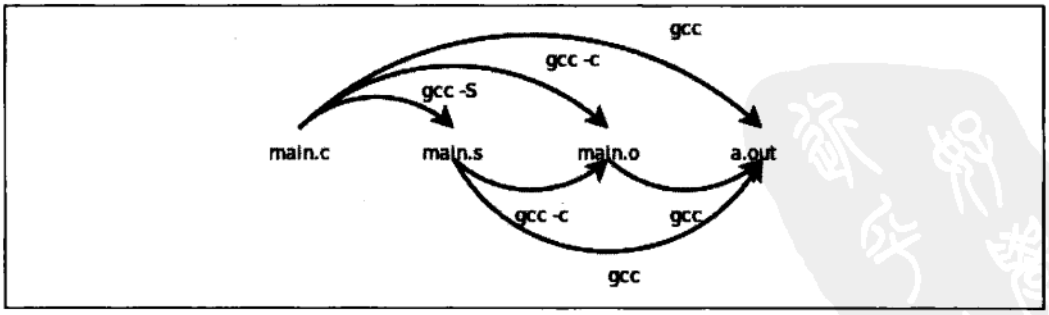


图 18.2 gcc 命令的选项

这些选项都可以和-o 搭配使用，给输出文件重新命名而不使用 gcc 默认的输出文件名（xxx.s、xxx.o 和 a.out），例如 gcc main.o -o main 将 main.o 链接生成可执行文件 main。

即使用 gcc main.c -o main 一步完成编译，gcc 内部也还是要分三步来做，用-v 选项可以了解详细的编译过程：

```
$ gcc -v main.c -o main
Using built-in specs.
Target: i486-linux-gnu
...
 /usr/lib/gcc/i486-linux-gnu/4.4.3/cc1  -quiet  -v  main.c  -D_
FORTIFY_SOURCE=2 -quiet -dumpbase main.c -mtune=generic -march=
i486 -auxbase main -version -fstack-protector -o /tmp/ccMTcipT.s
...
 as -V -Qy -o /tmp/ccY0lGoS.o /tmp/ccMTcipT.s
...
 /usr/lib/gcc/i486-linux-gnu/4.4.3/collect2 --build-id --eh-frame-
hdr -m elf_i386 --hash-style=both -dynamic-linker /lib/ld-linux.so.2
-o  main  -z  relro  /usr/lib/gcc/i486-linux-gnu/4.4.3/../../../../
lib/crt1.o /usr/lib/gcc/i486-linux-gnu/4.4.3/../../../../lib/crti.o/
usr/lib/gcc/i486-linux-gnu/4.4.3/crtbegin.o-L/usr/lib/gcc/i486-linux-
gnu/4.4.3-L/usr/lib/gcc/i486-linux-gnu/4.4.3-L/usr/lib/gcc/i486-linux-
gnu/4.4.3/../../../../lib  -L/lib/../lib  -L/usr/lib/../lib  -L/usr/
lib/gcc/i486-linux-gnu/4.4.3/../../.. -L/usr/lib/i486-linux-gnu /tmp/
ccY0lGoS.o  -lgcc  --as-needed  -lgcc_s  --no-as-needed  -lc  -lgcc
--as-needed -lgcc_s --no-as-needed /usr/lib/gcc/i486-linux-gnu/4.4.3/
crtend.o /usr/lib/gcc/i486-linux-gnu/4.4.3/../../../../lib/crtn.o
```

gcc 只是一个外壳而不是真正的编译器，真正的 C 编译器是/usr/lib/gcc/i486-linux-gnu/4.4.3/cc1，gcc 调动 C 编译器、汇编器和链接器完成 C 代码的编译链接工作。/usr/lib/gcc/i486-linux-gnu/4.4.3/collect2 是链接器 ld 的外壳，它调动 ld 完成链接。具体步骤如下：

1．main.c 被 cc1 编译成汇编程序/tmp/ccMTcipT.s。

2．这个汇编程序被 as 汇编成目标文件/tmp/ccY0lGoS.o。

3．这个目标文件连同另外几个目标文件（crt1.o、crti.o、crtbegin.o、crtend.o、crtn.o）一起链接成可执行文件 main。在链接过程中还用-l 选项指定了一些库文件，有 libc、libgcc、libgcc_s，其中有些库是共享库，需要动态链接，所以用-dynamic-linker 选项指定动态链接器是/lib/ld-linux.so.2。这些链接选项到下一章再详细解释，目前我们只要理解可执行文件 main 是由 main.c 生成的目标文件和编译器提供的另外几个目标文件链接在一起生成的就可以了。

现在看看编译器提供的目标文件里都有什么，我们只看符号表，可以用 readelf 命令的-s 选项，也可以用 nm 命令。我们重点分析 crt1.o 中的符号。

```
$ nm /usr/lib/crt1.o
00000000 R _IO_stdin_used
00000000 D __data_start
         U __libc_csu_fini
         U __libc_csu_init
         U __libc_start_main
00000000 R _fp_hw
00000000 T _start
00000000 W data_start
         U main
```

符号表的每一行由地址、符号类型和符号名组成，目标文件中的地址是待定的，所以是00000000，符号类型用一个字母表示，大写字母是全局符号，小写字母是局部符号，具体每种类型的含义请参考nm(1)。U main这一行表示main这个符号在crt1.o中引用了，但是没有定义（U表示Undefined），因此需要别的目标文件提供一个定义并且和crt1.o链接在一起。T _start这一行表示_start这个符号在crt1.o中提供了定义，这个符号的类型是代码（T表示Text）。

C程序的入口点其实是crt1.o提供的_start，它首先做一些初始化工作（以下称为启动例程，Startup Routine），然后调用我们写的main函数。所以，以前我们说main函数是程序的入口点其实不准确，_start才是真正的入口点，而main函数是被_start调用的。下面我们反汇编查看_start的定义：

```
$ objdump -d /usr/lib/crt1.o

/usr/lib/crt1.o:     file format elf32-i386


Disassembly of section .text:

00000000 <_start>:
   0:   31 ed                   xor    %ebp,%ebp
   2:   5e                      pop    %esi
   3:   89 e1                   mov    %esp,%ecx
   5:   83 e4 f0                and    $0xfffffff0,%esp
   8:   50                      push   %eax
   9:   54                      push   %esp
   a:   52                      push   %edx
   b:   68 00 00 00 00          push   $0x0
  10:   68 00 00 00 00          push   $0x0
  15:   51                      push   %ecx
  16:   56                      push   %esi
  17:   68 00 00 00 00          push   $0x0
  1c:   e8 fc ff ff ff          call   1d <_start+0x1d>
  21:   f4                      hlt
  22:   90                      nop
  23:   90                      nop
```

call指令前面的那条push $0x0指令其实想把main这个符号所代表的地址压栈，但不知道这个地址是多少，因为这个符号在另一个目标文件中定义，到链接时才能确定其地址，所以在指令中暂时写成0x0。

现在我们把main.c编译成目标文件main.o，然后和编译器提供的目标文件链接，对生成的可执行文件main做反汇编分析：

```
$ gcc -c main.c
$ gcc main.o -o main
$ objdump -d main
...
Disassembly of section .text:

08048300 <_start>:
 8048300:   31 ed             xor    %ebp,%ebp
 8048302:   5e                pop    %esi
 8048303:   89 e1             mov    %esp,%ecx
```

```
 8048305:   83 e4 f0          and    $0xfffffff0,%esp
 8048308:   50                push   %eax
 8048309:   54                push   %esp
 804830a:   52                push   %edx
 804830b:   68 10 84 04 08    push   $0x8048410
 8048310:   68 20 84 04 08    push   $0x8048420
 8048315:   51                push   %ecx
 8048316:   56                push   %esi
 8048317:   68 e5 83 04 08    push   $0x80483e5
 804831c:   e8 c7 ff ff ff    call   80482e8 <__libc_start_main@plt>
 8048321:   f4                hlt
...
080483b4 <bar>:
 80483b4:   55                push   %ebp
 80483b5:   89 e5             mov    %esp,%ebp
 80483b7:   83 ec 10          sub    $0x10,%esp
...
080483cb <foo>:
 80483cb:   55                push   %ebp
 80483cc:   89 e5             mov    %esp,%ebp
 80483ce:   83 ec 08          sub    $0x8,%esp
...
080483e5 <main>:
 80483e5:   55                push   %ebp
 80483e6:   89 e5             mov    %esp,%ebp
 80483e8:   83 ec 08          sub    $0x8,%esp
...
```

main.c 中除了 main 函数还定义了 foo、bar 两个函数，链接完成后，crt1.o 中定义的符号_start 和 main.o 中定义的符号 bar、foo、main 都合并到可执行文件的.text 段中。符号 main 的地址是 0x080483e5，因此_start 中的 push $0x0 指令被链接器改成了 push $0x80483e5。一个目标文件中引用了某个符号，链接器在另一个目标文件中找到这个符号的定义并确定它的地址，这个过程叫做符号解析（Symbol Resolution）。符号解析和重定位都是通过修改指令中的地址实现的，链接器也是一种编辑器，vi 和 emacs 编辑的是源文件，而链接器编辑的是目标文件，所以链接器叫做 Link Editor。链接的过程如图 18.3 所示。

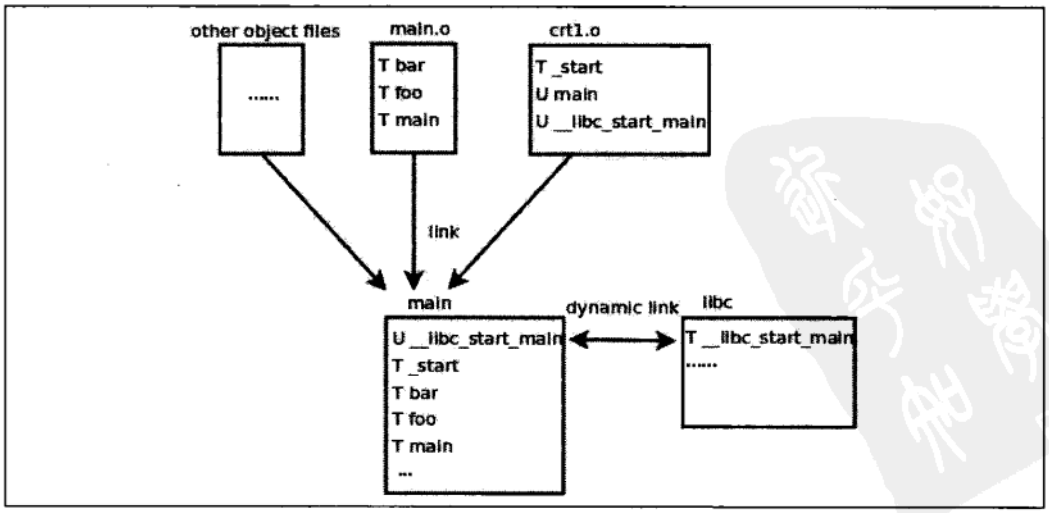


图 18.3　C 程序的链接过程

从图 18.3 可以看出，crt1.o 还引用了一个未定义符号__libc_start_main，这个符号在其他几个目标文件中也没有定义，所以链接生成可执行文件之后仍然是个未定义符号。事实上这个符号在 libc 中定义，libc 是一个共享库，它并不像其他目标文件一样链接到可执行文件 main 中，而是在运行时做动态链接：

1．操作系统在加载执行 main 这个程序时，首先查看它有没有需要动态链接的未定义符号。

2．如果需要做动态链接，就查看这个程序指定了哪些共享库，以及用什么动态链接器来做动态链接。我们在链接时用-lc 选项指定了共享库 libc，用-dynamic-linker /lib/ld-linux.so.2 指定了动态链接器，这些信息都会写到可执行文件中。

3．动态链接器加载共享库，在其中查找这些未定义符号的定义，完成链接过程。

我们回头看_start 的反汇编。首先将一系列参数压栈，然后通过 call 80482e8 指令调用库函数__libc_start_main 做初始化工作，其中最后一个压栈的参数 push $0x80483e5 正是 main 函数的首地址，__libc_start_main 在做完初始化工作之后会根据这个参数调用 main 函数。由于__libc_start_main 需要动态链接，所以这个库函数的指令在可执行文件 main 的反汇编中肯定是找不到的，然而我们在地址 0x80482e8 处找到了这几条指令：

```
Disassembly of section .plt:
...
080482e8 <__libc_start_main@plt>:
 80482e8:   ff 25 04 a0 04 08       jmp    *0x804a004
 80482ee:   68 08 00 00 00          push   $0x8
 80482f3:   e9 d0 ff ff ff          jmp    80482c8 <_init+0x30>
```

这几条指令位于.plt 段而不是.text 段，.plt 段协助完成动态链接，我们到下一章再详细解释。

main 函数最标准的原型应该是 int main(int argc, char *argv[]);，也就是说启动例程会传两个参数给 main 函数，这两个参数的含义将在第 22.6 节介绍。到目前为止我们都把 main 函数的原型写成 int main(void);，这也是 C 标准允许的，如果你认真分析了上一节的习题就应该知道，多传了参数而不用是没有问题的，少传了参数却用了则会出问题。

由于 main 函数是被启动例程调用的，所以从 main 函数 return 时就返回到启动例程中，main 函数的返回值被启动例程得到，如果将启动例程表示成等价的 C 代码（实际上启动例程一般是直接用汇编写的），则它调用 main 函数的形式是：

```
exit(main(argc, argv));
```

也就是说，启动例程得到 main 函数的返回值后，会立刻用它做参数调用 exit 函数。exit 也是 libc 的库函数，它首先做一些清理工作，然后调上一章讲过的_exit 系统调用终止进程，main 函数的返回值最终被传给_exit 系统调用，成为进程的退出状态。我们也可以在 main 函数中直接调用 exit 函数终止进程而不返回到启动例程，

例如[②]：

```
#include <stdlib.h>

int main(void)
{
        exit(4);
}
```

注意要包含头文件 stdlib.h。这样和 int main(void) { return 4; }的效果是一样的。在 Shell 中运行这个程序并查看它的退出状态：

```
$ ./a.out
$ echo $?
4
```

按照惯例，退出状态为 0 表示程序执行成功，退出状态非 0 表示出错。注意，退出状态只有 8 位，而且被 Shell 解释成无符号数，如果将上面的代码改为 exit(-1); 或 return -1;，则运行结果为：

```
$ ./a.out
$ echo $?
255
```

在 C 程序中也可以调用_exit 函数退出（需要包含头文件 unistd.h），它是_exit 系统调用的简单包装。怎么包装呢？它可能是一个 C 函数，其中内嵌了 movl $1, %eax、movl ?, %ebx 和 int $0x80 三条指令（稍后在第 18.5 节介绍这种语法）。它也可能是纯用汇编写的，但要符合 C 编译器的 Calling Convention，这样才能当成一个 C 函数来调用。这种对 int $0x80 指令简单包装的 C 函数通常也称为系统调用，在 Man Page 中系统调用位于第 2 个 Section，例如_exit(2)，而库函数位于第 3 个 Section，例如 exit(3)。第 3 个 Section 的库函数有些完全工作在用户模式，例如第 23.1.1 节要介绍的 strcpy(3)，而有些要调第 2 个 Section 的系统调用完成它的工作，例如 exit(3)函数首先做一些清理工作，然后调用_exit(2)进内核终止当前进程。那么所谓“清理工作”到底指哪些工作呢？到第 24.2.10 节再详细解释。头文件 unistd.h 中声明的函数并不是 C 标准库函数，而是 POSIX 标准定义的 UNIX 系统函数，但也在 libc 中实现。

> **关于 UNIX 标准**
>
> POSIX（Portable Operating System Interface）是由 IEEE 制定的标准，致力于统一各种 UNIX 系统的接口，促进各种 UNIX 系统向互相兼容的方向发展。IEEE 1003.1（也称为 POSIX.1）定义了 UNIX 系统的函数接口，既包括 C 标准库函数，也包括系统调用和其他 UNIX 库函数。POSIX.1

② 如果声明一个函数的返回值类型是 int，函数中每个分支控制流程必须写 return 语句指定返回值，否则返回值不确定（想想这是为什么），在启用-Wall 选项时编译器是会报警告的。但如果某个分支控制流程调用了 exit 而不写 return，编译器是允许的，因为它都没机会返回了，指不指定返回值也就无所谓了。

只定义接口而不定义实现，所以并不区分一个函数是库函数还是系统调用，至于哪些函数在用户空间实现，哪些函数在内核中实现，由操作系统的开发者决定，各种UNIX系统都不太一样。IEEE 1003.2定义了Shell的语法和各种基本命令的选项等。

在UNIX的发展历史上有BSD和SYSV两个分支，各自实现了很多不同的接口，比如BSD的网络编程接口是socket，而SYSV的网络编程接口是基于STREAMS的TLI。POSIX在统一接口的过程中，有些接口借鉴BSD的，有些接口借鉴SYSV的，还有些接口既不是来自BSD也不是来自SYSV，而是凭空发明出来的（例如线程库pthread就属于这种情况），通过Man Page的COMFORMING TO部分可以看出来一个函数接口属于哪种情况。Linux内核是完全从头编写的，并不继承BSD或SYSV的源代码，为了兼容现有的应用程序，Linux既实现了BSD的部分特性也实现了SYSV的部分特性，此外还有一些Linux独有的特性，比如epoll(7)，依赖于epoll这种接口的应用程序难以移植到其他系统，但在Linux系统上运行效率很高。

POSIX规定有些接口是必须实现的，而另外一些接口是可以选择实现的。有些非UNIX系统也实现了POSIX中必须实现的部分，那么也可以声称自己是POSIX兼容的，然而要想声称自己是UNIX，还必须实现一部分在POSIX中规定为可选实现的接口，这由另外一个标准SUS（Single UNIX Specification）规定。SUS是POSIX的超集，一部分在POSIX中规定为可选实现的接口在SUS中规定为必须实现，完整实现了这些接口的系统称为XSI（X/Open System Interface）兼容的。SUS标准由The Open Group维护，该组织拥有UNIX的注册商标（http://www.unix.org/），XSI兼容的系统可以从该组织获得授权使用UNIX这个商标。

一个进程调用exit或_exit终止，或者从main函数返回而终止，都属于正常终止（Normal Termination），也称为退出（Exit）。但并非所有的进程终止都是正常的，比如按Ctrl+C组合键终止一个进程，或者运行时产生段错误，或者用kill命令终止一个进程，这几种情况本质上都是进程收到一个信号然后内核把进程强行终止掉了，进程并没有执行_exit系统调用，也没有退出状态，这称为异常终止（Abnormal Termination）。关于信号请查阅参考文献[31]的第10章。

18.3 变量的存储布局

首先看下面的例子：

例18.2 研究变量的存储布局

```
#include <stdio.h>

const int A = 10;
int a = 20;
static int b = 30;
int c;
```

```
int main(void)
{
        static int a = 40;
        char b[] = "Hello world";
        register int c = 50;

        printf("Hello world %d\n", c);

        return 0;
}
```

我们在全局作用域和 main 函数的局部作用域各定义了一些变量，并且引入一些新的关键字 const、static、register 来修饰变量，那么这些变量的存储空间怎么分配呢？我们编译之后用 readelf 命令看它的符号表，了解各变量的地址分布。注意在下面的清单中我把符号表按地址从低到高的顺序重新排列了，并且只截取我们关心的那几行。

```
$ gcc main.c -g
$ readelf -a a.out
...
   65: 08048570     4 OBJECT  GLOBAL DEFAULT   16 A
   66: 0804a018     4 OBJECT  GLOBAL DEFAULT   24 a
   49: 0804a01c     4 OBJECT  LOCAL  DEFAULT   24 b
   50: 0804a020     4 OBJECT  LOCAL  DEFAULT   24 a.1706
   78: 0804a02c     4 OBJECT  GLOBAL DEFAULT   25 c
...
```

变量 A 用 const 修饰，表示 A 是只读的，不可修改，它被分配的地址是 0x8048570，从 readelf 的输出可以看到这个地址位于.rodata 段：

```
Section Headers:
  [Nr] Name      Type        Addr     Off    Size   ES Flg Lk Inf Al
...
  [14] .text     PROGBITS    08048390 000390 0001bc 00  AX  0   0 16
...
  [16] .rodata   PROGBITS    08048568 000568 00001c 00   A  0   0  4
...
  [24] .data     PROGBITS    0804a010 001010 000014 00  WA  0   0  4
  [25] .bss      NOBITS      0804a024 001024 00000c 00  WA  0   0  4
...
```

.rodata 段在内存中的地址是 0x08048568～0x08048583，在文件中的地址是 0x568～0x583，我们用 hexdump 命令看看这个段的内容：

```
$ hexdump -C a.out
...
00000560  5c fe ff ff 59 5b c9 c3  03 00 00 00 01 00 02 00  |\...Y[..........|
00000570  0a 00 00 00 48 65 6c 6c  6f 20 77 6f 72 6c 64 20  |....Hello world |
00000580  25 64 0a 00 00 00 00 00  00 00 00 00 00 00 00 00  |%d..............|
...
```

其中 0x570 地址处的 0a 00 00 00 就是变量 A。我们还看到程序中的字符串字面值

"Hello world %d\n"分配在.rodata段的末尾，在第8.4节讲过字符串字面值是只读的，相当于在全局作用域定义了一个const数组：

```
const char helloworld[] = {'H', 'e', 'l', 'l', 'o', ' ',
              'w', 'o', 'r', 'l', 'd', ' ', '%', 'd', '\n', '\0'};
```

在链接时.rodata段和.text段合并到Text Segment中，在加载运行时操作系统把Text Segment的页面只读保护起来，防止意外改写。从readelf的输出可以看出.rodata段和.text段被合并到一个Segment，.data段和.bss段被合并到另一个Segment。

```
Section to Segment mapping:
 Segment Sections...
  00
  01     .interp
  02     .interp .note.ABI-tag .note.gnu.build-id .hash .gnu.hash
.dynsym .dynstr .gnu.version .gnu.version_r .rel.dyn .rel.plt .init
 .plt .text .fini .rodata .eh_frame
  03     .ctors .dtors .jcr .dynamic .got .got.plt .data .bss
  04     .dynamic
  05     .note.ABI-tag .note.gnu.build-id
  06
  07     .ctors .dtors .jcr .dynamic .got
```

注意，像A这种const变量在定义时必须初始化。因为只有初始化时才有机会给它一个值，一旦定义之后就不能再改写了，如果给它赋值编译器会报错，也就是说，操作系统的内存管理和编译器的语义检查为全局const变量提供了双重保护。我们知道函数的局部变量在栈上分配，如果把局部变量声明为const就少了一层保护，操作系统无法对栈空间只读保护（因为栈上的其他数据要求可读可写），但编译器仍可以做语义检查。

.data段在内存中的地址是0x804a010～0x804a023，在.data段中有三个变量，a，b和a.1706。a是一个全局符号，而b被static关键字修饰了，导致它成为一个局部符号，所以static在这里的作用是声明b为局部符号，如果把多个目标文件链接在一起，局部符号只能在某一个目标文件中定义和使用，而不能在一个目标文件中定义却在另一个目标文件中引用，因为链接器不会对局部符号做符号解析。一个函数定义前面也可以用static修饰，表示这个函数名是局部符号。

还有一个a.1706是什么呢？它就是main函数中的static int a。函数中的static变量不同于我们以前讲的局部变量，它并不是在调用函数时分配，在函数返回时释放，而是像全局变量一样静态分配，所以用“static”（静态）这个词。另一方面，函数中的static变量也是局部作用域的，和以前讲的局部变量一样，a这个变量名只在main函数中起作用，在别的函数中说变量a就不是指它了，所以编译器给它的符号名加了一个后缀，变成a.1706，以便和全局变量a以及其他函数的静态变量a区分开。

.bss段在内存中的地址是0x804a024～0x804a02f（紧挨着.data段），变量c位于这个段。.data段和.bss段在链接时合并到Data Segment中，在加载运行时Data

Segment 的页面是可读可写的。.bss 段和.data 段的不同之处在于，.bss 段在文件中不占存储空间，加载到内存时这个段用 0 填充，C 语言规定全局变量和 static 变量（不管是函数里的还是函数外的）如果不初始化则初值为 0，未初始化的和明确初始化为 0 的全局变量、static 变量都会分配在.bss 段[③]。

现在还剩下 main 函数中的变量 b 和 c 没有分析。b 是一个数组，在栈上分配。我们看 main 函数的反汇编代码：

```
$ objdump -dS a.out
...
    char b[] = "Hello world";
 804845a:   c7 44 24 10 48 65 6c    movl   $0x6c6c6548,0x10(%esp)
 8048461:   6c
 8048462:   c7 44 24 14 6f 20 77    movl   $0x6f77206f,0x14(%esp)
 8048469:   6f
 804846a:   c7 44 24 18 72 6c 64    movl   $0x646c72,0x18(%esp)
 8048471:   00
    register int c = 50;
 8048472:   bb 32 00 00 00          mov    $0x32,%ebx

    printf("Hello world %d\n", c);
 8048477:   b8 74 85 04 08          mov    $0x8048574,%eax
 804847c:   89 5c 24 04             mov    %ebx,0x4(%esp)
 8048480:   89 04 24                mov    %eax,(%esp)
 8048483:   e8 dc fe ff ff          call   8048364 <printf@plt>
...
```

可见，给 b 初始化用的这个字符串"Hello world"不需要在.rodata 段分配，而是直接写在指令里，通过三条 movl 指令把 12 个字节写到栈上，这就是 b 的存储空间，如图 18.4 所示。

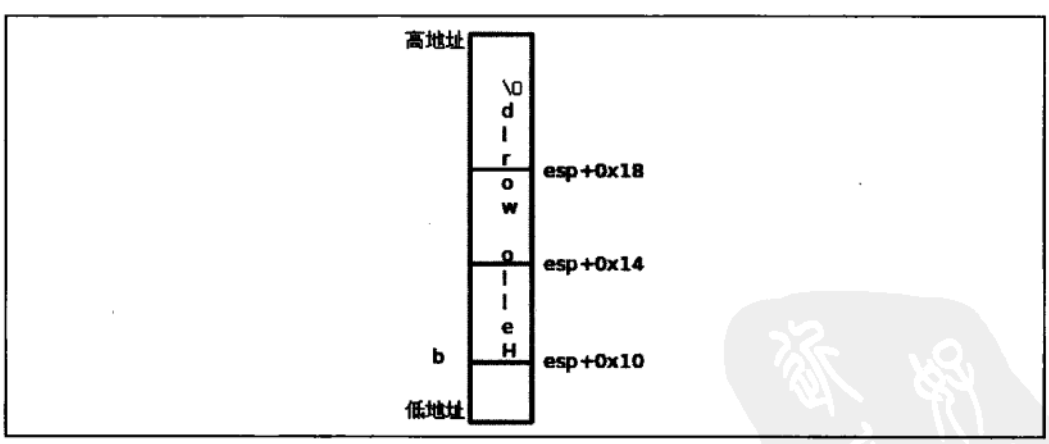


图 18.4　数组的存储布局

注意，虽然栈是从高地址向低地址增长的，但数组总是从低地址向高地址排列的，按从低地址到高地址的顺序依次是 b[0]、b[1]、b[2]等。

③ “bss”是历史遗留下来的名词，它的全称是“Block Started by Symbol”，最初是 IBM 704 汇编器的一条伪指令的名字，一直沿用至今。不过你也可以记成“Better Save Space”，因为.bss 段在文件中不占存储空间。

变量c并没有在栈上分配存储空间，而是直接存在ebx寄存器里，后面调用printf也是直接从ebx寄存器里取出c的值当参数压栈，这就是register关键字的作用，指示编译器尽可能分配一个寄存器来保存这个变量。我们还看到调用printf时对于"Hello world %d\n"这个参数压栈的是它在.rodata段中的首地址0x8048574，而不是把整个字符串压栈。在第8.4节讲过，字符串字面值和数组名类似，做右值使用时表示首元素的地址（或者说指向首元素的指针），我们在第22.4节还要继续讨论这个问题。

以前我们用“全局变量”和“局部变量”这两个概念，主要是从作用域上区分的，现在看来用这两个概念给变量分类太笼统了，需要进一步细分。我们总结一下相关的C语法。

作用域（Scope）这个概念适用于所有标识符，而不仅仅是变量，C语言的作用域分为以下几类：

- 函数作用域（Function Scope），标识符在整个函数中都有效。只有标号属于函数作用域。标号在函数中不需要先声明后使用，在前面用一个goto语句也可以跳转到后面的某个标号，但仅限于同一个函数之中。即使在函数内的某个语句块中定义一个标号，它也不局限于这个语句块，也是在整个函数中都有效。
- 文件作用域（File Scope），标识符在函数外声明，从它声明的位置开始直到这个源文件末尾都有效——严格说应该是直到编译单元（Translation Unit）末尾都有效。比如有源文件a.c包含了b.h和c.h，那么经过预处理把b.h和c.h在a.c中展开之后得到的代码称为一个编译单元。编译器将每个编译单元分别编译成一个目标文件，最后链接器把这些目标文件链接到一起成为一个可执行文件。

  上例中在函数外声明的标识符A、a、b、c以及标识符main都是文件作用域的，标识符printf在stdio.h中声明然后被包含到这个编译单元中，所以也是文件作用域的。此外，在函数外声明的类型名或Tag也属于文件作用域。
- 块作用域（Block Scope），标识符在一对{}括号中声明，即在函数体或语句块中声明，那么从它声明的位置开始到右}括号之间有效。上例中main函数里的a、b、c都是块作用域的。此外，函数定义中的形参也算块作用域的，从声明的位置开始到函数末尾之间有效。
- 函数原型作用域（Function Prototype Scope），如果标识符出现在函数原型中，这个函数原型只是一个声明而不是定义（没有函数体），那么标识符从声明的位置开始到这个原型结束之前有效。例如int foo(int a, int b);中的a和b。这样的函数原型只是告诉编译器函数返回值类型、参数个数以及各参数的类型，参数名其实是无关紧要的，省略掉也可以，比如写成int foo(int, int);也可以。

  对属于同一命名空间（Name Space）的重名标识符，内层作用域的标识符将覆盖外层作用域的标识符，例如局部变量名在它的函数中将覆盖重名的全局变量。命名空间可分为以下几类：

- 语句标号单独属于一个命名空间。例如在函数中局部变量和语句标号可以重名，互不影响。由于使用标号的语法和使用其他标识符的语法都不一样，编译器不会把它和别的标识符弄混。
- struct、enum 和 union（下一节介绍 union）的 Tag 属于一个命名空间。由于 Tag 前面总是带 struct、enum 或 union 关键字，所以编译器不会把它和别的标识符弄混。
- struct 和 union 的成员名属于一个命名空间。由于成员名总是通过.或->运算符来访问而不会单独使用，所以编译器不会把它和别的标识符弄混。
- 所有其他标识符，例如变量名、函数名、宏定义、typedef 定义的类型名、enum 成员等都属于同一个命名空间，如果有重名则按内层作用域覆盖外层作用域的规则处理。
- 如果宏定义和其他标识符重名，则宏定义覆盖所有其他标识符，因为宏定义在预处理阶段先处理，而其他标识符在编译阶段处理。

标识符的链接属性（Linkage）有三种：

- 外部链接（External Linkage），一个标识符在不同的编译单元中可能被声明多次，当这些编译单元链接成一个可执行文件时，如果这些声明都代表同一个变量或函数（即代表同一个内存地址），则这个标识符具有 External Linkage。具有 External Linkage 的标识符编译后在目标文件中是全局符号。上例中在函数外声明的标识符 a、c 以及 main 和 printf 都具有 External Linkage。
- 内部链接（Internal Linkage），一个标识符在某个编译单元中可能被声明多次，这些声明都代表同一个内存地址，但如果这个标识符在不同的编译单元中被声明多次，在链接时这些声明就不代表同一个内存地址，这样的标识符具有 Internal Linkage。上例中在函数外声明的标识符 b 具有 Internal Linkage。如果有另一个程序 foo.c 和上例的 main.c 链接在一起，在 foo.c 中也声明一个 static int b;，则这个 b 和那个 b 不代表同一个变量。具有 Internal Linkage 的标识符编译后在目标文件中是局部符号，在链接时不做符号解析。

  注意上例中 main 函数里面声明的那个 static int a = 40;不能算 Internal Linkage 的。如果在同一编译单元的另一个函数中也声明一个 static int a;，则这个 a 和那个 a 不代表同一个变量，在编译时会把这两个标识符改名成两个不同的符号。
- 无链接属性（No Linkage）。除以上情况之外的标识符都属于 No Linkage，例如函数的局部变量，以及不表示变量和函数的其他标识符。除了函数、全局变量、静态变量之外的标识符在编译时不会变成符号，所以没有链接属性。

最后还有两个问题，具有 External Linkage 的标识符如何在多个编译单元中声明多次？具有 Internal Linkage 的标识符如何在一个编译单元中声明多次？我们将在下

一章详细介绍，读者在学习第19.2节时应该回到这里复习Linkage的概念。

存储类修饰符（Storage Class Specifier）指的是以下几个关键字，可以修饰变量或函数声明：

- static，用它修饰的变量的存储空间是静态分配的，用它修饰的文件作用域的变量或函数具有Internal Linkage。
- auto，用它修饰的变量在函数调用时自动在栈上分配存储空间，函数返回时自动释放，上例中main函数里的b其实就是用auto修饰的，只不过auto可以省略不写，auto不能修饰文件作用域的变量。
- register，编译器对于用register修饰的变量会尽可能分配一个专门的寄存器来存储，但如果实在分配不开寄存器，编译器就把它当auto变量处理了，register不能修饰文件作用域的变量。现在一般编译器的优化都做得很好了，编译器自己会想办法有效地利用CPU寄存器，所以现在register关键字也用得比较少了。
- extern，用于多次声明同一个具有External Linkage或Internal Linkage的标识符，下一章再详细介绍它的用法。
- typedef，在第15.2.4节讲过这个关键字，它并不是用来修饰变量的，而是定义一个类型名。在那一节也讲过，看typedef声明怎么看呢，首先去掉typedef把它看成变量声明，看这个变量是什么类型的，那么typedef就给什么类型起了一个类型名。因此，typedef在语法结构中出现的位置和前面几个关键字一样，也是修饰变量声明的，所以从语法（而不是语义）的角度把它和前面几个关键字归类到一起。

注意，上面介绍的const关键字不是一个Storage Class Specifier，虽然看起来它也修饰一个变量声明，但是在以后介绍的更复杂的声明中const在语法结构中允许出现的位置和Storage Class Specifier是不完全相同的。const和volatile、restrict关键字属于同一类语法元素，称为类型限定符（Type Qualifier），volatile关键字在第18.6节介绍，restrict关键字在第24.1.3节介绍。

在一个变量或函数声明的开头如果有修饰或限定，通常按Storage Class Specifier、Function Specifier（即第20.2.2节要讲的inline关键字）、Type Qualifier、Type Specifier（例如int、double等关键字）的顺序来写，例如：

```
static const int i = 1;
```

其实在语法上这四种修饰限定的位置可以任意排列，例如int const typedef constant_t;也是合乎语法的，但是可读性很差。

变量的生存期（Storage Duration，或者Lifetime）分为以下几类：

- 静态生存期（Static Storage Duration），具有External或Internal Linkage，或者被static修饰的变量，在程序开始执行时分配内存和初始化，此后便一直存在直到程序结束。这种变量通常位于.rodata、.data或.bss段，上例

中在函数外声明的变量 A、a、b、c 以及 main 函数里声明的变量 a 都属于静态生存期。

- 自动生存期（Automatic Storage Duration），无链接属性并且没有被 static 修饰的变量，这种变量在进入块作用域时在栈上或寄存器中分配，在退出块作用域时释放。上例中 main 函数里声明的变量 b 和 c 属于自动生存期。
- 动态分配生存期（Allocated Storage Duration），在第 23.1.2 节会讲到调用 malloc 函数可以在进程的堆空间分配内存，调用 free 函数可以释放这块内存。动态分配生存期不同于静态生存期，因为它不是在程序开始执行时就分配，直到程序结束才释放，它是在某个函数中调用 malloc 分配的；也不同于自动生存期，因为它不是在调用函数时自动分配，也不在函数返回时自动释放，而是需要调用 free 来释放。

18.4　结构体和联合体

我们再用反汇编的方法研究一下 C 语言的结构体：

例 18.3　研究结构体

```
#include <stdio.h>

int main(void)
{
    struct {
            char a;
            short b;
            int c;
            char d;
    } s;

    s.a = 1;
    s.b = 2;
    s.c = 3;
    s.d = 4;
    printf("%u\n", sizeof(s));

    return 0;
}
```

main 函数中几条语句的反汇编结果如下：

```
    s.a = 1;
80483ed:   c6 44 24 14 01          movb   $0x1,0x14(%esp)
    s.b = 2;
80483f2:   66 c7 44 24 16 02 00    movw   $0x2,0x16(%esp)
    s.c = 3;
80483f9:   c7 44 24 18 03 00 00    movl   $0x3,0x18(%esp)
8048400:   00
    s.d = 4;
8048401:   c6 44 24 1c 04          movb   $0x4,0x1c(%esp)
```

从访问结构体成员的指令可以看出，结构体的四个成员在栈上的排列如图 18.5 所示。

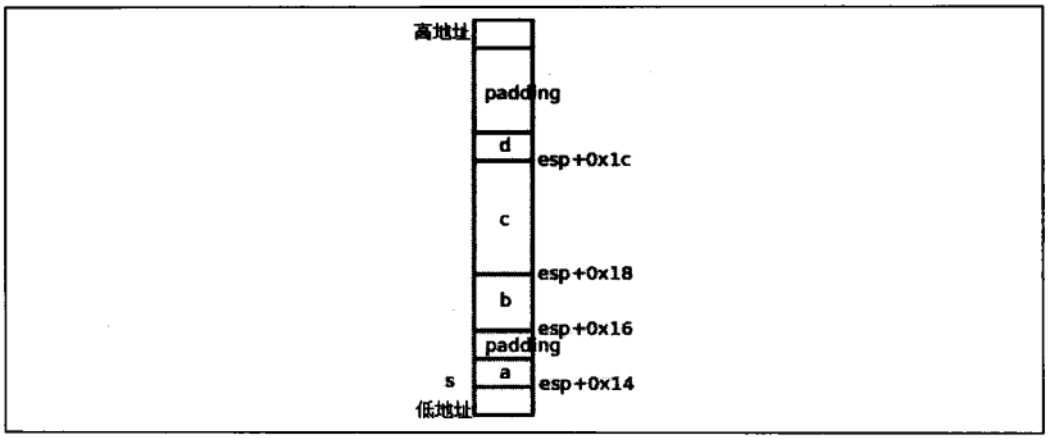


图 18.5 结构体的存储布局

虽然栈是从高地址向低地址增长的，但结构体成员也是从低地址向高地址排列的，这一点和数组类似。但有一点和数组不同，结构体的各成员并不是一个紧挨一个排列的，中间有空隙，称为填充（Padding），不仅如此，在这个结构体的末尾也有三个字节的填充，所以 sizeof(s)的值是 12。

为什么编译器要这样处理呢？有一个知识点我此前一直回避没讲，大多数计算机体系结构对于访问内存的指令是有限制的，在 32 位平台上，如果一条指令访问 4 个字节（比如上面的 movl），起始内存地址应该是 4 的整数倍，如果一条指令访问两个字节（比如上面的 movw），起始内存地址应该是 2 的整数倍，这称为对齐（Alignment），访问一个字节的指令（比如上面的 movb）没有对齐要求。如果指令所访问的内存地址没有正确对齐会怎么样呢？在有些平台上将不能访问内存，引发一个异常，在 x86 平台上倒是能访问内存，但是不对齐的指令比对齐的指令执行效率要低，所以编译器在安排各种变量的地址时都会考虑到对齐的问题。

对于本例中的结构体，编译器会把它的基地址对齐到 4 字节边界，也就是说，esp+0x14 这个地址一定是 4 的整数倍。s.a 占一个字节，没有对齐的问题。s.b 占两个字节，如果 s.b 紧挨在 s.a 后面，它的地址就不能是 2 的整数倍了，所以编译器会在结构体中插入一个填充字节，使 s.b 的地址是 2 的整数倍。s.c 占 4 字节，紧挨在 s.b 的后面就可以了，因为 esp+0x18 这个地址也是 4 的整数倍。为什么 s.d 的后面也要有填充位填充到 4 字节边界呢？这是为了便于安排这个结构体后面的变量地址，假如用这种结构体类型组成一个数组，那么由于前一个结构体的末尾已经有填充字节对齐到 4 字节边界了，后一个结构体只需和前一个结构体紧挨着排列就可以了。事实上，C 标准规定数组元素必须紧挨着排列，不能有空隙，这样才能保证每个元素的地址可以按“基地址+n×每个元素的字节数”简单计算出来。

合理设计结构体各成员的排列顺序可以节省存储空间，如果上例中的结构体改成这样就可以避免产生填充字节：

```
struct {
        char a;
        char d;
        short b;
        int c;
} s;
```

此外，gcc 提供了一种扩展语法可以消除结构体中的填充字节：

```
struct {
        char a;
        short b;
        int c;
        char d;
} __attribute__((packed)) s;
```

但这样就不能保证结构体成员的对齐了，在访问 b 和 c 的时候可能会有效率问题，甚至无法访问，所以除非有特别的理由，一般不要使用这种语法。

以前我们讲过的数据类型最少也要占一个字节，而结构体中还可以使用 Bit-field 语法定义只占几个 bit 的成员。下面这个例子出自王聪的网站（http://www.wangcong.org/）：

例 18.4　Bit-field

```
#include <stdio.h>

typedef struct {
        unsigned int one:1;
        unsigned int two:3;
        unsigned int three:10;
        unsigned int four:5;
        unsigned int :2;
        unsigned int five:8;
        unsigned int six:8;
} demo_type;

int main(void)
{
        demo_type s = { 1, 5, 513, 17, 129, 0x81 };
        printf("sizeof demo_type = %u\n", sizeof(demo_type));
        printf("values: s=%u,%u,%u,%u,%u,%u\n",
              s.one, s.two, s.three, s.four, s.five, s.six);

        return 0;
}
```

s 这个结构体的布局如图 18.6 所示：

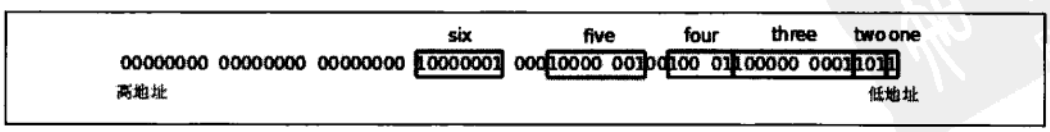


图 18.6　Bit-field 的存储布局

Bit-field 也属于整型，可以用 int 或 unsigned int 声明，表示有符号数或无符号数，但它不像普通的 int 型一样占 4 个字节，冒号后面的数字表示这个 Bit-field 占几个 bit。上例中的 unsigned int :2;定义一个未命名的 Bit-field 占两个 bit。即使不写未命名的 Bit-field，编译器也有可能在两个成员之间插入填充位，例如图 18.6 的 five 和 six 之间有填充位，这样 six 这个成员就刚好单独占一个字节了，访问效率会比较高，这个结构体的末尾还填充了 3 个字节，以便对齐到 4 字节边界。以前我们说过 x86 的 Byte Order 是小端的，从图 18.6 中 one 和 two 的排列顺序可以看出，如果对一个字节再细分，则字节中的 Bit Order 也是小端的，因为排在结构体前面的成员（靠近低地址一边的成员）取字节中的低位。关于如何排列 Bit-field 在 C 标准中没有明确规定，这跟 Byte Order、Bit Order、对齐等问题都有关，不同的平台和编译器可能会排列得很不一样，要编写可移植的代码就不能假定 Bit-field 是按某种固定方式排列的。Bit-field 在驱动程序中是很有用的，因为经常需要单独操作设备寄存器中的一个或几个 bit，但一定要小心使用，首先弄清楚每个 Bit-field 和设备寄存器中每个 bit 的对应关系。

在上例中我没有给出反汇编结果，直接画了个图说这个结构体的布局是这样的，那我有什么证据这么说呢？上例的反汇编结果比较繁琐，我们可以用另一种手段得到这个结构体的内存布局：

例 18.5 联合体

```
#include <stdio.h>

typedef union {
    struct {
            unsigned int one:1;
            unsigned int two:3;
            unsigned int three:10;
            unsigned int four:5;
            unsigned int :2;
            unsigned int five:8;
            unsigned int six:8;
        } bitfield;
        unsigned char byte[8];
} demo_type;

int main(void)
{
        demo_type u = {{ 1, 5, 513, 17, 129, 0x81 }};
        printf("sizeof demo_type = %u\n", sizeof(demo_type));
        printf("values: u=%u,%u,%u,%u,%u,%u\n",
              u.bitfield.one, u.bitfield.two, u.bitfield.three,
              u.bitfield.four, u.bitfield.five, u.bitfield.six);
        printf("hex dump of u: %x %x %x %x %x %x %x %x\n",
              u.byte[0], u.byte[1], u.byte[2], u.byte[3],
              u.byte[4], u.byte[5], u.byte[6], u.byte[7]);

        return 0;
}
```

关键字 union 定义一种新的数据类型，称为联合体，其语法类似于结构体。一

个联合体的各个成员占用相同的内存空间，联合体的长度等于其中最长成员的长度。比如 u 这个联合体占 8 个字节，如果访问成员 u.bitfield，则把这 8 个字节看成一个由 Bit-field 组成的结构体，如果访问成员 u.byte，则把这 8 个字节看成一个数组。

联合体如果用 Initializer 初始化，则只初始化它的第一个成员，例如 demo_type u = {{ 1, 5, 513, 17, 129, 0x81 }};初始化的是 u.bitfield，这样我们只知道 u.bitfield 结构体各成员的值是多少，却不知道它的内存布局是什么样的，然后我们换一个视角，同样是这 8 个字节，我们把它看成一个 u.byte 数组，就可以看出每个字节分别是多少，内存布局是什么样了。

如果用 C99 的 Memberwise 初始化语法，则可以初始化联合体的任意一个成员，例如：

```
demo_type u = { .byte = {0x1b, 0x60, 0x24, 0x10, 0x81, 0, 0, 0} };
```

最后回顾一下我们讲过的这些概念：

1．数据类型的长度（例如 ILP32、LP64）

2．Calling Convention

3．访问内存地址的对齐要求

4．结构体和 Bit-field 的填充方式

5．字节序（大端、小端）

6．用什么指令做系统调用，各种系统调用的参数

7．可执行文件和库文件格式（例如 ELF 格式）

这些统称为应用程序二进制接口规范（ABI，Application Binary Interface），如果两个平台具有相同的体系结构，并且遵循相同的 ABI，就可以保证一个平台上的二进制程序直接拷贝到另一个平台就能运行，不用重新编译。比如有两台 x86 计算机，一台是 PC，另一台是上网本，分别装了不同的 Linux 发行版，那么从一台机器拷贝一个二进制程序到另一台机器同样也能运行，因为这两台机器具有相同的体系结构，并且操作系统遵循相同的 ABI。如果在同一台计算机上装了 Linux 和 Windows 两个操作系统，在 Windows 系统中运行一个 Linux 的二进制程序是不行的，因为这两个操作系统的 ABI 不同。

习题

1．编写一个程序，测试运行它的平台是大端还是小端字节序。

18.5 C内联汇编

用C写程序比直接用汇编写程序更简洁，可读性更好，但效率可能不如汇编程序，因为C程序毕竟要经由编译器生成汇编代码，尽管现代编译器的优化已经做得很好了，但还是不如手写的汇编代码。另外，有些平台相关的指令必须手写，在C语言中没有等价的语法，因为C语言中的概念是对各种平台的抽象，每种平台特有的一些东西就不会在C语言中出现了，例如x86是端口I/O，而C语言就没有这个概念，所以in/out指令必须用汇编来写。

C语言简洁易读，容易组织规模较大的代码，汇编效率高，而且写一些特殊指令必须用汇编，为了把这两方面的好处都占全了，gcc提供了一种扩展语法可以在C代码中使用内联汇编（Inline Assembly）。最简单的格式是__asm__("assembly code");，例如__asm__("nop");，nop这条指令什么都不做，只是让CPU空转一个指令执行周期。如果需要执行多条汇编指令，则应该用\n\t将各条指令分隔开，例如：

```
__asm__("movl $1, %eax\n\t"
        "movl $4, %ebx\n\t"
        "int $0x80");
```

通常内联汇编需要和C代码中的变量建立关联，要用到完整的内联汇编格式：

```
__asm__(assembler template
        : output operands                /* optional */
        : input operands                 /* optional */
        : list of clobbered registers    /* optional */
        );
```

这种格式由四部分组成，第一部分是汇编指令，和上面的例子一样，第二部分和第三部分是约束条件，第二部分告诉编译器汇编指令的运算结果要输出到哪些C语言操作数中，这些操作数应该是左值表达式，第三部分告诉编译器汇编指令需要从哪些C语言操作数获得输入，第四部分是在汇编指令中被修改的寄存器列表（称为Clobber List），告诉编译器哪些寄存器的值在执行这条__asm__语句时会改变。后三个部分是可选的，如果有就填写，没有就空着只写个冒号，例如：

例18.6 内联汇编

```
#include <stdio.h>

int main(void)
{
        int a = 10, b;

        __asm__("movl %1, %%eax\n\t"
                "movl %%eax, %0\n\t"
                :"=r"(b)        /* output */
                :"r"(a)         /* input */
                :"%eax"         /* clobbered register */
                );
```

```
        printf("Result: %d, %d\n", a, b);
        return 0;
}
```

这个程序将变量 a 的值赋给 b。"r"(a)告诉编译器分配一个寄存器保存变量 a 的值，作为汇编指令的输入，也就是指令中的%1（按照约束条件的顺序，b 对应%0，a 对应 1%），至于%1 究竟代表哪个寄存器则由编译器自己决定。汇编指令首先把%1 所代表的寄存器的值传给 eax（为了和%1 这种占位符区分，eax 前面要求加两个%号），然后把 eax 的值再传给%0 所代表的寄存器。"=r"(b)表示把%0 所代表的寄存器的值输出给变量 b。在执行这两条指令的过程中，寄存器 eax 的值被改变了，所以把"%eax"写在第四部分，告诉编译器在执行这条__asm__语句时 eax 要被改写，所以在此期间不要用 eax 保存其他值。

我们看一下这个程序的反汇编结果：

```
    __asm__("movl %1, %%eax\n\t"
80483f5:   8b 54 24 1c          mov    0x1c(%esp),%edx
80483f9:   89 d0                mov    %edx,%eax
80483fb:   89 c2                mov    %eax,%edx
80483fd:   89 54 24 18          mov    %edx,0x18(%esp)
           "movl %%eax, %0\n\t"
           :"=r"(b)        /* output */
           :"r"(a)         /* input */
           :"%eax"         /* clobbered register */
           );
```

可见%0 和%1 都代表 edx 寄存器，首先把变量 a（位于 esp+0x1c 的位置）的值传给 edx 然后执行内联汇编的两条指令，然后把 edx 的值传给 b（位于 esp+0x18 的位置）。

关于内联汇编就介绍这么多，本书不做深入讨论，关于 gcc 的各种扩展语法读者可以查阅参考文献[24]或参考文献[25]。

18.6　volatile 限定符

本节探讨一下编译器优化会对生成的指令产生什么影响，在此基础上介绍 C 语言的 volatile 限定符。首先看下面的例子。

例 18.7　volatile 限定符

```
/* artificial device registers */
unsigned char recv;
unsigned char send;

/* memory buffer */
unsigned char buf[3];

int main(void)
{
        buf[0] = recv;
        buf[1] = recv;
```

```
        buf[2] = recv;
        send = ~buf[0];
        send = ~buf[1];
        send = ~buf[2];

        return 0;
}
```

我们用 recv 和 send 这两个全局变量来模拟设备寄存器。假设某种平台采用内存映射 I/O，串口发送寄存器和串口接收寄存器位于固定的内存地址，而 recv 和 send 这两个全局变量也有固定的内存地址，所以在这个例子中把它们假想成串口接收寄存器和串口发送寄存器。在 main 函数中，首先从串口接收三个字节存到 buf 数组，然后把这三个字节取反，依次从串口发送出去[④]。我们查看这段代码的反汇编结果：

```
      buf[0] = recv;
80483b7:      0f b6 05 19 a0 04 08     movzbl 0x804a019,%eax
80483be:      a2 1a a0 04 08          mov    %al,0x804a01a
      buf[1] = recv;
80483c3:      0f b6 05 19 a0 04 08     movzbl 0x804a019,%eax
80483ca:      a2 1b a0 04 08          mov    %al,0x804a01b
      buf[2] = recv;
80483cf:      0f b6 05 19 a0 04 08     movzbl 0x804a019,%eax
80483d6:      a2 1c a0 04 08          mov    %al,0x804a01c
      send = ~buf[0];
80483db:      0f b6 05 1a a0 04 08     movzbl 0x804a01a,%eax
80483e2:      f7 d0                   not    %eax
80483e4:      a2 18 a0 04 08          mov    %al,0x804a018
      send = ~buf[1];
80483e9:      0f b6 05 1b a0 04 08     movzbl 0x804a01b,%eax
80483f0:      f7 d0                   not    %eax
80483f2:      a2 18 a0 04 08          mov    %al,0x804a018
      send = ~buf[2];
80483f7:      0f b6 05 1c a0 04 08     movzbl 0x804a01c,%eax
80483fe:      f7 d0                   not    %eax
8048400:      a2 18 a0 04 08          mov    %al,0x804a018
```

movz 指令把字长较短的值存到字长较长的存储单元中，存储单元的高位用 0 填充。该指令可以有 b（byte）、w（word）、l（long）三种后缀，分别表示单字节、两字节和四字节。比如 movzbl 0x804a019,%eax 表示把地址 0x804a019 处的一个字节存到 eax 寄存器中，而 eax 寄存器的长度是 4 个字节，高 3 个字节用 0 填充，如果高 3 个字节采用符号扩展则应该用 movsbl 指令，z 表示 zero 而 s 表示 sign。下一条指令 mov %al,0x804a01a 中的 al 寄存器正是 eax 寄存器的低字节，这条指令把 eax 寄存器的低字节存到地址 0x804a01a 处的一个字节中。可以用不同的名字单独访问 x86 寄存器的低 8 位、次低 8 位、低 16 位或者完整的 32 位，以 eax 为例，al 表示低 8 位，ah 表示次低 8 位，ax 表示低 16 位，如图 18.7 所示。

④ 实际的串口设备通常有标志位指示是否有数据到达以及是否可以发送下一个字节的数据，通常要先查询标志位再做读写操作，在这个例子中我们抓主要矛盾，忽略这些细节。

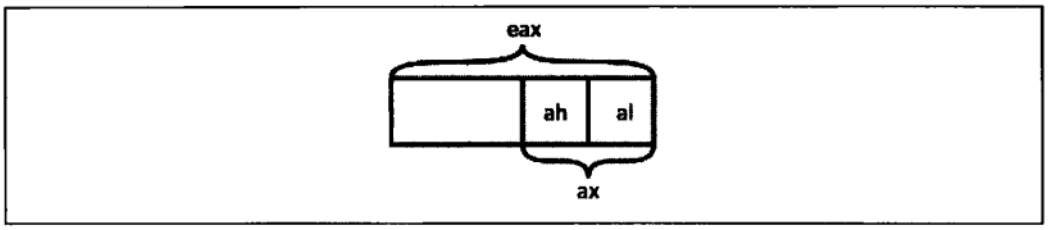


图 18.7　eax 寄存器

如果指定优化选项-O 编译，反汇编的结果就不一样了：

```
$ gcc main.c -g -O
$ objdump -dS a.out|less
...
      buf[0] = recv;
 80483b7:      0f b6 05 19 a0 04 08    movzbl 0x804a019,%eax
 80483be:      a2 1a a0 04 08          mov    %al,0x804a01a
      buf[1] = recv;
 80483c3:      a2 1b a0 04 08          mov    %al,0x804a01b
      buf[2] = recv;
 80483c8:      a2 1c a0 04 08          mov    %al,0x804a01c
      send = ~buf[0];
      send = ~buf[1];
      send = ~buf[2];
 80483cd:      f7 d0                   not    %eax
 80483cf:      a2 18 a0 04 08          mov    %al,0x804a018
...
```

前三条语句从串口接收三个字节，而编译生成的指令显然不符合我们的意图：只有第一条语句从内存地址 0x804a019 读一个字节到寄存器 eax 中，然后从寄存器 al 保存到 buf[0]，后两条语句就不再从内存地址 0x804a019 读取，而是直接把寄存器 al 的值保存到 buf[1]和 buf[2]。后三条语句把 buf 中的三个字节取反再发送到串口，编译生成的指令也不符合我们的意图：只有最后一条语句把 eax 取反然后写到内存地址 0x804a018 了，前两条语句形同虚设，根本不生成指令。

为什么编译器优化的结果会错呢？因为编译器并不知道 0x804a018 和 0x804a019 是设备寄存器的地址，把它们当成普通的内存单元了。如果是普通的内存单元，只要程序不去改写它，它就不会变，可以先把内存单元里的值读到寄存器缓存起来，以后每次用到这个值就直接从寄存器读取，这样效率更高，我们知道读寄存器远比读内存要快。另一方面，如果对一个普通的内存单元连续做三次写操作，只有最后一次的值会保存到内存单元中，所以前两次写操作是多余的，可以优化掉。然而访问设备寄存器的代码这样优化就错了，因为设备寄存器通常具有以下特性：

- 设备寄存器中的数据不需要改写就可以自己发生变化，每次读上来的值可能不一样。
- 连续多次向设备寄存器中写数据并不是在做无用功，而是给设备发命令，是有意义的。

用优化选项编译生成的指令明显效率更高，但使用不当会出错，为了避免编译器自作聪明，把不该优化的也优化了，程序员应该明确告诉编译器哪些内存单元的

访问是不能优化的，在C语言中可以用volatile限定符修饰变量，就是告诉编译器，即使在编译时指定了优化选项，每次读这个变量仍然要老老实实从内存读取，每次写这个变量也仍然要老老实实写回内存，不能省略任何步骤。我们把代码的开头几行改成：

```
/* artificial device registers */
volatile unsigned char recv;
volatile unsigned char send;
```

然后指定优化选项-O编译，查看反汇编的结果：

```
      buf[0] = recv;
80483b7:      0f b6 0d 19 a0 04 08     movzbl 0x804a019,%ecx
80483be:      88 0d 1a a0 04 08        mov    %cl,0x804a01a
      buf[1] = recv;
80483c4:      0f b6 15 19 a0 04 08     movzbl 0x804a019,%edx
80483cb:      88 15 1b a0 04 08        mov    %dl,0x804a01b
      buf[2] = recv;
80483d1:      0f b6 05 19 a0 04 08     movzbl 0x804a019,%eax
80483d8:      a2 1c a0 04 08           mov    %al,0x804a01c
      send = ~buf[0];
80483dd:      f7 d1                 not    %ecx
80483df:      88 0d 18 a0 04 08        mov    %cl,0x804a018
      send = ~buf[1];
80483e5:      f7 d2                 not    %edx
80483e7:      88 15 18 a0 04 08        mov    %dl,0x804a018
      send = ~buf[2];
80483ed:      f7 d0                 not    %eax
80483ef:      a2 18 a0 04 08           mov    %al,0x804a018
```

确实每次读recv都从内存地址0x804a019读取，每次写send也都写到内存地址0x804a018了。值得注意的是，每次写send并不需要取出buf中的值，而是取出先前缓存在寄存器eax、edx、ecx中的值，做取反运算然后写下去，这是因为buf并没有用volatile限定，读者可以试着在buf的定义前面也加上volatile，再优化编译，再查看和比较反汇编的结果。

gcc的编译优化选项有-O0、-O、-O1、-O2、-O3、-Os几种。-O0表示不优化，这是缺省的选项。-O1、-O2和-O3这几个选项一个比一个优化得更多，编译时间也更长。-O和-O1相同。-Os表示为缩小目标文件的尺寸而优化。具体每种选项做了哪些优化请参考gcc(1)。

从上面的例子还可以看到，如果在编译时指定了优化选项，源代码和生成指令的次序可能无法对应，甚至有些源代码可能不对应任何指令，被彻底优化掉了。这一点在用gdb做源码级调试时尤其需要注意（做指令级调试没关系），在为调试而编译时不要指定优化选项，否则可能无法一步步跟踪源代码的执行过程。

有了volatile限定符可以防止编译器优化对设备寄存器的访问，但对于有Cache的平台仅仅这样还不够，还是无法防止Cache优化对设备寄存器的访问。在访问内存地址时Cache对程序员是透明的，比如执行movzbl 0x804a019,%eax这样一条指令，我们并不知道eax的值是真的从内存地址0x804a019读到的，还是从Cache

中读到的，如果 Cache 已经缓存了这个地址的数据就从 Cache 读，如果 Cache 没有缓存就从内存读，这些步骤都是硬件自动做的，而不是用指令控制 Cache 去做的，程序员写的指令中只有寄存器、内存地址，而没有 Cache，程序员甚至不需要知道 Cache 的存在。同样道理，如果执行了 mov %al,0x804a01a 这样一条指令，我们并不知道寄存器的值是真的写回内存了，还是只写到了 Cache 中，以后再由 Cache 写回内存，即使只写到了 Cache 中而暂时没有写回内存，下次读 0x804a01a 这个地址时仍然可以从 Cache 中读到上次写的数据。然而，在读写设备寄存器时 Cache 的存在就不容忽视了，如果串口发送和接收寄存器的内存地址被 Cache 缓存了会有什么问题呢？如图 18.8 所示。

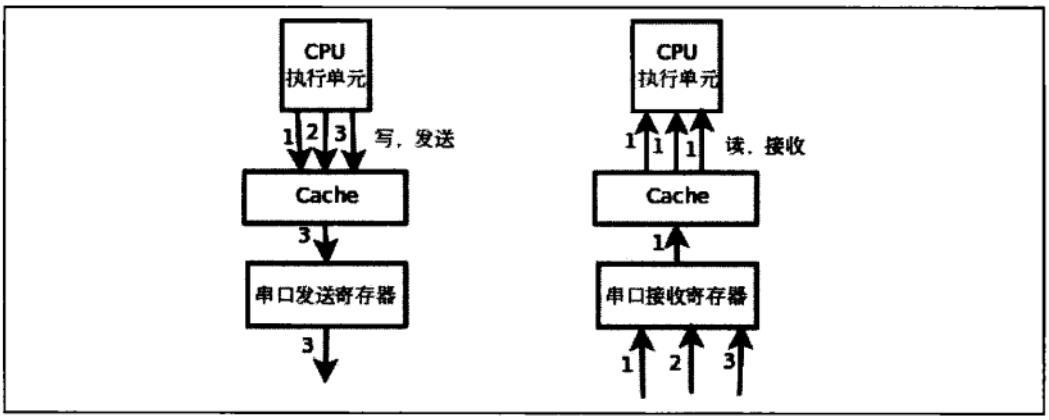


图 18.8　串口发送和接收寄存器被 Cache 缓存会有什么问题

如果串口发送寄存器的地址被 Cahce 缓存，CPU 执行单元对串口发送寄存器做写操作都写到 Cache 中去了，串口发送寄存器并没有及时得到数据，也就不能及时发送，CPU 执行单元先后发出的 1、2、3 三个字节都会写到 Cache 中的同一个单元，最后 Cache 中只保存了第 3 个字节，如果这时 Cache 把数据写回到串口发送寄存器，只能把第 3 个字节发出去，前两个字节就丢失了。与此类似，如果串口接收寄存器的地址被 Cache 缓存，CPU 执行单元在读第 1 个字节时，Cache 会从串口接收寄存器读上来并缓存，然而串口接收寄存器后面收到的 2、3 两个字节 Cache 并不知道，因为 Cache 把串口接收寄存器当做普通内存单元，并且相信内存单元中的数据是不会自己变的，以后每次读串口接收寄存器时，Cache 都会把缓存的第 1 个字节提供给 CPU 执行单元。

通常，有 Cache 的平台都有办法对某一段地址范围禁用 Cache，一般是在页表中设置的，可以设定哪些页面允许 Cache 缓存，哪些页面不允许 Cache 缓存，MMU 不仅要做地址转换和访问权限检查，也要配合 Cache 工作。

除了设备寄存器需要用 volatile 限定之外，当一个全局变量被同一进程中的多个控制流程访问时也要用 volatile 限定，比如信号处理函数和多线程就属于这种情况，读者可以查阅参考文献[31]的 10.15 节。

第 19 章

链接详解

19.1 多目标文件的链接

现在我们把例 12.1 拆成两个.c 文件，stack.c 实现堆栈，而 main.c 使用堆栈：

```
/* stack.c */
char stack[512];
int top = -1;

void push(char c)
{
        stack[++top] = c;
}

char pop(void)
{
        return stack[top--];
}

int is_empty(void)
{
        return top == -1;
}
```

这段程序和原来有点不同，在例 12.1 中 top 总是指向栈顶元素的下一个元素，而在这段程序中 top 总是指向栈顶元素，所以要初始化成-1 才表示空堆栈，这两种堆栈使用习惯都很常见。

```
/* main.c */
#include <stdio.h>

int a, b = 1;

int main(void)
{
        push('a');
        push('b');
        push('c');

        while(!is_empty())
                putchar(pop());
        putchar('\n');
```

```
    return 0;
}
```

a 和 b 这两个变量没有用，只是为了顺便说明链接过程才加上的。编译的步骤和以前一样，可以一步编译：

```
$ gcc main.c stack.c -o main
```

也可以分多步编译：

```
$ gcc -c main.c
$ gcc -c stack.c
$ gcc main.o stack.o -o main
```

如果用 nm 命令查看目标文件的符号表，会发现 main.o 中有未定义的符号 push、pop、is_empty、putchar，前三个符号在 stack.o 中定义了，在链接时做符号解析，而 putchar 是 libc 的库函数，在可执行文件 main 中仍然是未定义的，要在程序运行时做动态链接。

通过 readelf -a main 命令可以看到：main 的.bss 段合并了 main.o 和 stack.o 的.bss 段，其中包含了变量 a 和 stack；main 的.data 段合并了 main.o 和 stack.o 的.data 段，其中包含了变量 b 和 top；main 的.text 段合并了 main.o 和 stack.o 的.text 段，包含了各函数的指令，如图 19.1 所示。

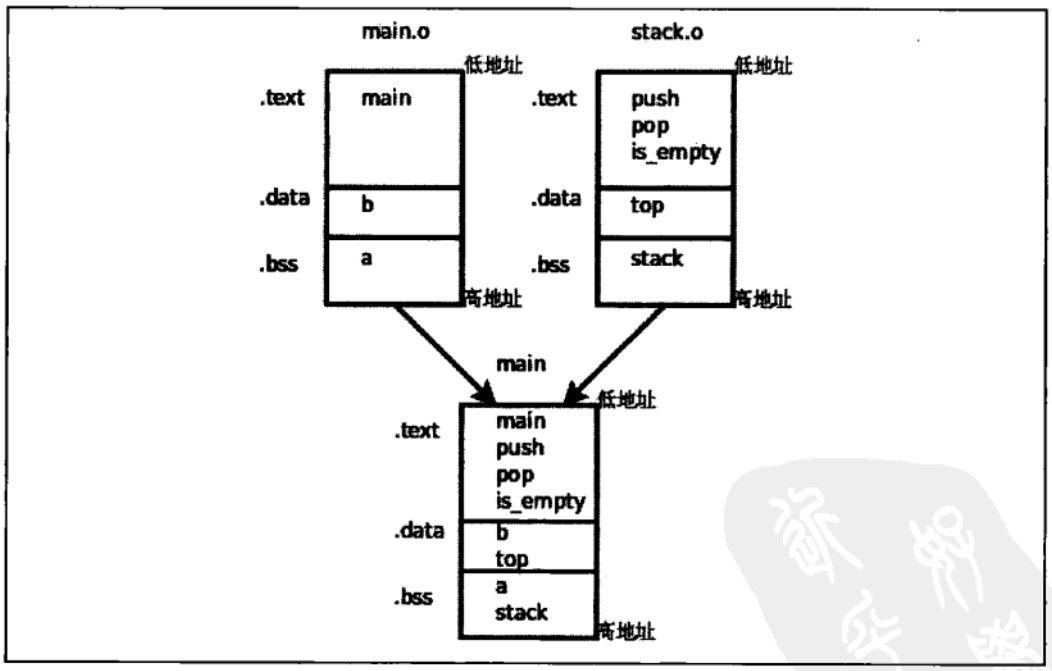


图 19.1　多目标文件的链接

为什么在可执行文件 main 的每个段中来自 main.o 的变量或函数都在前面，而来自 stack.o 的变量或函数都在后面呢？我们可以试试把 gcc 命令中的两个目标文件反过来写：

```
$ gcc stack.o main.o -o main
```

结果正如我们所预料，可执行文件 main 的每个段中来自 main.o 的变量或函数都排到后面了。实际上链接过程是由一个链接脚本（Linker Script）控制的，链接脚本决定了给每个段分配什么地址，如何对齐，哪个段在前，哪个段在后，哪些段合并到同一个 Segment。另外链接脚本还要把一些特殊地址定义成符号，例如 __bss_start 代表.bss 段的起始地址，_end 代表.bss 段的结束地址，这些符号会出现在可执行文件的符号表中，加载器可以由这些符号得知.bss 段的地址范围，以便把它清零。如果用 ld 做链接时没有通过-T 选项指定链接脚本，则使用 ld 的默认链接脚本，默认链接脚本可以用 ld --verbose 命令查看（由于比较长，只列出一些片断）：

```
$ ld --verbose
...
using internal linker script:
==================================================
/* Script for -z combreloc: combine and sort reloc sections */
OUTPUT_FORMAT("elf32-i386", "elf32-i386",
              "elf32-i386")
OUTPUT_ARCH(i386)
ENTRY(_start)
SEARCH_DIR("/usr/i486-linux-gnu/lib32");
SEARCH_DIR("/usr/local/lib32");   SEARCH_DIR("/lib32");   SEARCH_DIR
("/usr/lib32");   SEARCH_DIR("/usr/i486-linux-gnu/lib");   SEARCH_DIR
("/usr/local/lib"); SEARCH_DIR("/lib"); SEARCH_DIR("/usr/lib");
SECTIONS
{
  /* Read-only sections, merged into text segment: */
  PROVIDE  (__executable_start  =  SEGMENT_START("text-segment",
0x08048000)); . = SEGMENT_START("text-segment", 0x08048000) + SIZEOF_
HEADERS;
  .interp         : { *(.interp) }
  .note.gnu.build-id : { *(.note.gnu.build-id) }
  .hash           : { *(.hash) }
  .gnu.hash       : { *(.gnu.hash) }
  .dynsym         : { *(.dynsym) }
  .dynstr         : { *(.dynstr) }
  .gnu.version    : { *(.gnu.version) }
  .gnu.version_d  : { *(.gnu.version_d) }
  .gnu.version_r  : { *(.gnu.version_r) }
  .rel.dyn        :
...
  .rel.plt        :
...
  .init           :
...
  .plt            : { *(.plt) *(.iplt) }
  .text           :
...
  .fini           :
...
  .rodata         : { *(.rodata .rodata.* .gnu.linkonce.r.*) }
...
  .eh_frame       : ONLY_IF_RO { KEEP (*(.eh_frame)) }
...
```

```
      /* Adjust the address for the data segment.  We want to adjust up to
         the same address within the page on the next page up.  */
      . = ALIGN (CONSTANT (MAXPAGESIZE)) - ((CONSTANT (MAXPAGESIZE) - .)
& (CONSTANT (MAXPAGESIZE) - 1)); . = DATA_SEGMENT_ALIGN (CONSTANT
(MAXPAGESIZE), CONSTANT (COMMONPAGESIZE));
    ...
      .ctors          :
    ...
      .dtors          :
    ...
      .jcr            : { KEEP (*(.jcr)) }
    ...
      .dynamic        : { *(.dynamic) }
      .got            : { *(.got) *(.igot) }
    ...
      .got.plt        : { *(.got.plt)  *(.igot.plt) }
      .data           :
    ...
      _edata = .; PROVIDE (edata = .);
      __bss_start = .;
      .bss            :
    ...
      _end = .; PROVIDE (end = .);
      . = DATA_SEGMENT_END (.);
      /* Stabs debugging sections.  */
    ...
      /* DWARF debug sections.
         Symbols in the DWARF debugging sections are relative to the
beginning
         of the section so we begin them at 0.  */
    ...
    }

    ==================================================
```

ENTRY(_start)指明整个程序的入口点是_start，这并不是规定死的，修改链接脚本就可以改用其他符号做入口点。

```
      /* Read-only sections, merged into text segment: */
      PROVIDE  (__executable_start  =  SEGMENT_START("text-segment",
0x08048000));   .   =   SEGMENT_START("text-segment",   0x08048000)   +
SIZEOF_HEADERS;
      .interp         : { *(.interp) }
      .note.gnu.build-id : { *(.note.gnu.build-id) }
    ...
```

PROVIDE (__executable_start = SEGMENT_START("text-segment", 0x08048000));语句导出一个 GLOBAL 的符号__executable_start，它的值是 Text Segment 的起始地址（默认值是 0x8048000）。

再看. = SEGMENT_START("text-segment", 0x08048000) + SIZEOF_HEADERS;这一句。“.”表示当前链接地址，即程序加载运行时的虚拟地址，链接器每组装一个段就把当前链接地址自动加上这个段的长度，因此各段在加载时一般是紧挨着的，中间没有空隙，只有一种情况例外：如果在链接脚本中给“.”赋值，那么链

接器组装下一个段就从赋值的新地址开始，而不是和前一个段紧挨着了。所以这条语句表示把当前链接地址改成 Text Segment 的起始地址加上 SIZEOF_HEADERS 偏移量，后面的段从这里开始组装，后面的段依次是.interp 段、.note.gnu.build-id 段等（其中包括我们熟悉的.plt 段、.text 段和.rodata 段），这些段都被组装到 Text Segment 中。

每个段的描述格式都是“段名 :{ 组成 }”，例如.plt : { *(.plt) *(.iplt) }，左边表示链接生成的文件的.plt 段，右边表示所有目标文件的.plt 段和.iplt 段，意思是链接生成的文件的.plt 段由各目标文件的.plt 段和.iplt 段组成。

组装完 Text Segment 之后又给当前链接地址赋了新值，从新的虚拟地址开始组装 Data Segment：

```
      /* Adjust the address for the data segment.  We want to adjust up to
         the same address within the page on the next page up.  */
      . = ALIGN (CONSTANT (MAXPAGESIZE)) - ((CONSTANT (MAXPAGESIZE) - .)
& (CONSTANT (MAXPAGESIZE) - 1)); . = DATA_SEGMENT_ALIGN (CONSTANT
(MAXPAGESIZE), CONSTANT (COMMONPAGESIZE));
    ...
```

计算 Data Segment 的起始地址要做一系列对齐操作，可以结合图 17.2 来理解，Data Segment 从 Text Segment 的下一个页面开始，并且不是从该页面的起始地址开始，而是有一个偏移量。上面这两个表达式的详细计算过程我们就不深入讨论了。计算出当前地址之后，从该地址开始组装链接脚本后面列出的几个段，例如.data 段、.bss 段等。

组装完 Data Segment 之后又给当前链接地址赋了新值，从新的虚拟地址开始组装调试信息等其他 Segment：

```
      . = DATA_SEGMENT_END (.);
      /* Stabs debugging sections.  */
    ...
      /* DWARF debug sections.
         Symbols in the DWARF debugging sections are relative to the
beginning
         of the section so we begin them at 0.  */
    ...
```

关于链接脚本就介绍这么多，本书不做深入讨论，读者可以查阅参考文献[26]。

从现在开始我们写的很多程序都是由多个.c 文件编译链接在一起的，在 gdb 调试时如何指定某个.c 文件中的某一行代码呢？现在我们调试这个程序，在 push 函数和 pop 函数里设断点，注意 gdb 命令的写法。

```
    $ gcc stack.c main.c -g -o main
    $ gdb main
    GNU gdb (GDB) 7.1-ubuntu
    Copyright (C) 2010 Free Software Foundation, Inc.
    License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/
gpl.html>
```

```
    This is free software: you are free to change and redistribute it.
    There is NO WARRANTY, to the extent permitted by law.  Type "show
copying"
    and "show warranty" for details.
    This GDB was configured as "i486-linux-gnu".
    For bug reporting instructions, please see:
    <http://www.gnu.org/software/gdb/bugs/>...
    Reading symbols from /home/akaedu/main...done.
    (gdb) l
    1   /* main.c */
    2   #include <stdio.h>
    3
    4   int a, b = 1;
    5
    6   int main(void)
    7   {
    8       push('a');
    9       push('b');
    10      push('c');
    (gdb) l stack.c:1
    1   /* stack.c */
    2   char stack[512];
    3   int top = -1;
    4
    5   void push(char c)
    6   {
    7       stack[++top] = c;
    8   }
    9
    10  char pop(void)
    (gdb) b push
    Breakpoint 1 at 0x80483f0: file stack.c, line 7.
    (gdb) b stack.c:10
    Breakpoint 2 at 0x8048411: file stack.c, line 10.
    (gdb) r
    Starting program: /home/akaedu/main

    Breakpoint 1, push (c=97 'a') at stack.c:7
    7       stack[++top] = c;
    (gdb) c
    Continuing.

    Breakpoint 1, push (c=98 'b') at stack.c:7
    7       stack[++top] = c;
    (gdb) c
    Continuing.

    Breakpoint 1, push (c=99 'c') at stack.c:7
    7       stack[++top] = c;
    (gdb) c
    Continuing.

    Breakpoint 2, pop () at stack.c:12
    12      return stack[top--];
```

在 gdb 命令中指定某个.c 文件中的某一行或某个函数，可以用“文件名:行号”或“文件名:函数名”的语法。

19.2 定义和声明

19.2.1 extern 和 static 关键字

在上一节我们把两个.c 文件放在一起编译链接，main.c 用到的函数 push、pop 和 is_empty 由 stack.c 提供，其实有一点小问题，我们用-Wall 选项编译 main.c 可以看到：

```
$ gcc -c main.c -Wall
main.c: In function 'main':
main.c:8: warning: implicit declaration of function 'push'
main.c:12: warning: implicit declaration of function 'is_empty'
main.c:13: warning: implicit declaration of function 'pop'
```

这个问题我们在第 3.2 节讨论过，编译器在处理 main.c 中的函数调用时找不到函数原型，也就不知道函数的参数和返回值类型，而只能根据函数调用的实参类型做隐式声明，并假定返回值是 int 型。编译器把这三个函数隐式声明为：

```
int push(int);
int pop(void);
int is_empty(void);
```

结合上一章讲过的知识想想，为什么编译器在处理函数调用时需要知道函数原型？因为必须知道参数的类型和个数以及返回值类型才知道应该生成什么样的指令。为什么隐式声明靠不住呢？因为隐式声明是根据函数调用代码推测的，第一，函数的形参类型可能跟函数调用的实参类型不一致，第二，如果函数定义带有可变参数（例如 printf），从函数调用代码也看不出来它带可变参数，第三，从函数调用代码看不出来返回值应该是什么类型，隐式声明只能假定返回值都是 int 型。既然隐式声明靠不住，那编译器为什么不自己去找函数定义，非要我们在调用之前提供函数原型呢？因为编译器不知道去哪里找函数定义，像上面的例子，我让编译器编译 main.c，而这几个函数的定义却在 stack.c 里，编译器怎么会知道呢？所以编译器只能通过隐式声明来推测函数原型，这种推测往往是错的，但在比较简单的情况下还算可用，比如上一节的例子这么编译过去了也能得到正确结果。

现在我们在 main.c 中声明这几个函数的原型：

```
/* main.c */
#include <stdio.h>

extern void push(char);
extern char pop(void);
extern int is_empty(void);

int main(void)
{
        push('a');
        push('b');
        push('c');

        while(!is_empty())
```

```
            putchar(pop());
        putchar('\n');

        return 0;
}
```

这样编译器就不会报警告了。用 extern 关键字修饰的函数名具有 External Linkage。External Linkage 的定义在上一章讲过，现在应该更容易理解了，push 这个标识符具有 External Linkage 指的是：push 在 main.c 和 stack.c 中都有声明（stack.c 中的声明同时也是定义），如果把 main.c 和 stack.c 链接在一起，那么这些声明指的是同一个函数，在链接时各目标文件中的全局符号 push 代表同一个地址。函数声明中的 extern 关键字也可以省略不写。

用 static 关键字修饰的函数名具有 Internal Linkage。例如有以下两个.c 文件：

```
/* foo.c */
static void foo(void) {}
/* main.c */
void foo(void);
int main(void) { foo(); return 0; }
```

编译链接在一起会出错：

```
$ gcc foo.c main.c
/tmp/ccRC2Yjn.o: In function `main':
main.c:(.text+0x12): undefined reference to `foo'
collect2: ld returned 1 exit status
```

虽然 foo.c 中定义了函数 foo，但这个函数名只有 Internal Linkage，只有在 foo.c 中引用这个函数名才表示同一个函数，而在 main.c 中声明的那个 foo 应该表示另一个具有 External Linkage 的函数名。如果把 foo.c 编译成目标文件，函数名 foo 在其中是一个局部符号，在链接时不参与符号解析。所以，在 main.c 中引用了一个具有 External Linkage 的函数名 foo，但链接器却找不到它的定义在哪儿，无法确定它的地址，只好报错。**凡是被多次声明的变量或函数，必须有且只有一个声明是定义，如果有多个定义，或者一个定义都没有，链接器就无法完成链接。**

以上讲了用 static 和 extern 修饰函数声明的情况，现在来看用它们修饰变量声明的情况。仍然用 stack.c 和 main.c 的例子，如果我想在 main.c 中直接访问 stack.c 中定义的变量 top，可以用 extern 声明它：

```
/* main.c */
#include <stdio.h>

void push(char);
char pop(void);
int is_empty(void);
extern int top;

int main(void)
{
        push('a');
        push('b');
```

```
        push('c');
        printf("%d\n", top);

        while(!is_empty())
                putchar(pop());
        putchar('\n');
        printf("%d\n", top);

        return 0;
}
```

变量top具有External Linkage，extern int top;只是一个声明而不是定义，因为它不在main.c中分配存储空间，而是在stack.c中定义和分配存储空间，main.c只是引用这个变量名。以上函数和变量声明也可以写在main函数体里面，使所声明的标识符具有块作用域：

```
int main(void)
{
        void push(char);
        char pop(void);
        int is_empty(void);
        extern int top;

        push('a');
        push('b');
        push('c');
        printf("%d\n", top);

        while(!is_empty())
                putchar(pop());
        putchar('\n');
        printf("%d\n", top);

        return 0;
}
```

注意，变量声明和函数声明有一点不同，函数声明的extern关键字可以省略，而变量声明如果不写extern意思就完全变了，如果上面的例子不写extern就表示在main函数中定义一个局部变量top。另外要注意，变量定义可以初始化而声明不可以，stack.c中的定义可以写成int top = -1;，而main.c中的声明却不能写成extern int top = -1;，否则编译器会报错。

在main.c中可以通过extern声明来访问stack.c中的变量top，但是从实现stack.c这个模块的角度来看，top这个变量是不希望被外界访问到的，变量top和stack都属于这个模块的内部状态，外界应该只允许通过push、pop函数来改变模块的内部状态，这样才能保证堆栈的LIFO访问特性，如果外界可以随机访问stack或者随便修改top，那么堆栈的状态就乱了。怎么才能阻止外界访问top和stack呢？答案就是用static关键字把它们声明为Internal Linkage的：

```
/* stack.c */
static char stack[512];
static int top = -1;
```

```
void push(char c)
{
        stack[++top] = c;
}

char pop(void)
{
        return stack[top--];
}

int is_empty(void)
{
        return top == -1;
}
```

这样，即使在 main.c 中用 extern 声明也访问不到 stack.c 的变量 top 和 stack，从而保护了 stack.c 模块的内部状态，这也是一种封装。

用 static 关键字声明具有 Internal Linkage 的函数也是出于这个目的。在一个模块中，有些函数是提供给外界使用的，或者说导出（Export）给外界使用，这些函数声明为 External Linkage 的。有些函数只在模块内部使用而不希望被外界访问到，则声明为 Internal Linkage 的。

19.2.2　头文件

我们继续前面关于 stack.c 和 main.c 的讨论。stack.c 这个模块封装了 top 和 stack 两个变量，导出了 push、pop、is_empty 三个函数接口，已经设计得比较完善了。但是使用这个模块的每个.c 文件都要写三个函数声明也是很麻烦的，假设又有一个 foo.c 也使用这个模块，main.c 和 foo.c 中各自要写三个函数声明。重复的代码总是应该尽量避免的，比如在第 8.2 节讲过用宏定义避免硬编码。要避免写重复的声明也有办法，可以自己写一个头文件 stack.h：

```
/* stack.h */
#ifndef STACK_H
#define STACK_H
extern void push(char);
extern char pop(void);
extern int is_empty(void);
#endif
```

main.c 和 foo.c 都可以包含这个头文件，就相当于声明了这三个函数。比如 main.c 可以改成这样：

```
/* main.c */
#include <stdio.h>
#include "stack.h"

int main(void)
{
        push('a');
        push('b');
        push('c');
```

```
        while(!is_empty())
                putchar(pop());
        putchar('\n');

        return 0;
}
```

首先说为什么#include <stdio.h>用角括号，而#include "stack.h"用引号。对于用角括号包含的头文件，gcc 首先查找-I 选项指定的目录，然后查找系统的头文件目录（在我的系统上是按/usr/local/include、/usr/lib/gcc/i486-linux-gnu/4.4.3/include、/usr/i486-linux-gnu/include、/usr/include 的顺序依次查找）；而对于用引号包含的头文件，gcc 首先查找正在被处理的#include 指示所在的当前文件所在的目录，然后查找-I 选项指定的目录，然后查找系统的头文件目录。

假如三个代码文件都放在当前目录下：

```
$ tree
.
|-- main.c
|-- stack.c
`-- stack.h

0 directories, 3 files
```

则可以用 gcc -c main.c 编译，gcc 会自动在 main.c 所在的目录中找到 stack.h。假如把 stack.h 移到一个子目录下：

```
$ tree
.
|-- main.c
`-- stack
    |-- stack.c
    `-- stack.h

1 directory, 3 files
```

则需要用 gcc -c main.c -Istack 编译，用-I 选项告诉 gcc 头文件要到子目录 stack 里找。

在#include 预处理指示中可以使用相对路径，例如把上面的代码改成#include "stack/stack.h"，那么编译时就不需要加-Istack 选项了，因为是 main.c 要包含头文件，gcc 会自动在 main.c 所在的目录中查找，而头文件相对于 main.c 所在目录的相对路径正是 stack/stack.h。

注意，-I 选项可以指定相对路径也可以指定绝对路径，如果指定相对路径，它是相对于 gcc 进程的当前工作目录的路径，而不是相对于正在被处理的#include 指示所在的当前文件的路径。

在这里先解释一下当前工作目录（Current Working Directory）的概念。每个进程都有自己的当前工作目录，Shell 进程的当前工作目录可以用 pwd 命令查看：

```
$ pwd
/home/akaedu
```

通常 Linux 发行版缺省配置的 Shell 提示符会显示当前工作目录，例如~$表示当前工作目录是主目录，/etc$表示当前工作目录是/etc 目录。用 cd 命令可以改变 Shell 进程的当前工作目录。如果在 Shell 下敲命令启动新进程（比如 gcc），则新进程会继承 Shell 的当前工作目录，新进程也可以调用 chdir(2)改变自己的当前工作目录。

下面再举个例子来加深理解。比如有以下目录结构：

```
$ tree
.
`-- foo
    |-- bar.h
    `-- foo.h

1 directory, 2 files
```

在 foo.h 中有一行代码#include "bar.h"，用 gcc -E 命令可以查看预处理的结果：

```
$ gcc -E foo/foo.h
# 1 "foo/foo.h"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "foo/foo.h"
# 1 "foo/bar.h" 1
content of bar.h
# 1 "foo/foo.h" 2
```

如果在 foo.h 中有一行代码#incldue <bar.h>，则用 gcc -E 命令做预处理会出错，因为 gcc 不会查找 foo.h 所在的目录：

```
$ gcc -E foo/foo.h
# 1 "foo/foo.h"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "foo/foo.h"
foo/foo.h:1:17: error: bar.h: No such file or directory
```

指定了-I.选项也不管用，因为 gcc 的当前工作目录是 bar.h 的上一层目录：

```
$ gcc -E -I. foo/foo.h
# 1 "foo/foo.h"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "foo/foo.h"
foo/foo.h:1:17: error: bar.h: No such file or directory
```

如果先 cd 到 foo 子目录中再指定-I.选项就管用了，这时 gcc 的当前工作目录正是 bar.h 所在的目录：

```
$ cd foo/
$ gcc -E foo.h
# 1 "foo.h"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "foo.h"
foo.h:1:17: error: bar.h: No such file or directory
$ gcc -E -I. foo.h
```

```
# 1 "foo.h"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "foo.h"
# 1 "./bar.h" 1
content of bar.h
# 1 "foo.h" 2
```

回到正题。在 stack.h 中我们又看到两个新的预处理指示#ifndef STACK_H 和#endif，意思是说，如果 STACK_H 这个宏没有定义过，那么从#ifndef 到#endif 之间的代码就包含在预处理的输出结果中，否则这一段代码就不出现在预处理的输出结果中。stack.h 这个头文件的内容全部被#ifndef 和#endif 括起来了，如果在包含这个头文件时 STACK_H 这个宏已经定义过了，则相当于这个头文件里什么都没有，包含了一个空文件。这有什么用呢？假如 main.c 包含了两次 stack.h：

```
...
#include "stack.h"
#include "stack.h"

int main(void)
{
...
```

则第一次包含 stack.h 时并没有定义 STACK_H 这个宏，因此头文件的内容包含在预处理的输出结果中：

```
...
#define STACK_H
extern void push(char);
extern char pop(void);
extern int is_empty(void);
#include "stack.h"

int main(void)
{
...
```

其中已经定义了 STACK_H 这个宏，第二次再包含 stack.h 就相当于包含了一个空文件，这就避免了头文件的内容被重复包含。这种保护头文件的写法称为 Header Guard，以后我们写的每个头文件都要加上 Header Guard，宏定义名就用头文件名的大写形式，这是规范的做法。

为什么需要防止重复包含呢？谁会把一个头文件包含两次呢？像上面这样明显的错误没人会犯，但有时候重复包含的错误并不是那么明显的。比如：

```
#include "stack.h"
#include "foo.h"
```

foo.h 里又包含了 bar.h，bar.h 里又包含了 stack.h。在规模较大的项目中头文件包含头文件的情况很常见，经常会包含四五层，这时候重复包含的问题就很难发现了。比如在我的系统头文件目录/usr/include 中，errno.h 包含了 bits/errno.h，后者又包含了 linux/errno.h，后者又包含了 asm/errno.h，后者又包含了 asm-generic/errno.h。

另外一个问题是，就算我重复包含了头文件，那有什么危害么？像上面的三个函数声明，在一个编译单元中多出现几次也没有错。重复包含头文件主要有以下问题：

1．预处理和编译的速度变慢了，要处理很多本来不需要处理的代码。

2．如果不小心出现 foo.h 包含 bar.h、bar.h 又包含 foo.h 的情况，就陷入死循环了。其实一般编译器都会规定一个包含层数的上限，超过这个上限就报错。

3．头文件里有些代码不允许重复出现，虽然变量和函数允许多次声明（只要不是多次定义就行），但头文件里有些代码是不允许多次出现的，比如用 typedef 定义一个类型名，在一个编译单元中只允许定义一次。

还有一个问题，既然要#include 头文件，那我不如直接在 main.c 中#include "stack.c" 得了。这样把 stack.c 和 main.c 合并成一个编译单元，相当于又回到最初例 12.1 的代码了，用 gcc main.c -o main 也能编译通过。这样不是更简单吗？连头文件都不用写了。

假如又有一个 foo.c 也要用 stack.c 这个模块怎么办呢？如果在 foo.c 里面也#include "stack.c"，就相当于 push、pop、is_empty 这三个函数在 main.c 和 foo.c 的编译单元中都有定义，那么 main.c 和 foo.c 就不能链接在一起了。如果采用包含头文件的办法，这三个函数只在 stack.c 中定义一次，在 main.c 和 foo.c 中只是声明，就可以把 main.c、stack.c、foo.c 三个编译单元链接在一起，如图 19.2 所示。

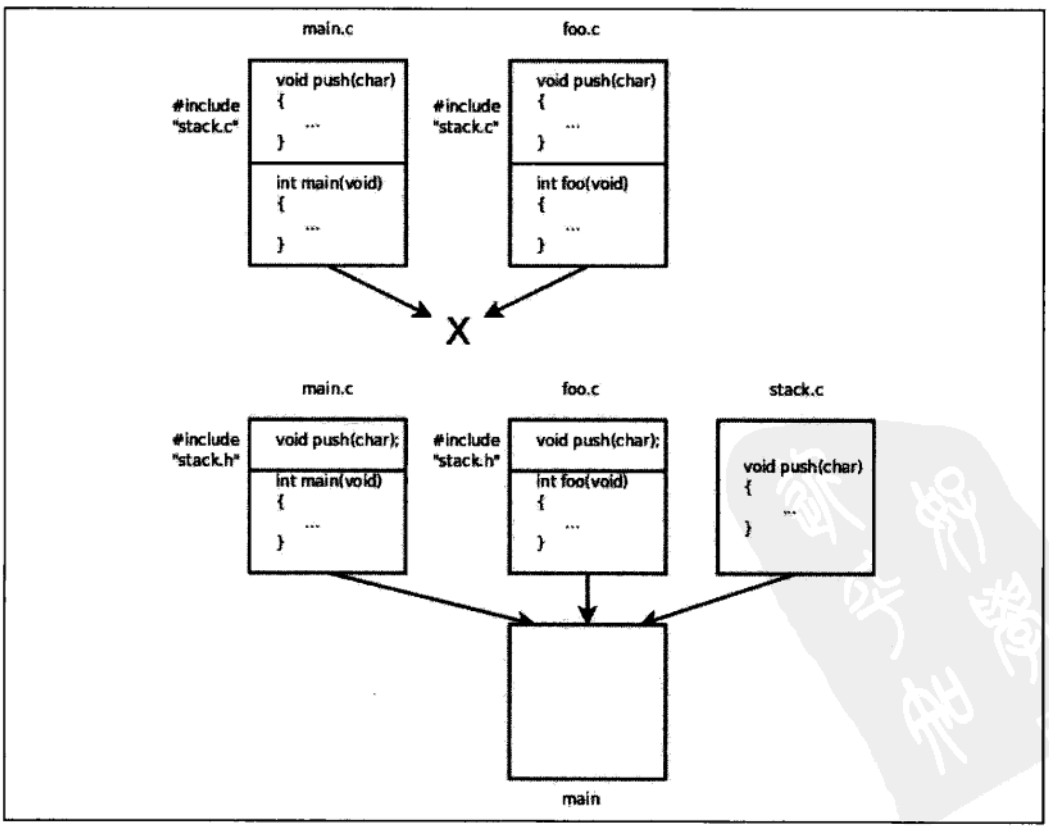


图 19.2　为什么要包含头文件而不是.c 文件

一般来说，应遵循以下原则：

1. .c 文件中可以有变量或函数定义，而.h 文件中应该只有变量或函数声明而没有定义。

2. 不要把一个.c 文件包含到另一个.c 文件中。

“二般”来说，也有些特别的 Convention 不遵循这两条原则：有些程序员喜欢在.h 文件中定义 static inline 函数，也有些情况下确实会在一个.c 文件中 include 另一个.c 文件（例如在单元测试代码中 include 被测代码），在此不深入探讨这些问题。

19.2.3 定义和声明的详细规则

以上两节关于变量、函数的定义和声明只介绍了最基本的规则，在写代码时掌握这些基本规则就够用了，但其实 C 语言关于定义和声明还有很多复杂的规则，在维护别人的代码时还是有必要了解这些规则的。首先看关于函数声明的规则，表 19.1 和表 19.2 出自参考文献[4]，我做了一些修改。

表 19.1 Storage Class 关键字对函数声明的作用

Storage Class	File Scope Declaration	Block Scope Declaration
none	previous linkage can define	previous linkage cannot define
extern	previous linkage can define	previous linkage cannot define
static	internal linkage can define	N/A

前面我说“extern 关键字表示这个标识符具有 External Linkage”其实是不准确的，准确地说应该是 Previous Linkage。Previous Linkage 的定义是：当前声明的这个标识符具有什么样的 Linkage 取决于该编译单元中前面对这个标识符的声明（而且必须是文件作用域的声明），如果在前面找不到这个标识符的声明，即当前声明是该编译单元中对这个标识符的第一次声明，那么这个标识符具有 External Linkage。例如在一个编译单元中在文件作用域两次声明同一个函数：

```
static int f(void); /* internal linkage */
extern int f(void); /* previous linkage */
```

则这里的 extern 修饰的标识符 f 具有 Interanl Linkage 而不是 External Linkage。从表 19.1 的前两行可以总结出我们先前所说的规则——函数声明中的 extern 关键字可以省略不写。表 19.1 也说明了在文件作用域允许定义函数，而在块作用域不允许定义函数，或者说函数定义不能嵌套。另外，在块作用域中不允许用 static 关键字声明函数。

关于变量声明的规则要复杂一些，如表 19.2 所示。

表 19.2　Storage Class 关键字对变量声明的作用

Storage Class	File Scope Declaration	Block Scope Declaration
none	external linkage static duration static initializer definition or tentative definition	no linkage automatic duration dynamic initializer definition
extern	previous linkage static duration no initializer[*] not a definition[*]	previous linkage static duration no initializer not a definition
static	internal linkage static duration static initializer definition or tentative definition	no linkage static duration static initializer definition

表 19.2 的每个单元格里分成四行，分别描述变量的链接属性、变量的生存期、这种变量如何初始化，以及这种声明是否算变量定义。链接属性有 External Linkage、Internal Linkage、No Linkage 和 Previous Linkage 四种情况，生存期有 Static Duration 和 Automatic Duration 两种情况。初始化有 Static Initializer 和 Dynamic Initializer 两种情况，前者表示 Initializer 中只能使用常量表达式，表达式的值必须在编译时确定，后者表示 Initializer 中可以使用任意右值表达式，表达式的值可以在运行时计算。

是否算变量定义有三种情况，Definition、Not a Definition 和 Tentative Definition。前面我说“有 extern 的变量声明不是定义，没有 extern 的才是定义，变量定义可以初始化而声明不可以”，其实也不准确。C 标准是这么规定的：有初始化的变量声明是定义；没有初始化的变量声明如果加了 extern 修饰则属于 Previous Linkage，这种声明不是定义；如果没加 extern 修饰也没有初始化则属于 Tentative Definition。什么叫 Tentative Definition 呢？如果一个变量声明具有文件作用域，没有初始化，没有用 Storage Class 关键字修饰，或者用 static 关键字修饰，那么编译器认为这个变量是在该编译单元中定义的，但初始值待定，然后继续编译下面的代码，到整个编译单元编译结束时如果没有遇到这个变量的带初始化的定义，就用 0 来初始化它。在参考文献[8]的 6.9.2 节有一个例子：

```
int i1 = 1; // definition, external linkage
static int i2 = 2; // definition, internal linkage
extern int i3 = 3; // definition, external linkage
int i4; // tentative definition, external linkage
static int i5; // tentative definition, internal linkage
int i1; // valid tentative definition, refers to previous
int i2; // 6.2.2 renders undefined, linkage disagreement
int i3; // valid tentative definition, refers to previous
int i4; // valid tentative definition, refers to previous
int i5; // 6.2.2 renders undefined, linkage disagreement
extern int i1; // refers to previous, whose linkage is external
```

```
extern int i2; // refers to previous, whose linkage is internal
extern int i3; // refers to previous, whose linkage is external
extern int i4; // refers to previous, whose linkage is external
extern int i5; // refers to previous, whose linkage is internal
```

变量 i2 和 i5 第一次声明为 Internal Linkage，第二次又声明为 External Linkage，这是不允许的，编译器会报错。

注意表 19.2 中标有[*]的单元格，对于文件作用域的 extern 变量声明，C99 是允许带 Initializer 的，并且认为它是一个定义，但是 gcc 对于这种写法会报警告，为了兼容性应避免这种写法。gcc 对于 Tentative Definition 的处理也和 C99 的规定不一致。比如在 foo.c 中用 int i;定义一个变量 i，是 Tentative Definition，C99 规定这个变量应该在 foo.c 中定义，初始值是 0，而 gcc 编译的结果是：

```
$ gcc -c foo.c
$ nm foo.o
00000004 C i
```

符号 i 的类型是 Common，nm(1)中说 Common 符号的定义在链接时确定。如果 bar.c 中定义 int i = 1;，则 foo.c 和 bar.c 链接在一起时 foo.c 的那个只能算声明而不算定义。如果 bar.c 中也定义 int i;，也是一个 Tentative Definition，则 foo.c 和 bar.c 链接在一起时才定义变量 i，并且用 0 初始化。也就是说，C99 对 Tentative Definition 的处理是在编译一个单元时做，而 gcc 是推迟到链接时才做。如果编译时加上 -fno-common 选项则不会生成 Common 符号，gcc 对 Tentative Definition 的处理就和 C99 一致了。

19.3 静态库

有时候需要把一组代码编译成一个库，这个库在很多项目中都要用到，例如 libc 就是这样一个库，我们在不同的程序中都会用到 libc 中定义的库函数（例如 printf(3)）和全局变量（例如 environ(3)）。本节和下一节介绍怎么创建这样的库。

我们继续用 stack.c 的例子。为了便于理解，我们把 stack.c 拆成四个.c 文件（虽然实际上没太大必要），把 main.c 改得简单一些，头文件 stack.h 不变，本节用到的代码如下所示：

```
/* stack.c */
char stack[512];
int top = -1;
/* push.c */
extern char stack[512];
extern int top;

void push(char c)
{
        stack[++top] = c;
}
/* pop.c */
extern char stack[512];
extern int top;
```

```
char pop(void)
{
        return stack[top--];
}
/* is_empty.c */
extern int top;

int is_empty(void)
{
        return top == -1;
}
/* stack.h */
#ifndef STACK_H
#define STACK_H
extern void push(char);
extern char pop(void);
extern int is_empty(void);
#endif
/* main.c */
#include <stdio.h>
#include "stack.h"

int main(void)
{
        push('a');
        return 0;
}
```

这些文件的目录结构是：

```
$ tree
.
|-- main.c
`-- stack
    |-- is_empty.c
    |-- pop.c
    |-- push.c
    |-- stack.c
    `-- stack.h

1 directory, 6 files
```

我们把 stack.c、push.c、pop.c、is_empty.c 编译成目标文件：

```
$ gcc -c stack/stack.c stack/push.c stack/pop.c stack/is_empty.c
```

然后打包成一个静态库 libstack.a：

```
$ ar rs libstack.a stack.o push.o pop.o is_empty.o
ar: creating libstack.a
```

库文件名都是以 lib 开头的，静态库以.a 作为后缀，表示 Archive。ar 命令类似于 tar 命令，也是用来打包的，但是把目标文件打包成静态库的格式只能用 ar 命令而不能用 tar 命令。选项 r 表示将后面的目标文件列表添加到文件包 libstack.a，如果 libstack.a 不存在就创建它，如果 libstack.a 中已有同名的目标文件就替换成新的。选项 s 表示为静态库创建索引，这个索引被链接器使用。ranlib 命令也可以为

静态库创建索引，以上命令等价于：

```
$ ar r libstack.a stack.o push.o pop.o is_empty.o
$ ranlib libstack.a
```

然后我们把 libstack.a 和 main.c 编译链接在一起：

```
$ gcc main.c -L. -lstack -Istack -o main
```

-L 选项告诉编译器去哪里找需要的库文件，-L.表示在当前目录找。-lstack 选项告诉编译器要链接 libstack 库，-I 选项告诉编译器去哪里找头文件。注意，即使库文件就在当前目录，编译器默认也不会去找的，所以-L.选项不能少。编译器默认会找哪些目录呢？可以用-print-search-dirs 选项查看：

```
$ gcc -print-search-dirs
install: /usr/lib/gcc/i486-linux-gnu/4.4.3/
programs: =/usr/lib/gcc/i486-linux-gnu/4.4.3/:/usr/lib/gcc/i486-
linux-gnu/4.4.3/:/usr/lib/gcc/i486-linux-gnu/:/usr/lib/gcc/i486-lin
ux-gnu/4.4.3/:/usr/lib/gcc/i486-linux-gnu/:/usr/libexec/gcc/i486-li
nux-gnu/4.4.3/:/usr/libexec/gcc/i486-linux-gnu/:/usr/lib/gcc/i486-l
inux-gnu/4.4.3/:/usr/lib/gcc/i486-linux-gnu/:/usr/lib/gcc/i486-linu
x-gnu/4.4.3/../../../../i486-linux-gnu/bin/i486-linux-gnu/4.4.3/:/u
sr/lib/gcc/i486-linux-gnu/4.4.3/../../../../i486-linux-gnu/bin/
libraries: =/usr/lib/gcc/i486-linux-gnu/4.4.3/:/usr/lib/gcc/i486-
linux-gnu/4.4.3/:/usr/lib/gcc/i486-linux-gnu/4.4.3/../../../../i486
-linux-gnu/lib/i486-linux-gnu/4.4.3/:/usr/lib/gcc/i486-linux-gnu/4.
4.3/../../../../i486-linux-gnu/lib/../lib/:/usr/lib/gcc/i486-linux-
gnu/4.4.3/../../../i486-linux-gnu/4.4.3/:/usr/lib/gcc/i486-linux-gn
u/4.4.3/../../../../lib/:/lib/i486-linux-gnu/4.4.3/:/lib/../lib/:/u
sr/lib/i486-linux-gnu/4.4.3/:/usr/lib/../lib/:/usr/lib/i486-linux-g
nu/i486-linux-gnu/4.4.3/:/usr/lib/i486-linux-gnu/../lib/:/usr/lib/g
cc/i486-linux-gnu/4.4.3/../../../../i486-linux-gnu/lib/:/usr/lib/gc
c/i486-linux-gnu/4.4.3/../../../:/lib/:/usr/lib/:/usr/lib/i486-linu
x-gnu/
```

其中的 libraries 就是库文件的搜索路径列表，各路径之间用:号隔开。在处理-lstack 选项时，gcc 首先到-L 选项指定的目录下查找，首先看有没有共享库 libstack.so，如果有就链接它，否则再找有没有静态库 libstack.a，如果有就链接它，如果还是没有，就到默认搜索路径下按同样的步骤查找。gcc 在链接时优先考虑共享库，其次才是静态库，如果希望 gcc 只考虑静态库，可以指定-static 选项。

那么链接共享库和链接静态库有什么区别呢？在第 18.2 节我们看到，链接 libc 共享库时，链接器只是确认可执行文件 main 引用的某些符号在 libc 中有定义，并没有最终确定这些符号的地址，这些符号在可执行文件 main 中仍然是未定义符号，要在运行时做动态链接。而链接静态库时，链接器会把静态库中的目标文件取出来和可执行文件真正链接在一起。我们反汇编查看上一步生成的可执行文件 main：

```
$ objdump -d main
...
080483b4 <main>:
 80483b4:   55                      push   %ebp
 80483b5:   89 e5                   mov    %esp,%ebp
```

```
 80483b7:   83 e4 f0             and    $0xfffffff0,%esp
 80483ba:   83 ec 10             sub    $0x10,%esp
...
080483d0 <push>:
 80483d0:   55                   push   %ebp
 80483d1:   89 e5                mov    %esp,%ebp
 80483d3:   83 ec 04             sub    $0x4,%esp
...
```

有意思的是，main.c 只调用了 push 这一个函数，所以链接生成的可执行文件中也只有 push 而没有 pop 和 is_empty。这是使用静态库的一个好处，链接器从静态库中只取出需要的目标文件来做链接，不需要的目标文件可以不链接。如果直接把那些目标文件和 main.c 编译链接在一起：

```
$ gcc main.c stack.o push.o pop.o is_empty.o -Istack -o main
```

则没有用到的函数也会链接进来。使用静态库的另一个好处是只需写一个库文件名，而不需要写一长串目标文件名。

19.4　共享库

19.4.1　编译、链接、运行

组成共享库的目标文件和一般的目标文件有所不同，在编译时要加-fPIC 选项，例如：

```
$ gcc -c -fPIC stack/stack.c stack/push.c stack/pop.c
stack/is_empty.c
```

-f 后面跟一些编译选项，PIC 是其中一种，表示生成位置无关代码（Position Independent Code）。那么用-fPIC 生成的目标文件和一般的目标文件有什么不同呢？下面分析这个问题。

我们知道目标文件也称为 Relocatable，在链接时可以把目标文件中各段的地址做重定位，重定位需要修改指令中的地址。我们先不加-fPIC 选项编译生成目标文件：

```
$ gcc -c -g stack/stack.c stack/push.c stack/pop.c stack/is_empty.c
```

由于接下来要用 objdump -dS 把反汇编指令和源代码穿插起来分析，所以用-g 选项加调试信息。注意，必须在编译每个目标文件时加-g 选项，而不能只在最后链接时加-g 选项，如果目标文件中没有调试信息，链接生成的可执行文件也不会有。反汇编查看 push.o：

```
$ objdump -dS push.o

push.o:     file format elf32-i386


Disassembly of section .text:
```

```
00000000 <push>:
/* push.c */
extern char stack[512];
extern int top;

void push(char c)
{
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   83 ec 04                sub    $0x4,%esp
   6:   8b 45 08                mov    0x8(%ebp),%eax
   9:   88 45 fc                mov    %al,-0x4(%ebp)
     stack[++top] = c;
   c:   a1 00 00 00 00          mov    0x0,%eax
  11:   83 c0 01                add    $0x1,%eax
  14:   a3 00 00 00 00          mov    %eax,0x0
  19:   a1 00 00 00 00          mov    0x0,%eax
  1e:   0f b6 55 fc             movzbl-0x4(%ebp),%edx
  22:   88 90 00 00 00 00       mov    %dl,0x0(%eax)
}
  28:   c9                      leave
  29:   c3                      ret
```

指令中凡是用到 stack 和 top 的地址都用 0x0 表示，准备在重定位时修改。再看 readelf 输出的.rel.text 段的信息：

```
Relocation section '.rel.text' at offset 0x840 contains 4 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
0000000d  00001001 R_386_32          00000000   top
00000015  00001001 R_386_32          00000000   top
0000001a  00001001 R_386_32          00000000   top
00000024  00001101 R_386_32          00000000   stack
```

标出了指令中有四处需要在重定位时修改。编译链接成可执行文件之后再做反汇编分析：

```
$ gcc -g main.c stack.o push.o pop.o is_empty.o -Istack -o main
$ objdump -dS main
...
080483d0 <push>:
/* push.c */
extern char stack[512];
extern int top;

void push(char c)
{
 80483d0:   55                      push   %ebp
 80483d1:   89 e5                   mov    %esp,%ebp
 80483d3:   83 ec 04                sub    $0x4,%esp
 80483d6:   8b 45 08                mov    0x8(%ebp),%eax
 80483d9:   88 45 fc                mov    %al,-0x4(%ebp)
    stack[++top] = c;
 80483dc:   a1 10 a0 04 08          mov    0x804a010,%eax
 80483e1:   83 c0 01                add    $0x1,%eax
 80483e4:   a3 10 a0 04 08          mov    %eax,0x804a010
 80483e9:   a1 10 a0 04 08          mov    0x804a010,%eax
 80483ee:   0f b6 55 fc             movzbl -0x4(%ebp),%edx
 80483f2:   88 90 40 a0 04 08       mov    %dl,0x804a040(%eax)
```

```
}
 80483f8:   c9                  leave
 80483f9:   c3                  ret
...
```

原来指令中的 0x0 被改成了 0x804a010 和 0x804a040，这样做了重定位之后，各段的加载地址就定死了，因为在指令中使用了绝对地址。

我们看看用-fPIC 选项编译生成的目标文件有什么不同：

```
    $ gcc -c -g -fPIC stack/stack.c stack/push.c stack/pop.c stack/is_
empty.c
    $ objdump -dS push.o

    push.o:     file format elf32-i386


    Disassembly of section .text:

    00000000 <push>:
    /* push.c */
    extern char stack[512];
    extern int top;

    void push(char c)
    {
       0:   55                      push   %ebp
       1:   89 e5                   mov    %esp,%ebp
       3:   53                      push   %ebx
       4:   83 ec 04                sub    $0x4,%esp
       7:   e8 fc ff ff ff          call   8 <push+0x8>
       c:   81 c3 02 00 00 00       add    $0x2,%ebx
      12:   8b 45 08                mov    0x8(%ebp),%eax
      15:   88 45 f8                mov    %al,-0x8(%ebp)
        stack[++top] = c;
      18:   8b 83 00 00 00 00       mov    0x0(%ebx),%eax
      1e:   8b 00                   mov    (%eax),%eax
      20:   8d 50 01                lea    0x1(%eax),%edx
      23:   8b 83 00 00 00 00       mov    0x0(%ebx),%eax
      29:   89 10                   mov    %edx,(%eax)
      2b:   8b 83 00 00 00 00       mov    0x0(%ebx),%eax
      31:   8b 00                   mov    (%eax),%eax
      33:   8b 93 00 00 00 00       mov    0x0(%ebx),%edx
      39:   0f b6 4d f8             movzbl -0x8(%ebp),%ecx
      3d:   88 0c 02                mov    %cl,(%edx,%eax,1)
    }
      40:   83 c4 04                add    $0x4,%esp
      43:   5b                      pop    %ebx
      44:   5d                      pop    %ebp
      45:   c3                      ret

    Disassembly of section .text.__i686.get_pc_thunk.bx:

    00000000 <__i686.get_pc_thunk.bx>:
       0:   8b 1c 24                mov    (%esp),%ebx
       3:   c3                      ret
```

指令中用到的 stack 和 top 的地址不再以 0x0 表示，而是以 0x0(%ebx)表示，但其

中还是留有0x0准备做进一步修改。再看readelf输出的.rel.text段的信息：

```
Relocation section '.rel.text' at offset 0x950 contains 6 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
00000008  00001202 R_386_PC32     00000000   __i686.get_pc_thunk.bx
0000000e  0000130a R_386_GOTPC    00000000   _GLOBAL_OFFSET_TABLE_
0000001a  00001403 R_386_GOT32    00000000   top
00000025  00001403 R_386_GOT32    00000000   top
0000002d  00001403 R_386_GOT32    00000000   top
00000035  00001503 R_386_GOT32    00000000   stack
```

top和stack对应的记录类型不再是R_386_32，而是R_386_GOT32，有什么区别呢？我们先编译生成共享库再做反汇编分析：

```
$ gcc -shared -o libstack.so stack.o push.o pop.o is_empty.o
$ objdump -dS libstack.so
...
000004bc <push>:
/* push.c */
extern char stack[512];
extern int top;

void push(char c)
{
 4bc:   55                      push   %ebp
 4bd:   89 e5                   mov    %esp,%ebp
 4bf:   53                      push   %ebx
 4c0:   83 ec 04                sub    $0x4,%esp
 4c3:   e8 ef ff ff ff          call   4b7 <__i686.get_pc_thunk.bx>
 4c8:   81 c3 2c 1b 00 00       add    $0x1b2c,%ebx
 4ce:   8b 45 08                mov    0x8(%ebp),%eax
 4d1:   88 45 f8                mov    %al,-0x8(%ebp)
    stack[++top] = c;
 4d4:   8b 83 f4 ff ff ff       mov    -0xc(%ebx),%eax
 4da:   8b 00                   mov    (%eax),%eax
 4dc:   8d 50 01                lea    0x1(%eax),%edx
 4df:   8b 83 f4 ff ff ff       mov    -0xc(%ebx),%eax
 4e5:   89 10                   mov    %edx,(%eax)
 4e7:   8b 83 f4 ff ff ff       mov    -0xc(%ebx),%eax
 4ed:   8b 00                   mov    (%eax),%eax
 4ef:   8b 93 f8 ff ff ff       mov    -0x8(%ebx),%edx
 4f5:   0f b6 4d f8             movzbl -0x8(%ebp),%ecx
 4f9:   88 0c 02                mov    %cl,(%edx,%eax,1)
}
 4fc:   83 c4 04                add    $0x4,%esp
 4ff:   5b                      pop    %ebx
 500:   5d                      pop    %ebp
 501:   c3                      ret
...
```

和先前的结果不同，指令中的0x0(%ebx)被修改成-0xc(%ebx)和-0x8(%ebx)，而不是修改成绝对地址。所以共享库各段的加载地址并没有定死，可以加载到任意位置，因为指令中的地址都是相对于ebx的，没有使用绝对地址，只要根据实际的加载情况修改ebx就可以了，这就是位置无关代码的特点。另外，注意这几条指令：

```
4d4:   8b 83 f4 ff ff ff       mov    -0xc(%ebx),%eax
```

```
4da:    8b 00                mov    (%eax),%eax
```

和先前的指令对比一下：

```
80483dc:    a1 10 a0 04 08          mov    0x804a010,%eax
```

可以发现，-0xc(%ebx)这个地址并不是变量 top 的地址，这个地址的内存单元中又保存了另外一个地址，这另外一个地址才是变量 top 的地址。指令 mov -0xc(%ebx),%eax 是从地址 ebx-12 取出变量 top 的地址传给 eax，而指令 mov (%eax),%eax 才是从 top 的地址取出 top 的值传给 eax，如图 19.3 所示。

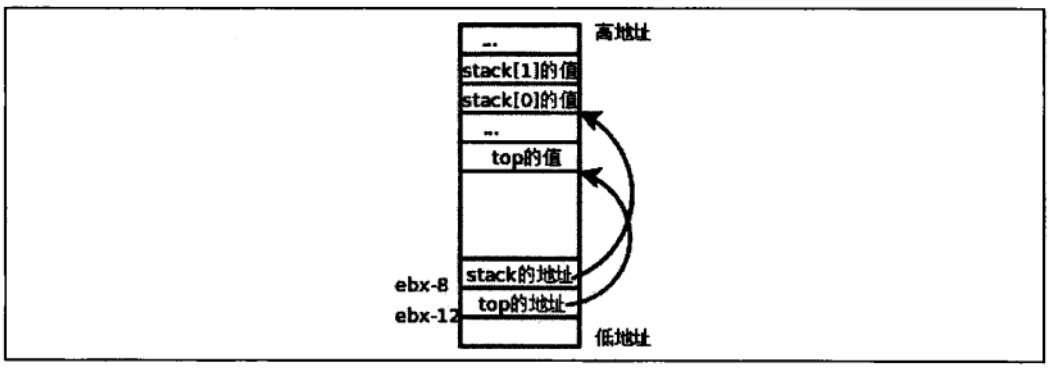


图 19.3　间接寻址

top 和 stack 的绝对地址保存在一个地址表中，而指令通过地址表做间接寻址，可以避免将绝对地址写死在指令中，这也是一种避免硬编码的策略。

现在把 main.c 和共享库编译链接在一起，然后运行：

```
$ gcc main.c -g -L. -lstack -Istack -o main
$ ./main
./main: error while loading shared libraries: libstack.so: cannot
open shared object file: No such file or directory
```

结果出乎意料，编译的时候没问题，由于指定了-L.选项，编译器可以在当前目录下找到 libstack.so，而运行时却说找不到 libstack.so。那么运行时在哪些路径下找共享库呢？我们先用 ldd 命令查看可执行文件依赖于哪些共享库：

```
$ ldd main
    linux-gate.so.1 =>  (0x00670000)
    libstack.so => not found
    libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0x00975000)
    /lib/ld-linux.so.2 (0x00afd000)
```

ldd 模拟运行一遍 main 程序，在运行过程中做动态链接，从而得知这个程序依赖于哪些共享库，这些共享库都在什么路径下。我们在第 18.2 节讲过 gcc 调动 ld 做链接时用-dynamic-linker /lib/ld-linux.so.2 选项指定动态链接器的路径，动态链接器它也像其他共享库一样加载到进程的地址空间中。而另外一个选项-lc 只说明需要链接 libc 库，却没有指出 libc 库的完整路径，-lstack 也是如此，共享库的路径需要在运行时由动态链接器/lib/ld-linux.so.2 去查找。在上面的例子中，动态链

接器找到 libc 的路径是/lib/tls/i686/cmov/ libc.so.6，而 libstack 的路径没有找到，无法完成链接。linux-gate.so.1 这个共享库文件其实并不存在，它是由内核虚拟出来的，所以没有对应的路径。linux-gate.so.1 负责处理一些特殊的系统调用，感兴趣的读者可以参考 http://www.trilithium.com/ johan/2005/08/linux-gate/。那么动态链接器会到哪些目录下搜索共享库呢？从 ld.so(8)可以查到共享库路径的搜索顺序：

1．首先在环境变量 LD_LIBRARY_PATH 保存的路径中查找。

2．然后从缓存文件/etc/ld.so.cache 中查找。这个缓存文件是由 ldconfig 命令读取配置文件/etc/ld.so.conf 生成的，稍后详细解释。

3．如果上述步骤都找不到，则到默认的系统库文件目录中查找，先是/lib 然后是/usr/lib。

先试试第一种方法，在运行程序时设置环境变量 LD_LIBRARY_PATH 把 libstack.so 共享库所在的目录添加到搜索路径：

```
$ LD_LIBRARY_PATH=/home/akaedu/testdir ./main
```

这种方法只适合在开发调试中临时用一下，设置环境变量 LD_LIBRARY_PATH 通常是不推荐的，理由可以参考 Why LD_LIBRARY_PATH is bad.（http://xahlee.org/UnixResource_dir/_/ldpath.html）。环境变量（Environment Variable）是进程运行时保存在内存中的一组字符串，每个字符串都是“key=value”这样的形式，key 是变量名，value 是变量的值。上面这条命令告诉 Shell，在创建进程 main 时传给它一个环境变量 LD_LIBRARY_PATH=/home/akaedu/testdir。也可以用这样两条命令：

```
$ export LD_LIBRARY_PATH=/home/akaedu/testdir
$ ./main
```

第一条命令在当前 Shell 进程中设置一个环境变量 LD_LIBRARY_PATH=/home/akaedu/testdir，一旦在 Shell 进程中设置了环境变量，以后每次执行命令时 Shell 进程都会把自己的环境变量传给新创建的进程，所以第二条命令创建的进程 main 就会获得这个环境变量。注意这两种执行方式的区别，上面的 LD_LIBRARY_PATH=/home/akaedu/testdir ./main 命令表示只有当前创建的 main 进程才获得这个环境变量，Shell 进程本身不保存这个环境变量，以后执行的其他命令也不会获得它。

再试试第二种方法，这是最常用的方法。把 libstack.so 所在目录的绝对路径（比如/home/akaedu/testdir）添加到配置文件/etc/ld.so.conf（该文件中每个路径占一行），然后运行 ldconfig：

```
$ sudo ldconfig -v
...
/home/akaedu/testdir:
        libstack.so -> libstack.so
```

```
/lib:
...
/usr/lib:
...
/lib/tls: (hwcap: 0x8000000000000000)
...
/lib/i486: (hwcap: 0x0002000000000000)
...
/lib/i586: (hwcap: 0x0004000000000000)
...
/lib/i686: (hwcap: 0x0008000000000000)
...
/usr/lib/i686: (hwcap: 0x0008000000000000)
...
/usr/lib/sse2: (hwcap: 0x0000000004000000)
...
/lib/tls/i686: (hwcap: 0x8008000000000000)
...
/lib/i686/cmov: (hwcap: 0x0008000000008000)
...
/usr/lib/i686/cmov: (hwcap: 0x0008000000008000)
...
/usr/lib/i686/sse2: (hwcap: 0x0008000004000000)
...
/lib/tls/i686/cmov: (hwcap: 0x8008000000008000)
...
```

ldconfig 命令除了处理/etc/ld.so.conf 中配置的目录之外，还处理一些默认目录，如/lib、/usr/lib 等，处理的过程主要是建立索引以便快速查找，处理之后生成/etc/ld.so.cache 缓存文件，动态链接器就从这个缓存文件中搜索共享库。hwcap 是 x86 平台 Linux 特有的一种机制，系统检测到当前平台是 i686 而不是 i586 或 i486，所以在运行程序时使用 i686 的库，这样可以更好地发挥平台的性能，所以上面 ldd 命令的输出结果显示动态链接器搜索到的 libc 库是/lib/tls/i686/cmov/libc.so.6，而不是/lib/libc.so.6。更新了缓存文件之后再用 ldd 命令查看，libstack.so 就能找到了：

```
$ ldd main
    linux-gate.so.1 =>  (0x00cc2000)
    libstack.so => /home/akaedu/testdir/libstack.so (0x00fa0000)
    libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0x00aae000)
    /lib/ld-linux.so.2 (0x00f5a000)
```

第三种方法就是把 libstack.so 拷到/usr/lib 或/lib 目录，这样可以确保动态链接器能找到这个共享库。

其实还有第四种方法，在链接生成可执行文件时就把 libstack.so 的路径写到文件中：

```
$ gcc main.c -g -L. -lstack -Istack -o main -Wl,-rpath,/home/akaedu/
testdir
```

注意选项-Wl,-rpath,/home/akaedu/testdir，-Wl 表示 gcc 传给链接器的选项，在这个例子中传给链接器的选项是-rpath /home/akaedu/testdir。可以看到 readelf 输出的.dynamic 段的信息中多了一条 rpath 记录：

```
Dynamic section at offset 0xf10 contains 23 entries:
  Tag        Type                         Name/Value
 0x00000001 (NEEDED)                     Shared library: [libstack.so]
 0x00000001 (NEEDED)                     Shared library: [libc.so.6]
 0x0000000f (RPATH)                      Library rpath: [/home/akaedu/testdir]
...
```

还可以看出，可执行文件运行时需要哪些共享库也都记录在.dynamic 段中。当然rpath 这种办法也是不推荐的，把共享库的搜索路径写到可执行文件中也是一种硬编码的做法。

19.4.2 函数的动态链接过程

我们研究一下在 main.c 中调用共享库的函数 push 是怎样一个过程。用 gcc main.c -g -L. -lstack -Istack -o main 命令行编译，然后反汇编查看可执行文件 main：

```
$ objdump -dS main
...
Disassembly of section .plt:

...
080483fc <push@plt>:
 80483fc:   ff 25 08 a0 04 08       jmp    *0x804a008
 8048402:   68 10 00 00 00          push   $0x10
 8048407:   e9 c0 ff ff ff          jmp    80483cc <_init+0x30>

Disassembly of section .text:

...
080484c4 <main>:
/* main.c */
#include <stdio.h>
#include "stack.h"

int main(void)
{
 80484c4:   55                      push   %ebp
 80484c5:   89 e5                   mov    %esp,%ebp
 80484c7:   83 e4 f0                and    $0xfffffff0,%esp
 80484ca:   83 ec 10                sub    $0x10,%esp
    push('a');
 80484cd:   c7 04 24 61 00 00 00    movl   $0x61,(%esp)
 80484d4:   e8 23 ff ff ff          call   80483fc <push@plt>
...
```

和链接静态库的情况不同，push 函数的指令没有链接到可执行文件中，而且 call 80483fc <push@plt>这条指令调用的也不是 push 函数的地址，而是.plt 段里的地址。PLT 是 Procedure Linkage Table 的缩写，.plt 段里保存的也是指令，和.text 一起合并到 Text Segment。那么地址 0x80483fc 处的三条指令和 push 函数是什么关系？通过这三条指令能最终调用到 push 函数吗？我们用 gdb 跟踪一下：

```
$ gdb main
...
(gdb) start
```

```
Temporary breakpoint 1 at 0x80484cd: file main.c, line 7.
Starting program: /home/akaedu/testdir/main

Temporary breakpoint 1, main () at main.c:7
7       push('a');
(gdb) si
0x080484d4  7       push('a');
(gdb) si
0x080483fc in push@plt ()
(gdb) disassemble
Dump of assembler code for function push@plt:
=> 0x080483fc <+0>: jmp    *0x804a008
   0x08048402 <+6>: push   $0x10
   0x08048407 <+11>:    jmp    0x80483cc
End of assembler dump.
```

进入.plt 段之后要执行的第一条指令是 jmp *0x804a008，这条指令并不是跳转到地址 0x804a008，而是读取地址 0x804a008 处保存的 4 个字节，把它作为跳转的目标地址。我们看看这个目标地址到底是多少：

```
(gdb) x 0x804a008
0x804a008 <_GLOBAL_OFFSET_TABLE_+20>:  0x08048402
```

原来就是下一条指令 push $0x10 的地址。继续跟踪下去：

```
(gdb) si
0x08048402 in push@plt ()
(gdb) si
0x08048407 in push@plt ()
(gdb) si
0x080483cc in ?? ()
(gdb) si
0x080483d2 in ?? ()
(gdb) si
0x001233b0 in ?? () from /lib/ld-linux.so.2
```

最终进入了动态链接器/lib/ld-linux.so.2，在其中完成动态链接的过程并调用共享库的 push 函数，我们不深入这些细节了，直接用 finish 命令返回到 main 函数：

```
(gdb) finish
Run till exit from #0  0x001233b0 in ?? () from /lib/ld-linux.so.2
main () at main.c:8
8       return 0;
```

这时再看看地址 0x804a008 处保存的跳转目标地址：

```
(gdb) x 0x804a008
0x804a008 <_GLOBAL_OFFSET_TABLE_+20>:  0x0012e4bc
(gdb) disassemble 0x0012e4bc
Dump of assembler code for function push:
   0x0012e4bc <+0>:     push   %ebp
   0x0012e4bd <+1>:     mov    %esp,%ebp
   0x0012e4bf <+3>:     push   %ebx
   0x0012e4c0 <+4>:     sub    $0x4,%esp
   0x0012e4c3 <+7>:     call   0x12e4b7 <__i686.get_pc_thunk.bx>
   0x0012e4c8 <+12>:    add    $0x1b2c,%ebx
   0x0012e4ce <+18>:    mov    0x8(%ebp),%eax
```

```
   0x0012e4d1 <+21>:    mov    %al,-0x8(%ebp)
   0x0012e4d4 <+24>:    mov    -0xc(%ebx),%eax
   0x0012e4da <+30>:    mov    (%eax),%eax
   0x0012e4dc <+32>:    lea    0x1(%eax),%edx
   0x0012e4df <+35>:    mov    -0xc(%ebx),%eax
   0x0012e4e5 <+41>:    mov    %edx,(%eax)
   0x0012e4e7 <+43>:    mov    -0xc(%ebx),%eax
   0x0012e4ed <+49>:    mov    (%eax),%eax
   0x0012e4ef <+51>:    mov    -0x8(%ebx),%edx
   0x0012e4f5 <+57>:    movzbl -0x8(%ebp),%ecx
   0x0012e4f9 <+61>:    mov    %cl,(%edx,%eax,1)
   0x0012e4fc <+64>:    add    $0x4,%esp
   0x0012e4ff <+67>:    pop    %ebx
   0x0012e500 <+68>:    pop    %ebp
   0x0012e501 <+69>:    ret
End of assembler dump.
```

不再是 0x08048402，而是改成了 0x0012e4bc，反汇编可以看到这正是 push 函数的首地址。如果下次再执行 call 80483fc <push@plt>指令，进入.plt 段执行第一条指令 jmp *0x804a008 就会立刻跳到 push 函数中，而不必再进入/lib/ld-linux.so.2 做动态链接了。地址 0x804a008 位于 Global Offset Table 中，动态链接器利用 Global Offset Table 的表项保存共享库中符号的绝对地址，链接完成后，通过 Global Offset Table 的表项间接寻址即可访问共享库中的符号。

19.4.3 共享库的命名惯例

你可能已经注意到了，系统的共享库通常带有符号链接，例如：

```
$ ls -l /lib
...
-rwxr-xr-x  1 root root  113964 2010-08-17 17:32 ld-2.11.1.so
lrwxrwxrwx  1 root root      12 2010-08-22 21:08 ld-linux.so.2 ->
ld-2.11.1.so
...
-rwxr-xr-x  1 root root 1335560 2010-08-17 17:32 libc-2.11.1.so
lrwxrwxrwx  1 root root      14 2010-05-01 06:30 libcap.so.2 ->
libcap.so.2.17
-rw-r--r--  1 root root   13852 2010-03-09 05:42 libcap.so.2.17
...
lrwxrwxrwx  1 root root       14 2010-08-22 21:08 libc.so.6 ->
libc-2.11.1.so
...
$ ls -l /usr/lib/libc.so
-rw-r--r--  1 root root     238 2010-08-17 17:12 libc.so
```

按照共享库的命名惯例，每个共享库有三个文件名：real name、soname 和 linker name。真正的库文件（而不是符号链接）的名字是 real name，包含完整的共享库版本号，例如上面的 libcap.so.2.17、libc-2.11.1.so 等。注意动态链接器的名字特殊，不叫 libxxx，而叫 ld-2.11.1.so。

soname 是符号链接的名字，只包含共享库的主版本号，主版本号一致即可保证库函数的接口一致，可执行文件的.dynamic 段只记录共享库的 soname，动态链接器

只要找到 soname 一致的共享库就可以加载它做动态链接。例如上面的 libcap.so.2 其实是指向 libcap.so.2.17 的符号链接，这说明主版本号是 2，次版本号是 17，其实应用程序并不关心这个符号链接所指向的真正的库文件是 libcap.so.2.17 还是 libcap.so.2.16，应用程序只认 libcap.so.2 这个 soname，只要 soname 正确，这个共享库就应该提供了应用程序所需要的接口，就应该可以正确链接和运行，这意味着 libcap.so.2.17 和 libcap.so.2.16 相比可能增加了一些函数接口，或者修正了一些 Bug，但绝不应该修改或删除了一些函数接口。使用共享库可以很方便地升级库文件（改一下符号链接的指向就可以了）而不需要重新编译程序，这是静态库所没有的优点。注意 libc 的版本编号有一点特殊，libc-2.11.1.so 的主版本号是 6 而不是 2 或 2.11，这也是由于历史原因，Linux 曾经用过另外一套 libc 的实现，后来才改用 glibc，但主版本号仍沿用了原来的。另外，动态链接器的 soname 也很特殊，叫 ld-linux.so.2，它指向 ld-2.11.1.so。

linker name 仅在编译链接时使用，gcc 的-L 选项应该指定 linker name 所在的目录。有的 linker name 是库文件的一个符号链接，有的 linker name 是一段链接脚本。例如上面的 libc.so 就是一个 linker name，它是一段链接脚本，其中指定了 soname 的路径和其他需要提供给链接器的信息。

```
$ cat /usr/lib/libc.so
/* GNU ld script
   Use the shared library, but some functions are only in
   the static library, so try that secondarily.  */
OUTPUT_FORMAT(elf32-i386)
GROUP ( /lib/libc.so.6 /usr/lib/libc_nonshared.a AS_NEEDED
( /lib/ld-linux.so.2 ) )
```

以前我们编译 libstack 时没有指定 soname，默认的 soname 就是 libstack.so，现在我们重新编译 libstack 给它指定 soname：

```
$ gcc -shared -Wl,-soname,libstack.so.1 -o libstack.so.1.0 stack.o
push.o pop.o is_empty.o
```

这样编译生成的库文件是 libstack.so.1.0，是 real name，但这个库文件中记录了它的 soname 是 libstack.so.1：

```
$ readelf -a libstack.so.1.0
...
Dynamic section at offset 0xf10 contains 22 entries:
  Tag        Type                         Name/Value
 0x00000001 (NEEDED)                     Shared library: [libc.so.6]
 0x0000000e (SONAME)                     Library soname: [libstack.so.1]
...
```

如果把 libstack.so.1.0 所在的目录加入/etc/ld.so.conf 中，然后运行 ldconfig 命令，ldconfig 会自动创建一个 soname 的符号链接：

```
$ sudo ldconfig
$ ls -l libstack*
lrwxrwxrwx 1 root    root      15 2010-08-21 17:52 libstack.so.1 ->
libstack.so.1.0
-rwxr-xr-x 1 akaedu  akaedu  10142 2010-08-21 17:49 libstack.so.1.0
```

但这样编译链接 main.c 却会报错：

```
$ gcc main.c -L. -lstack -Istack -o main
/usr/bin/ld: cannot find -lstack
collect2: ld returned 1 exit status
```

注意，要做这个实验，你得把先前编译的 libstack 共享库、静态库都删掉，如果先前把共享库拷到/lib 或者/usr/lib 目录下了也要记得删掉，只留下 libstack.so.1.0 和 libstack.so.1，这样你会发现编译器不认这两个名字，因为编译器只认 linker name，先创建一个 linker name 的符号链接再编译就没问题了：

```
$ ln -s libstack.so.1.0 libstack.so
$ gcc main.c -L. -lstack -Istack -o main
```

如果用 readelf 命令查看可执行文件 main 的.dynamic 段，会看到这个程序依赖于 libstack 的 soname，而不是 real name 或 linker name。

19.5 虚拟内存管理

我们知道操作系统利用体系结构提供的 VA 到 PA 的转换机制实现虚拟内存管理机制，在第 16.4 节只是简单一提，现在有了共享库的基础知识，可以再深入讨论一下。首先分析一个例子：

```
$ ps
  PID TTY          TIME CMD
 1549 pts/0    00:00:00 bash
 2564 pts/0    00:00:00 ps
$ cat /proc/1549/maps
00110000-00263000 r-xp 00000000 08:02 152177     /lib/tls/i686/cmov/
libc-2.11.1.so
00263000-00264000 ---p 00153000 08:02 152177     /lib/tls/i686/cmov/
libc-2.11.1.so
00264000-00266000 r--p 00153000 08:02 152177     /lib/tls/i686/cmov/
libc-2.11.1.so
00266000-00267000 rw-p 00155000 08:02 152177     /lib/tls/i686/cmov/
libc-2.11.1.so
...
00343000-0035e000 r-xp 00000000 08:02 140034     /lib/ld-2.11.1.so
0035e000-0035f000 r--p 0001a000 08:02 140034     /lib/ld-2.11.1.so
0035f000-00360000 rw-p 0001b000 08:02 140034     /lib/ld-2.11.1.so
...
00c57000-00c58000 r-xp 00000000 00:00 0          [vdso]
...
08048000-0810b000 r-xp 00000000 08:02 130329     /bin/bash
0810b000-0810c000 r--p 000c2000 08:02 130329     /bin/bash
0810c000-08111000 rw-p 000c3000 08:02 130329     /bin/bash
...
08262000-084ad000 rw-p 00000000 00:00 0          [heap]
b75d8000-b7617000 r--p 00000000 08:02 145689     /usr/lib/locale/
zh_CN.utf8/LC_CTYPE
b7617000-b7769000 r--p 00000000 08:02 163640     /usr/lib/locale/
zh_CN.utf8/LC_COLLATE
...
bfbeb000-bfc00000 rw-p 00000000 00:00 0          [stack]
```

用 ps 命令查看当前终端下的进程，得知 bash 进程的 id 是 1549，然后用 cat /proc/1549/maps 命令查看它的虚拟地址空间[①]。/proc 目录下的文件并不是真正的磁盘文件，而是由内核虚拟出来的，当前运行的每个进程在/proc 下都有一个子目录，目录名就是进程的 id，查看子目录下的文件可以得到该进程的相关信息。比如/proc/进程号/maps 文件是进程地址空间的信息，当用户敲 cat 命令查看 maps 文件时，内核将该进程地址空间的信息按上面的格式打印输出，就好像这些信息保存在 maps 文件中一样如图 19.4 所示。

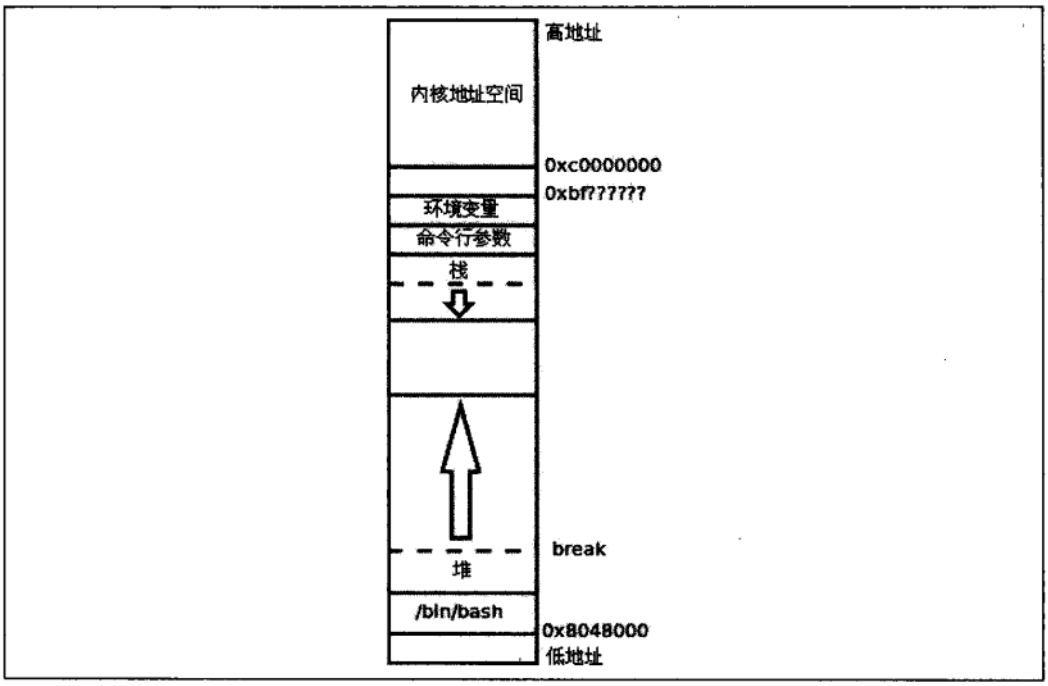


图 19.4　进程地址空间

在第 16.4 节讲过，x86 平台的虚拟地址空间是 0x00000000～0xffffffff，大致上前 3GB（0x00000000～0xbfffffff）是用户空间，后 1GB（0xc0000000～0xffffffff）是内核空间，在这里得到了印证，我们看到的进程地址空间全部位于 0xc0000000 之下。

/bin/bash 程序分三个地址段加载到进程地址空间，每一段有不同的访问权限，0x08048000～0x0810b000 的访问权限是 r-x（只读可执行），0x0810b000～0x0810c000 的访问权限是 r--（只读），0x0810c000～0x08111000 的访问权限是 rw-（可读可写）。动态链接器和各共享库的加载与此类似，也是根据不同的访问权限分几段加载到进程地址空间的。

有三个特殊的地址段不是从磁盘文件加载的，而是直接从内存里分配：

- 标有[vdso]的地址段 0x00c57000～0x00c58000 是 linux-gate.so.1 的映射空间，在第 19.4.1 节讲过这个共享库是由内核虚拟出来的。

① 用 pmap 1549 命令也可以得到类似的输出结果。

- 0x08262000～0x084ad000这一段称为堆（Heap），在第23.1.2节会讲到用malloc函数动态分配内存是在这里分配的。可以看到，堆空间的结束地址下面有很大的地址空洞（0x084ad000～b75d8000），动态分配内存时堆空间可以向高地址增长，并且有很大的增长余地。堆空间的结束地址（0x084ad000）称为Break，堆空间要向高地址增长就要抬高Break，映射新的虚拟内存页面到物理内存，这是通过系统调用brk实现的，malloc函数也是调用brk向内核请求分配内存的。
- 0xbfbeb000～0xbfc00000是栈空间，其中高地址的部分保存着进程的环境变量和命令行参数（命令行参数详见第22.6节），低地址的部分是栈空间，栈空间从高地址向低地址增长，但显然没有堆空间那么大的可供增长的余地，因为实际的应用程序动态分配大量内存的并不少见，但是需要很大栈空间的非常少见。设想一个应用程序有几十层深的函数调用，并且每层调用都有非常多的局部变量，这十分罕见。总之，栈空间是可能用尽的，并且比堆空间更容易用尽，比如在第5.3节讲过无穷递归会用尽栈空间，最终导致段错误。

操作系统的虚拟内存管理机制起到了什么作用呢？可以从以下几个方面来理解。

第一，可以控制物理内存的访问权限。物理内存本身是不限制访问的，任何地址都可以读写，而操作系统要求不同的页面具有不同的访问权限，这是利用CPU模式和MMU的内存保护机制实现的。例如，Text Segment被只读保护起来，防止执行了错误的指令意外改写Text Segment，内核地址空间也被保护起来，防止在用户模式下访问内核数据或执行内核代码。这样，错误的指令或恶意代码的破坏能力受到了限制，顶多使当前进程因段错误而终止，不会影响到整个系统的稳定性。

第二，使每个进程有独立的地址空间。不同进程中相同的VA被MMU映射到不同的PA，因此在某一个进程中访问任何虚拟地址都不可能访问到属于另外一个进程的物理内存页面，并且每个进程都认为自己独占0x00000000～0xbfffffff的整个用户地址空间。独立地址空间的好处是：任何一个进程由于执行了错误指令或恶意代码而导致的非法内存访问都不会意外改写其他进程的数据，不会影响其他进程的运行；链接器和加载器的实现也比较容易，不必考虑各进程的地址范围是否冲突。

继续前面的实验，再打开一个终端窗口，看一下这个新的bash进程的地址空间，可以发现和先前的bash进程地址空间的布局差不多：

```
$ ps
  PID TTY          TIME CMD
 2700 pts/1    00:00:00 bash
 2718 pts/1    00:00:00 ps
$ cat /proc/2700/maps
00110000-00111000 r-xp 00000000 00:00 0          [vdso]
...
0084b000-00866000 r-xp 00000000 08:02 140034     /lib/ld-2.11.1.so
```

```
    00866000-00867000 r--p 0001a000 08:02 140034   /lib/ld-2.11.1.so
    00867000-00868000 rw-p 0001b000 08:02 140034   /lib/ld-2.11.1.so
    00de4000-00f37000  r-xp  00000000  08:02  152177      /lib/tls/i686
/cmov/ libc-2.11.1.so
    00f37000-00f38000 ---p 00153000 08:02 152177      /lib/tls/i686/cmov/
libc-2.11.1.so
    00f38000-00f3a000 r--p 00153000 08:02 152177   /lib/tls/i686/cmov/
libc-2.11.1.so
    00f3a000-00f3b000 rw-p 00155000 08:02 152177   /lib/tls/i686/cmov/
libc-2.11.1.so
    ...
    08048000-0810b000 r-xp 00000000 08:02 130329     /bin/bash
    0810b000-0810c000 r--p 000c2000 08:02 130329     /bin/bash
    0810c000-08111000 rw-p 000c3000 08:02 130329     /bin/bash
    ...
    08f4b000-09193000 rw-p 00000000 00:00 0          [heap]
    b757d000-b75bc000 r--p 00000000 08:02 145689     /usr/lib/locale/
zh_CN.utf8/LC_CTYPE
    b75bc000-b770e000 r--p 00000000 08:02 163640     /usr/lib/locale/
zh_CN.utf8/LC_COLLATE
    ...
    bfeab000-bfec0000 rw-p 00000000 00:00 0          [stack]
```

该进程也占据了 0x00000000～0xbfffffff 的地址空间，并且/bin/bash 的加载进址和先前的进程一模一样，因为这些加载地址是在编译链接时写进/bin/bash 程序的，两个进程都加载这个程序，地址当然应该相同，但共享库的加载地址是在运行时由动态链接器决定的，可以看到两个进程的共享库加载地址大不相同。

现在想一下，这两个进程在同一个系统中同时运行着，它们的可读可写地址段占据相同的虚拟地址范围（0x0810c000～0x08111000），显然 Data Segment 也位于这个地址范围之中，但是两个进程各自干各自的事情，显然 Data Segment 中的数据应该是不同的，相同的虚拟地址范围中怎么会有不同的数据呢？因为它们被映射到不同的物理页面，而每个进程都有自己的一套 VA 到 PA 的映射表，在一个进程中通过 VA 只能访问到属于自己的物理页面，而不会访问到其他进程的物理页面，如图 19.5 所示。

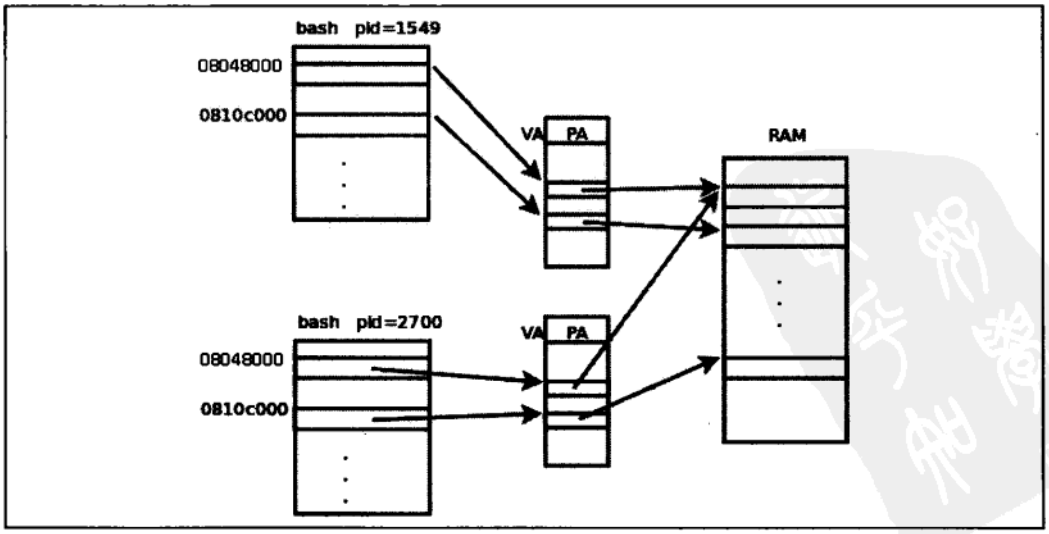


图 19.5　进程地址空间是独立的

从图 19.5 还可以看到，操作系统可以安排两个进程的 Text Segment（位于地址范围 0x08048000～0x0810b000 之间）共享相同的物理页面，因为这两个进程的 Text Segment 都是从/bin/bash 程序加载而来的，而且是只读的不会被改写，所以它们始终都应该是相同的，共享相同的物理页面也不会有问题。另外，虽然两个进程的共享库加载地址并不相同，但共享库中的只读部分也可以像/bin/bash 的 Text Segment 一样加载到物理页面中被多个进程共享，只要每个进程中不同的 VA 映射到相同的 PA 就可以实现共享。使用共享库可以节省物理内存，比如 libc，系统中几乎所有的进程都要加载 libc 到自己的进程地址空间，而 libc 的只读部分在物理内存中只需要保存一份就可以被所有进程共享访问，这就是“共享库”这个名字的由来了。

现在我们也可以理解为什么共享库必须是位置无关代码了。比如 libc，不同的进程虽然共享 libc 所在的物理页面，但这些物理页面被映射到各进程的虚拟地址空间时却位于不同的虚拟地址，所以要求 libc 中的指令**不管加载到什么虚拟地址都能正确执行**。

第三，VA 到 PA 的映射会给分配和释放内存带来方便，物理地址不连续的几块内存可以映射成虚拟地址连续的一块内存。比如要用 malloc 分配一块很大的内存空间，虽然有足够多的空闲物理内存，却没有足够大的**连续**空闲内存，这时就可以分配多个不连续的物理页面而映射到连续的虚拟地址范围，如图 19.6 所示。

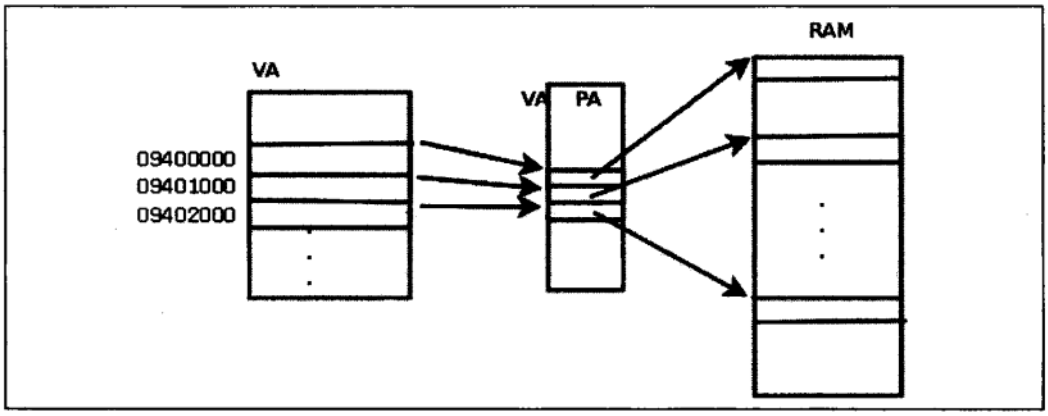


图 19.6 不连续的 PA 可以映射成连续的 VA

第四，一个系统如果同时运行着很多进程，为各进程分配的内存之和可能会大于实际可用的物理内存，虚拟内存管理机制使这种情况下各进程仍然能够正常运行。进程访问的是虚拟内存页面，这些页面的数据可以保存在物理页面中，也可以临时保存在磁盘上而不占用物理页面，可以在磁盘上开一个分区或者建一个文件专门用于临时保存虚拟内存页面的数据，这称为交换设备（Swap Device）。启用了交换设备之后，系统中可分配的内存总量 = 物理内存的大小 + 交换设备的大小。

当物理内存不够用时，操作系统将一些不常用的物理页面中的数据临时保存到交换设备，同时解除 VA 到 PA 的映射，这个物理页面就可以认为是空闲的了，可以重新分配给进程使用，这称为换出（Page out），如图 19.7（a）所示。如果进

程访问到被换出的虚拟内存页面，由于 VA 到 PA 的映射不存在，访问内存的指令会引发一个异常，称为缺页错误（Page Fault），这时进入异常处理程序，操作系统把缺失的页面从交换设备再加载回物理内存，并建立 VA 到 PA 的映射，然后回到用户模式重新执行那条访问内存的指令，这称为换入（Page in），如图 19.7（b）所示。换出和换入操作统称为换页（Paging）。

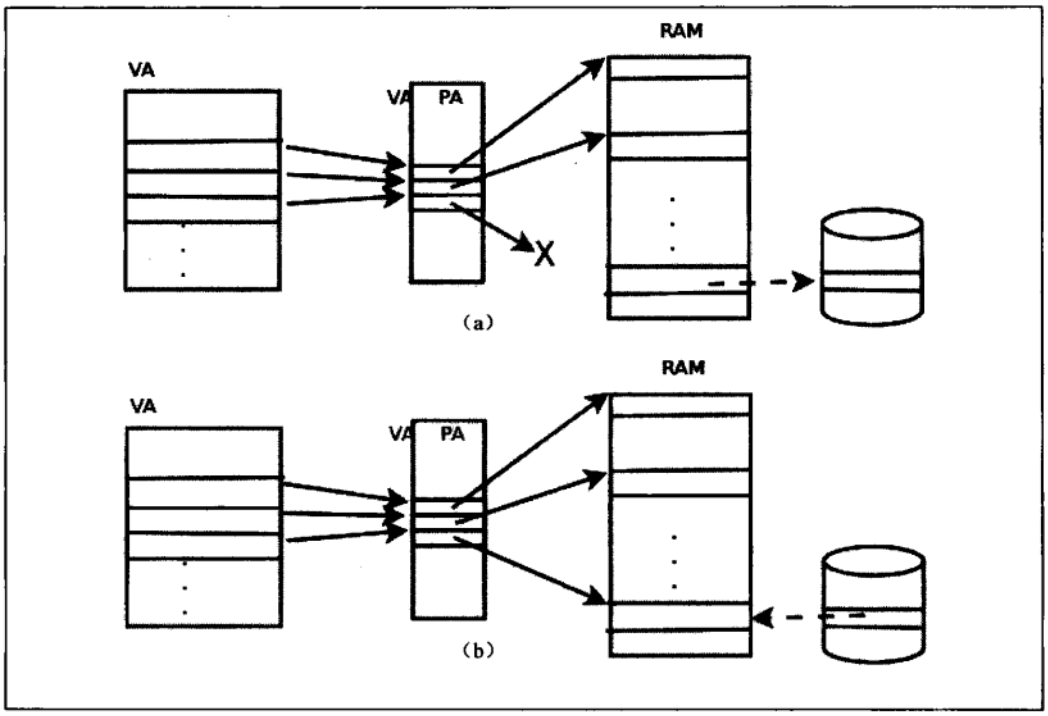


图 19.7　换页

第 20 章
预处理

20.1 预处理的步骤

现在我们全面了解一下 C 编译器做语法解析之前的预处理步骤。

1．把第 2.2 节提到过的三连符替换成相应的单字符。

2．把用\斜线续行的多行代码接成一行。例如：

```
#define STR "hello, "\
                "world"
```

经过这个预处理步骤之后接成一行：

```
#define STR "hello, "       "world"
```

这种续行的写法要求\斜线后面紧跟换行[①]，中间不能有其他字符（有空格或 Tab 也不行）。

3．把每个注释（不管是单行注释还是多行注释）替换成一个空格。

4．经过以上两步之后去掉了一些换行，有的换行在续行过程中去掉了，有的换行在多行注释之中，也随着注释一起去掉了，剩下的代码行称为逻辑代码行。然后预处理器把每个逻辑代码行划分成 Token 和空白字符（在第 2.6 节讲过 C 语言规定的空白字符包括空格、\t、\v、\r、\n、\f），这时的 Token 称为预处理 Token，包括标识符、关键字、整数常量、浮点数常量、字符常量、字符串字面值、运算符和其他标点符号，在划分 Token 时要遵循第 6.3 节讲过的最长匹配原则。继续分析上面的例子，在处理完续行之后两个源代码行被接成一个逻辑代码行，然后这个逻辑代码行被划分成 Token 和空白字符：#，define，空格，STR，空格，"hello,␣"，Tab，Tab，"world"，换行。

5．在 Token 中识别出宏定义和预处理指示，如果遇到宏定义则做宏展开，遇到预处理指示则做相应的预处理动作。如果遇到#include 预处理指示，则把相应的

① 在第 2.1 节提过，Windows 平台的文本文件用\r\n 做换行符，而 Linux 平台的文本文件用\n 做换行符，C 编译器一般都能处理各种不同的换行符，所以本章不考虑这个细节问题。

源文件包含进来，并对该源文件做以上 1～5 步预处理。

我们早在第 8.2 节就引入了预处理指示这个概念，现在给出它的严格定义。一条预处理指示由一个逻辑代码行组成，第一个预处理 Token 是#号，后面跟若干个预处理 Token，在逻辑代码行的开头、结尾以及各预处理 Token 之间可以出现任意多个空格和 Tab，但不允许出现其他空白字符（除预处理指示之外的其他逻辑代码行对空白字符没有此限制）。

6. 找出字符常量或字符串字面值中的转义序列，用相应的字节来替换它，比如把\n 替换成字节 0x0a。

7. 把相邻的字符串字面值连接起来。继续上面的例子，如果代码中有：

```
printf(
        STR);
```

经过第 4 步预处理之后划分成以下 Token：printf，(，换行，Tab，STR，)，;，换行。经过第 5 步宏展开后变成以下 Token：printf，(，换行，Tab，"hello, "，Tab，Tab，"world"，)，;，换行。然后把相邻的字符串连接起来，变成以下 Token：printf，(，换行，Tab，"hello, world"，)，;，换行。

8. 经过以上处理之后，把空白字符丢掉，把 Token 交给 C 编译器做语法解析，这时就不再是预处理 Token，而称为 C Token 了。继续上面的例子，最后交给 C 编译器做语法解析的 Token 是：printf，(，"hello, world"，)，;。注意，把一个预处理指示写成多行要用\斜线续行，因为根据定义，一条预处理指示只能由一个逻辑代码行组成，而把 C 代码写成多行则不必用\斜线续行，因为换行在 C 代码中只不过是一种空白字符，在做语法解析时所有空白字符都已经丢掉了。

20.2　宏定义

较大的项目都会用大量的宏定义来组织代码，你可以看看/usr/include 下面的头文件中用了多少个宏定义。看起来宏展开只是做个替换而已，其实里面有比较复杂的规则，C 语言有很多复杂但不常用的语法规则本书并不深入讲解，但有关宏展开的语法规则很重要也很常用，本节将会做全面详细的解释。

20.2.1　函数式宏定义

以前我们用过的#define N 20 或#define STR "hello, world"这种宏定义可以称为变量式宏定义（Object-like Macro），宏定义名可以像变量一样在代码中使用。另外一种宏定义可以像函数调用一样在代码中使用，称为函数式宏定义（Function-like Macro）。例如编辑一个文件 main.c：

```
#define MAX(a, b) ((a)>(b)?(a):(b))
k = MAX(i&0x0f, j&0x0f)
```

我们想看第二行的表达式展开成什么样，可以用gcc的-E选项或cpp命令，尽管这个C程序不合语法，但没关系，我们只做预处理而不编译，不会检查程序是否符合C语法。

```
$ cpp main.c
# 1 "main.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "main.c"

k = ((i&0x0f)>(j&0x0f)?(i&0x0f):(j&0x0f))
```

就像函数调用一样，把两个实参表达式分别替换到宏定义中形参a和b的位置。注意这种函数式宏定义和真正的函数调用之间的区别：

1. 函数式宏定义的参数没有类型，预处理器只负责做形式上的替换，而不做参数类型检查，所以传参时要格外小心。

2. 调用真正的函数和调用函数式宏定义的代码编译生成的指令不同。如果MAX是个真正的函数，那么它的函数体return a > b ? a : b;要编译生成指令，代码中出现的每次调用也要编译生成传参指令和call指令。而如果MAX是个函数式宏定义，这个宏定义本身倒不必编译生成指令，但是代码中出现的每次调用编译生成的指令都相当于一个函数体，而不是简单的几条传参指令和call指令。所以，使用函数式宏定义编译生成的目标文件会比较大。

3. 定义这种宏要格外小心，如果上面的定义写成#define MAX(a, b) (a>b?a:b)，省略内层括号，则宏展开就成了k = (i&0x0f>j&0x0f?i&0x0f:j&0x0f)，运算的优先级就错了。同样道理，这个宏定义的外层括号也不能省，想一想为什么。

4. 调用函数时先求实参表达式的值再传给形参，如果实参表达式有Side Effect，那么这些Side Effect只发生一次。例如MAX(++a, b)，如果MAX是个真正的函数，a只增加一次。但如果MAX是上面那样的宏定义，则要展开成k = ((++a)>(b)?(++a):(b))，a可能增加一次也可能增加两次。

5. 上面的例子说明：如果形参在宏定义中出现多次，实参表达式可能被重复求值。这种宏定义要小心使用，即使实参表达式没有Side Effect，执行结果也未必理想，因为代码的执行效率可能较低。下面举一个极端的例子，也是个很有意思的例子。

例20.1 函数式宏定义

```
#define MAX(a, b) ((a)>(b)?(a):(b))

int a[] = { 9, 3, 5, 2, 1, 0, 8, 7, 6, 4 };

int max(int n)
{
        return n == 0 ? a[0] : MAX(a[n], max(n-1));
}

int main(void)
```

```
{
        max(9);
        return 0;
}
```

这段代码从一个数组中找出最大的数，如果 MAX 是个真正的函数，这个算法就是从前到后遍历一遍数组，时间复杂度是Θ(n)，而现在 MAX 是这样一个函数式宏定义，思考一下这个算法的时间复杂度是多少？

尽管函数式宏定义和真正的函数相比有很多缺点，但只要小心使用还是会显著提高代码的执行效率，毕竟省去了分配和释放栈帧、传参、传返回值等一系列工作，一般来说简短的、被频繁调用的函数适合用函数式宏定义来代替实现。

函数式宏定义经常写成这样的形式（取自内核代码 include/linux/pm.h）：

```
#define device_init_wakeup(dev,val) \
        do { \
                device_can_wakeup(dev) = !!(val); \
                device_set_wakeup_enable(dev,val); \
        } while(0)
```

为什么要用 do {...} while(0)括起来呢？不括起来会有什么问题呢？假如这样定义和使用函数式宏定义：

```
#define device_init_wakeup(dev,val) \
                device_can_wakeup(dev) = !!(val); \
                device_set_wakeup_enable(dev,val);

if (n > 0)
    device_init_wakeup(d, v);
```

则宏展开之后函数体的第二条语句不在 if 条件中，结果是错的。那么，如果简单地用{...}括起来组成一个语句块不行吗？看下面的代码：

```
#define device_init_wakeup(dev,val) \
                { device_can_wakeup(dev) = !!(val); \
                device_set_wakeup_enable(dev,val); }

if (n > 0)
    device_init_wakeup(d, v);
else
    continue;
```

问题出在 device_init_wakeup(d, v);末尾的;号：如果不允许写这个;号，看起来不像个函数调用；可如果写了这个;号，宏展开之后就有语法错误。展开之后的形式是 if (...) {...}; else ...，其中{...}是一个由语句块组成的语句，而;号是一个空语句，也就是“if (控制表达式) 语句 语句 else 语句”的格式，这不符合“if (控制表达式) 语句 else 语句”的语法。用 do {...} while(0)是一种比较好的解决办法，展开之后的形式是 if (...) do {...} while (0); else ...，其中 do {...} while (0);就是一条语句。

20.2.2 内联函数

C99引入一个新关键字inline，用于定义内联函数（Inline Function）。这种用法在内核代码中很常见，例如include/linux/rwsem.h中：

```
static inline void down_read(struct rw_semaphore *sem)
{
        might_sleep();
        rwsemtrace(sem,"Entering down_read");
        __down_read(sem);
        rwsemtrace(sem,"Leaving down_read");
}
```

inline关键字告诉编译器，这个函数的调用要尽可能快，可以当普通的函数调用实现，也可以用宏展开的办法实现。我们做个实验，把上一节的例子改一下：

例20.2 内联函数

```
inline int MAX(int a, int b)
{
        return a > b ? a : b;
}

int a[] = { 9, 3, 5, 2, 1, 0, 8, 7, 6, 4 };

int max(int n)
{
        return n == 0 ? a[0] : MAX(a[n], max(n-1));
}

int main(void)
{
        max(9);
        return 0;
}
```

按往常的步骤编译然后反汇编：

```
$ gcc main.c -g
$ objdump -dS a.out
...
int max(int n)
{
 80483c5:       55                      push   %ebp
 80483c6:       89 e5                   mov    %esp,%ebp
 80483c8:       83 ec 18                sub    $0x18,%esp
        return n == 0 ? a[0] : MAX(a[n], max(n-1));
 80483cb:       83 7d 08 00             cmpl   $0x0,0x8(%ebp)
 80483cf:       75 07                   jne    80483d8 <max+0x13>
 80483d1:       a1 40 a0 04 08          mov    0x804a040,%eax
 80483d6:       eb 24                   jmp    80483fc <max+0x37>
 80483d8:       8b 45 08                mov    0x8(%ebp),%eax
 80483db:       83 e8 01                sub    $0x1,%eax
 80483de:       89 04 24                mov    %eax,(%esp)
 80483e1:       e8 df ff ff ff          call   80483c5 <max>
```

```
 80483e6:      8b 55 08              mov    0x8(%ebp),%edx
 80483e9:      8b 14 95 40 a0 04 08  mov    0x804a040(,%edx,4),%edx
 80483f0:      89 44 24 04           mov    %eax,0x4(%esp)
 80483f4:      89 14 24              mov    %edx,(%esp)
 80483f7:      e8 b8 ff ff ff        call   80483b4 <MAX>
}
 80483fc:      c9                    leave
 80483fd:      c3                    ret
...
```

可以看到MAX是作为普通函数调用的。如果指定优化选项编译，然后反汇编：

```
$ gcc main.c -g -O
$ objdump -dS a.out
...
int max(int n)
{
 80483c5:      55                    push   %ebp
 80483c6:      89 e5                 mov    %esp,%ebp
 80483c8:      53                    push   %ebx
 80483c9:      83 ec 14              sub    $0x14,%esp
 80483cc:      8b 5d 08              mov    0x8(%ebp),%ebx
      return n == 0 ? a[0] : MAX(a[n], max(n-1));
 80483cf:      85 db                 test   %ebx,%ebx
 80483d1:      75 07                 jne    80483da <max+0x15>
 80483d3:      a1 40 a0 04 08        mov    0x804a040,%eax
 80483d8:      eb 18                 jmp    80483f2 <max+0x2d>
 80483da:      8d 43 ff              lea    -0x1(%ebx),%eax
 80483dd:      89 04 24              mov    %eax,(%esp)
 80483e0:      e8 e0 ff ff ff        call   80483c5 <max>
inline int MAX(int a, int b)
{
      return a > b ? a : b;
 80483e5:      8b 14 9d 40 a0 04 08  mov    0x804a040(,%ebx,4),%edx
 80483ec:      39 d0                 cmp    %edx,%eax
 80483ee:      7d 02                 jge    80483f2 <max+0x2d>
 80483f0:      89 d0                 mov    %edx,%eax
int a[] = { 9, 3, 5, 2, 1, 0, 8, 7, 6, 4 };

int max(int n)
{
      return n == 0 ? a[0] : MAX(a[n], max(n-1));
}
 80483f2:      83 c4 14              add    $0x14,%esp
 80483f5:      5b                    pop    %ebx
 80483f6:      5d                    pop    %ebp
 80483f7:      c3                    ret
...
```

可以看到并没有生成调用MAX函数的call指令，事实上MAX函数并不独立存在，它的指令内联在max函数中，由于源代码和指令的次序无法对应，max和MAX函数的源代码也交错在一起显示。

请读者从开发和调试的角度思考一下，inline函数与函数式宏定义相比有哪些优点？

20.2.3 #、##运算符和可变参数

#和##是两个预处理运算符（注意不是C语言表达式的运算符），在函数式宏定

义中，#号运算符后面应该跟一个形参（#号和形参之间可以有空格或 Tab），用于创建字符串字面值，例如：

```
#define STR(s) # s
STR(hello   world)
```

用 cpp 命令预处理之后是"hello world"，预处理器用"号把实参括起来成为一个字符串字面值，并且实参中的连续多个空白字符被替换成一个空格。再比如：

```
#define STR(s) #s
fputs(STR(strncmp("ab\"c\0d", "abc", '\4"')
    == 0) STR(: @\n), s);
```

预处理之后是 fputs("strncmp(\"ab\\\"c\\0d\", \"abc\", '\\4\"') == 0" ": @\n", s);，注意如果实参中包含字符常量或字符串字面值，则宏展开之后字符串的界定符"要替换成\"，字符常量或字符串字面值中的\和"字符要替换成\\和\"。

在宏定义中可以用##运算符把前后两个预处理 Token 连接成一个预处理 Token。和#号运算符不同，##运算符不仅限于函数式宏定义，变量式宏定义也可以用。例如：

```
#define CONCAT(a, b) a##b
CONCAT(con, cat)
```

预处理之后是 concat。再比如，要定义一个宏展开成两个#号，可以这样定义：

```
#define HASH_HASH # ## #
```

这个宏定义由三个 Token 组成：#、##和#。中间的##是运算符，宏展开时前后两个#号被这个运算符连接在一起。注意中间的两个空格是不可少的，如果写成####，根据最长匹配原则会被划分成##和##两个 Token，而根据定义##运算符用于连接前后两个预处理 Token，不能出现在宏定义的开头或末尾，所以会报错。

我们知道 printf 函数带有可变参数，函数式宏定义也可以带可变参数，同样是在参数列表中用...表示可变参数。例如：

```
#define showlist(...) printf(#__VA_ARGS__)
#define report(test, ...) ((test)?printf(#test):\
        printf(__VA_ARGS__))
showlist(The first, second, and third items.);
report(x>y, "x is %d but y is %d", x, y);
```

预处理之后变成：

```
printf("The first, second, and third items.");
((x>y)?printf("x>y"): printf("x is %d but y is %d", x, y));
```

宏定义中可变参数的部分用__VA_ARGS__表示，在宏展开时和...对应的几个实参可以看成一个实参来替换掉__VA_ARGS__。

调用函数式宏定义允许传空参数，这一点和函数调用不同，我们通过下面几个例

子理解空参数的用法。

```
#define FOO() foo
FOO()
```

预处理之后变成 foo。FOO 在定义时不带参数，在调用时也不允许传参数给它。

```
#define FOO(a) foo##a
FOO(bar)
FOO()
```

预处理之后变成：

```
foobar
foo
```

FOO 在定义时带一个参数，在调用时必须传一个参数给它，如果不传参数则表示传了一个空参数。

```
#define FOO(a, b, c) a##b##c
FOO(1,2,3)
FOO(1,2,)
FOO(1,,3)
FOO(,,3)
```

预处理之后变成：

```
123
12
13
3
```

FOO 在定义时带三个参数，在调用时也必须传三个参数给它，空参数的位置可以空着，但必须给足三个参数，用两个逗号隔开，FOO(1,2)这样的调用是错误的。

```
#define FOO(a, ...) a##__VA_ARGS__
FOO(1)
FOO(1,2,3,)
```

预处理之后变成：

```
1
12,3,
```

FOO(1)这个调用相当于可变参数部分传了一个空参数，FOO(1,2,3,)这个调用相当于可变参数部分传了三个参数，第三个是空参数。

gcc 有一种扩展语法，如果##运算符用在__VA_ARGS__前面，除了起连接 Token 的作用之外还有一种特殊用法，例如内核代码 net/netfilter/nf_conntrack_proto_sctp.c 中的：

```
#define DEBUGP(format, ...) printk(format, ## __VA_ARGS__)
```

内核函数 printk 类似于 printf，也带有格式化字符串和可变参数，由于内核不能调用 libc 的库函数，所以另外实现了这样一个打印函数。这个函数式宏定义可以这样调用：DEBUGP("info no. %d", 1)。也可以这样调用：DEBUGP("info")。后者相当于可变参数部分传了一个空参数，但展开之后并不是 printk("info",)，而是 printk("info")，当__VA_ARGS__是空参数时，##运算符把它前面的逗号“吃”掉了。

20.2.4 #undef 预处理指示

如果在一个编译单元中重复定义一个宏，C 语言规定这些重复的宏定义必须一模一样。例如这样的重复定义是允许的：

```
    #define OBJ_LIKE (1 - 1)
    #define OBJ_LIKE /* comment */ (1/* comment */-/* comment */  1)/*
comment */
```

在定义的前后多些空白或少些空白没有关系（这里的空白包括空格、Tab、注释，因为前一步预处理要把注释替换成空格），在定义中间连续多个空白等价于一个空白，但在定义中间有空白和没有空白被认为是不同的，所以这样的重复定义是不允许的：

```
    #define OBJ_LIKE (1 - 1)
    #define OBJ_LIKE (1-1)
```

如果需要重新定义一个宏，和原来的定义不同，可以先用#undef 取消原来的定义，再重新定义，例如：

```
    #define X 3
    ... /* X is 3 */
    #undef X
    ... /* X has no definition */
    #define X 2
    ... /* now X is 2 */
```

#undef 的语法比较宽松。如果前面没有#define X 3，后面用#undef X 取消 X 的定义也不算错，当然这么写也不起任何作用。如果前面定义了#define X 3，后面用#undef X 取消 X 的定义之后，再用#undef X 取消一遍也不算错。总之，重复定义一个宏必须定义得一模一样才算对，否则算错，而重复取消一个宏的定义不算错。

20.2.5 宏展开的步骤

以上列举的宏展开的例子都是最简单的，有些宏展开的过程需要做多次替换，例如：

```
    #define sh(x) printf("n" #x "=%d, or %d\n",n##x,alt[x])
    #define sub_z 26
    sh(sub_z)
```

sh(sub_z)要用 sh(x)这个宏定义来展开，形参 x 对应的实参是 sub_z，替换过程如下：

1．#x 要替换成"sub_z"。

2．n##x 要替换成 nsub_z。

3．除了带#和##运算符的参数之外，其他参数在替换之前要对实参本身做充分展开，所以应该先把 sub_z 展开成 26 再替换到 alt[x]中 x 的位置。

4．现在展开成了 printf("n" "sub_z" "=%d, or %d\n",nsub_z,alt[26])，所有参数都替换完了，这时编译器会再扫描一遍，再找出可以展开的宏定义来展开，假设 nsub_z 或 alt 是变量式宏定义，这时会进一步展开。

再举一个例子：

```
#define x 3
#define f(a) f(x * (a))
#undef x
#define x 2
#define g f
#define t(a) a

t(t(g)(0) + t)(1);
```

展开的步骤是：

1．先把 g 展开成 f 再替换到#define t(a) a 中，得到 t(f(0) + t)(1);。

2．根据#define f(a) f(x * (a))，得到 t(f(x * (0)) + t)(1);。

3．把 x 替换成 2，得到 t(f(2 * (0)) + t)(1);。注意，一开始定义 x 为 3，但是后来用#undef x 取消了 x 的定义，又重新定义 x 为 2。当预处理器处理到 t(t(g)(0) + t)(1);这一行代码时 x 已经定义成 2 了，所以用 2 来替换。还要注意一点，现在得到的 t(f(2 * (0)) + t)(1);中仍然有 f，但不能再次根据#define f(a) f(x * (a))展开了，f(2 * (0))就是由展开 f(0)得到的，展开的结果中再出现 f(...)就不展开了，这样规定可以避免无穷展开（类似于无穷递归），因此我们可以放心地使用递归定义，例如#define a a[0]，#define a a.member 等。

4．根据#define t(a) a 最终展开成 f(2 * (0)) + t(1);。这时不能再展开 t(1)了，因为这里的 t 就是由展开 t(f(2 * (0)) + t)得到的，展开的结果中再出现 t(...)就不展开了。

20.3 条件预处理指示

我们在第 19.2.2 节中见过 Header Guard 的写法：

```
#ifndef HEADER_FILENAME
#define HEADER_FILENAME
/* body of header */
#endif
```

除了这种用法之外，条件预处理指示也常用于源代码的配置管理，例如：

```
#if MACHINE == 68000
   int x;
#elif MACHINE == 8086
   long x;
#else    /* all others */
   #error UNKNOWN TARGET MACHINE
#endif
```

假设这段程序是为多种平台编写的，在68000平台上需要定义x为int型，在8086平台上需要定义x为long型，对其他平台暂不提供支持，就可以用条件预处理指示来写。如果在预处理这段代码之前，MACHINE被定义为68000，则包含int x;这行代码；否则如果MACHINE被定义为8086，则包含long x;这行代码；其他情况下（MACHINE没有定义，或者定义为其他值）包含#error UNKNOWN TARGET MACHINE这行代码，编译器遇到#error预处理指示就报错退出，错误信息就是UNKNOWN TARGET MACHINE。

如果要为8086平台编译这段代码，有几种可选的办法：

1．手动编辑代码，在前面添一行#define MACHINE 8086。这样做的缺点是难以管理，如果这个项目中有很多源文件都需要定义MACHINE，每次要为8086平台编译就得把这些定义全部改成8086，每次要为68000平台编译就得把这些定义全部改成68000。

2．在所有需要配置的源文件开头包含一个头文件，在头文件中定义#define MACHINE 8086，这样只需要改一个头文件就可以影响所有包含它的源文件。通常这个头文件由配置工具生成，比如在Linux内核源代码的根目录下运行make menuconfig命令可以出来一个配置菜单（make命令在下一章详细介绍），在其中配置的选项会自动转换成内核源代码目录下include/linux/autoconf.h文件中的宏定义。

举一个具体的例子，在内核配置菜单中用回车键和方向键进入Device Drivers ---> Network device support，然后用空格键选中Network device support（菜单项左边的[ ]括号内会出现一个*号），然后保存退出，在内核源代码的根目录下会生成一个名为.config的隐藏文件，其内容类似于：

```
...
#
# Network device support
#
CONFIG_NETDEVICES=y
# CONFIG_DUMMY is not set
# CONFIG_BONDING is not set
# CONFIG_EQUALIZER is not set
# CONFIG_TUN is not set
...
```

然后在内核源代码的根目录下运行make命令编译内核，这时会根据.config文件生成头文件include/linux/autoconf.h，其内容类似于：

```
...
/*
 * Network device support
 */
#define CONFIG_NETDEVICES 1
#undef CONFIG_DUMMY
#undef CONFIG_BONDING
#undef CONFIG_EQUALIZER
#undef CONFIG_TUN
...
```

在内核配置菜单里选中的项会变成#define 代码，没选中的项会变成#undef 代码。#undef 可以确保取消某个宏的定义，比如#undef CONFIG_DUMMY，如果该编译单元的前面定义过 CONFIG_DUMMY 就取消它的定义，如果前面没定义过 CONFIG_DUMMY 就什么都不做。

include/linux/autoconf.h 被另一个头文件 include/linux/config.h 所包含，通常内核代码包含后一个头文件，例如 net/core/sock.c：

```
...
#include <linux/config.h>
...
int sock_setsockopt(struct socket *sock, int level, int optname,
                    char __user *optval, int optlen)
{
...
#ifdef CONFIG_NETDEVICES
                case SO_BINDTODEVICE:
                {
                ...
                }
#endif
...
```

再比如 drivers/isdn/i4l/isdn_common.c：

```
...
#include <linux/config.h>
...
static int
isdn_ioctl(struct inode *inode, struct file *file, uint cmd, ulong arg)
{
...
#ifdef CONFIG_NETDEVICES
                        case IIOCNETGPN:
                                /* Get peer phone number of a connected
                                 * isdn network interface */
                                if (arg) {
                                        if (copy_from_user(&phone, argp,
                                        sizeof(phone)))
                                                return -EFAULT;
                                        return isdn_net_getpeer(&phone,
                                        argp);
                                } else
                                        return -EINVAL;
#endif
```

```
...
#ifdef CONFIG_NETDEVICES
                    case IIOCNETAIF:
...
#endif                          /* CONFIG_NETDEVICES */
...
```

在内核配置菜单中所做的配置通过条件预处理影响到源代码，从而决定了哪些代码编译到内核中，哪些代码不编译到内核中。#ifdef 或#if 可以嵌套使用，但预处理指示通常都顶头写不缩进，为了区分嵌套的层次，可以像上面代码中最后一行那样，在#endif 处用注释写清楚它结束的是哪个#if 或#ifdef。

3．要定义一个宏不一定非得在代码中用#define 定义，早在第 11.6 节我们就见过用 gcc 的-D 选项定义一个宏 NDEBUG。对于上面的例子，我们需要给 MACHINE 定义一个值，可以写成这样的命令：gcc -c -DMACHINE=8086 main.c。这种方法需要给每个编译命令都加上适当的选项，和第 2 种方法相比似乎也很麻烦，第 2 种方法在头文件中只写一次宏定义就可以在很多源文件中生效，第 3 种方法能不能做到“只写一次到处生效”呢？等下一章学习了 Makefile 就有办法了。

最后通过一个例子看看预处理器如何求值#if 后面的表达式，这个表达式必须是常量表达式。

```
#define VERSION  2
#if defined x || y || VERSION < 3
```

1．首先看预处理运算符 defined，defined 运算符一般用作表达式中的一部分，如果单独使用，#if defined x 相当于#ifdef x，而#if !defined x 相当于#ifndef x。在这个例子中，如果 x 这个宏有定义，则把 defined x 替换成 1，否则替换成 0，因此变成#if 0 || y || VERSION < 3。

2．把有定义的宏展开，变成#if 0 || y || 2 < 3。

3．把没有定义的宏替换成 0，变成#if 0 || 0 || 2 < 3。注意，即使前面定义了一个变量名是 y，在这一步也还是替换成 0，#if 表达式在预处理时求值，预处理器不知道变量名，甚至也不知道枚举常量，#if 表达式中包含的标识符只能是宏定义。

4．把得到的表达式 0 || 0 || 2 < 3 像 C 表达式一样求值，求值的结果是#if 1，因此条件成立。

我们在开发调试中经常需要临时注释掉多行代码，可以用/* ... */注释：

```
/*
代码行
代码行
...
*/
```

这样做有一个问题：代码行中可能已经包含/* ... */注释了，而我们讲过/* ... */注释不能嵌套使用。更好的办法是用#if 0 注释掉多行代码：

```
#if 0
代码行
代码行
...
#endif
```

20.4　其他预处理特性

#pragma 预处理指示供编译器实现一些扩展特性，C 标准没有规定#pragma 后面应该写什么以及起什么作用，由编译器自己规定。有的编译器用#pragma 定义一些特殊功能寄存器名，有的编译器用#pragma 定位链接地址，本书不做深入讨论。如果编译器在代码中碰到不认识的#pragma 指示则忽略它，例如 gcc 的#pragma 指示都是#pragma GCC ...这种形式，别的编译器看到这样的指示直接忽略。

C 标准规定了几个特殊的宏，不需要定义即可使用，最常用的是__FILE__和__LINE__，__FILE__展开成当前源文件的文件名，是一个字符串，__LINE__展开成当前代码行的行号，是一个整数。这两个宏在源代码中不同的地方使用会自动展开成不同的值，显然不是用#define 能定义得出来的，它们是编译器内建的特殊宏定义。在打印调试信息时打印这两个宏可以给开发者非常有用的提示，例如在第 11.6 节我们看到 assert 打印的错误信息就有__FILE__和__LINE__的值。现在我们自己实现 assert，以便理解它的工作原理。这个实现出自参考文献[5]第 1 章：

例 20.3　assert.h 的一种实现

```
/* assert.h standard header */
#undef assert   /* remove existing definition */

#ifdef NDEBUG
        #define assert(test)    ((void)0)
#else       /* NDEBUG not defined */
        void _Assert(char *);
        /* macros */
        #define _STR(x) _VAL(x)
        #define _VAL(x) #x
        #define assert(test)    ((test) ? (void)0 \
                : _Assert(__FILE__ ":" _STR(__LINE__) " " #test))
#endif
```

通过这个例子可以全面复习本章所讲的知识。C 标准规定 assert 应该实现成函数式宏定义而不是一个真正的函数，并且 assert(test)这个表达式应该是 void 类型的。首先用#undef assert 确保取消前面对 assert 的定义，然后分两种情况：如果定义了 NDEBUG，那么 assert(test)直接定义成一个 void 类型的值，什么也不做；如果没有定义 NDEBUG，则要判断测试条件 test 是否成立，如果条件成立就什么也不做，如果条件不成立就调用_Assert 函数打印调试信息。

假设在 main.c 文件的第 33 行调用 assert(is_sorted())，那么__FILE__是字符串"main.c"，__LINE__是整数 33，#test 是字符串"is_sorted()"。注意_STR(__LINE__)的展开过程：首先展开成_VAL(33)，然后进一步展开成字符串"33"。这样，最后

_Assert 调用的形式是_Assert("main.c" ":" "33" " " "is_sorted()")，传给_Assert 函数的字符串是"main.c:33 is_sorted()"。请结合前面讲过的宏展开步骤思考一下，为什么不直接定义#define _STR(x) #x 呢？

_Assert 函数是我们自己定义的，在另一个源文件中：

```
/* xassert.c _Assert function */
#include <stdio.h>
#include <stdlib.h>

void _Assert(char *mesg)
{               /* print assertion message and abort */
        fputs(mesg, stderr);
        fputs(" -- assertion failed\n", stderr);
        abort();
}
```

我们在这个实现中定义了一些以_线开头的标识符，例如_STR、_VAL、_Assert，这些标识符是我们的实现内部使用的，并不打算提供给用户程序使用。在第 2.3 节讲过，C 标准库定义了很多以下划线开头的标识符留作内部使用，在/usr/include 下的头文件中你可以找到很多以_线开头的标识符。

另外一个问题是：为什么我们不直接在 assert 宏定义中调用 fputs 和 abort 函数呢？因为 C 标准规定 C 标准库的头文件是相互独立的，用户程序只要包含 assert.h 就应该能使用 assert 宏定义，assert.h 不应该依赖于别的头文件。如果在 assert.h 中直接调用 fputs 和 abort，那么用户程序只包含 assert.h 是不够的，还必须同时包含 stdio.h 和 stdlib.h，头文件之间就不是独立的了。fputs 函数向标准错误输出打印错误信息（详见第 24.2.7 节），abort 函数异常终止当前进程（本书不做详细介绍，请参考 Man Page）。

现在测试一下我们实现的 assert，把 assert.h、xassert.c 和测试代码 main.c 放在同一个目录下。

```
/* main.c */
#include "assert.h"

int main(void)
{
        assert(2>3);
        return 0;
}
```

注意#include "assert.h"要用"引号而不要用角括号，以保证包含的是我们自己写的 assert.h 而非 C 标准库的头文件。然后编译运行：

```
$ gcc main.c xassert.c
$ ./a.out
main.c:6 2>3 -- assertion failed
Aborted
```

在打印调试信息时除了文件名和行号之外还可以打印出当前函数名，C99 引入一

个特殊标识符__func__支持这一功能。这个标识符应该是一个变量名而不是宏定义，不在预处理阶段求值，但它的作用和__FILE__、__LINE__类似，所以放在一起讲。例如：

例 20.4　特殊标识符__func__

```
#include <stdio.h>

void myfunc(void)
{
        printf("%s\n", __func__);
}

int main(void)
{
        myfunc();
        printf("%s\n", __func__);
        return 0;
}
```

运行结果如下：

```
$ gcc main.c
$ ./a.out
myfunc
main
```

第 21 章

Makefile 基础

21.1 基本规则

除了 Hello World 这种极简单的程序之外，一般的程序都是由多个源文件编译链接而成的，这些源文件的编译过程通常用 Makefile 来管理。Makefile 起什么作用呢？我们先看一个例子，这个例子由例 12.3 改写而成：

```
/* main.c */
#include <stdio.h>
#include "main.h"
#include "stack.h"
#include "maze.h"

struct point predecessor[MAX_ROW][MAX_COL] = {
        {{-1,-1}, {-1,-1}, {-1,-1}, {-1,-1}, {-1,-1}},
        {{-1,-1}, {-1,-1}, {-1,-1}, {-1,-1}, {-1,-1}},
        {{-1,-1}, {-1,-1}, {-1,-1}, {-1,-1}, {-1,-1}},
        {{-1,-1}, {-1,-1}, {-1,-1}, {-1,-1}, {-1,-1}},
        {{-1,-1}, {-1,-1}, {-1,-1}, {-1,-1}, {-1,-1}},
};

void visit(int row, int col, struct point pre)
{
        struct point visit_point = { row, col };
        maze[row][col] = 2;
        predecessor[row][col] = pre;
        push(visit_point);
}

int main(void)
{
        struct point p = { 0, 0 };

        maze[p.row][p.col] = 2;
        push(p);

        while (!is_empty()) {
                p = pop();
                if (p.row == MAX_ROW - 1  /* goal */
                && p.col == MAX_COL - 1)
                        break;
                if (p.col+1 < MAX_COL    /* right */
                && maze[p.row][p.col+1] == 0)
                        visit(p.row, p.col+1, p);
```

```
                if (p.row+1 < MAX_ROW    /* down */
                && maze[p.row+1][p.col] == 0)
                        visit(p.row+1, p.col, p);
                if (p.col-1 >= 0         /* left */
                    && maze[p.row][p.col-1] == 0)
                            visit(p.row, p.col-1, p);
                if (p.row-1 >= 0         /* up */
                && maze[p.row-1][p.col] == 0)
                        visit(p.row-1, p.col, p);
                print_maze();
        }
        if (p.row == MAX_ROW - 1 && p.col == MAX_COL - 1) {
                printf("(%d, %d)\n", p.row, p.col);
                while (predecessor[p.row][p.col].row != -1) {
                        p = predecessor[p.row][p.col];
                        printf("(%d, %d)\n", p.row, p.col);
                }
        } else
                printf("No path!\n");

        return 0;
}
```

我们把堆栈和迷宫的代码分别转移到stack.c和maze.c中，main.c包含它们提供的头文件stack.h和maze.h。

```
/* main.h */
#ifndef MAIN_H
#define MAIN_H

typedef struct point { int row, col; } item_t;

#define MAX_ROW 5
#define MAX_COL 5

#endif
```

在main.h中定义了一个类型和两个常量，main.c、stack.c和maze.c都要用到这些定义，都要包含这个头文件。

```
/* stack.c */
#include "stack.h"

static item_t stack[512];
static int top = 0;

void push(item_t p)
{
        stack[top++] = p;
}

item_t pop(void)
{
        return stack[--top];
}

int is_empty(void)
{
```

```
        return top == 0;
}
/* stack.h */
#ifndef STACK_H
#define STACK_H

#include "main.h" /* provides definition for item_t */

extern void push(item_t);
extern item_t pop(void);
extern int is_empty(void);

#endif
```

例 12.3 代码中的堆栈规定死了只能放 char 型数据，现在我们做进一步抽象，堆栈中放 item_t 类型的数据，item_t 可以定义为任意类型，只要它能够通过函数的参数和返回值传递并且支持赋值操作就行。这也是一种避免硬编码的策略，stack.c 中多次使用 item_t 类型，要改变它的定义只需改变 main.h 中的一行代码。

```
/* maze.c */
#include <stdio.h>
#include "maze.h"

int maze[MAX_ROW][MAX_COL] = {
        0, 1, 0, 0, 0,
        0, 1, 0, 1, 0,
        0, 0, 0, 0, 0,
        0, 1, 1, 1, 0,
        0, 0, 0, 1, 0,
};

void print_maze(void)
{
        int i, j;
        for (i = 0; i < MAX_ROW; i++) {
                for (j = 0; j < MAX_COL; j++)
                        printf("%d ", maze[i][j]);
                putchar('\n');
        }
        printf("*********\n");
}
/* maze.h */
#ifndef MAZE_H
#define MAZE_H

#include "main.h" /* provides defintion for MAX_ROW and MAX_COL */

extern int maze[MAX_ROW][MAX_COL];
void print_maze(void);

#endif
```

maze.c 中定义了一个 maze 数组和一个 print_maze 函数，需要在头文件 maze.h 中声明，以便提供给 main.c 使用。注意 print_maze 函数的声明可以不加 extern，而 maze 数组的声明必须加 extern。

这些源文件可以这样编译链接在一起：

```
$ gcc main.c stack.c maze.c -o main
```

但这不是个好办法，如果编译之后又对 maze.c 做了修改，又要把所有源文件编译一遍，即使 main.c、stack.c 以及那几个头文件都没有修改也要跟着重新编译。一个大型的软件项目往往由上千个源文件组成，全部编译一遍需要几个小时，只改一个源文件就要求全部重新编译肯定是不合理的。

这样编译也许更好一些：

```
$ gcc -c main.c
$ gcc -c stack.c
$ gcc -c maze.c
$ gcc main.o stack.o maze.o -o main
```

如果编译之后又对 maze.c 做了修改，只需重新编译 maze.c，然后和原来编译好的 main.o、stack.o 做链接：

```
$ gcc -c maze.c
$ gcc main.o stack.o maze.o -o main
```

这样又有一个问题，每次编译敲的命令都不一样，很容易出错，比如我修改了三个源文件，可能有一个忘了重新编译，结果编译完了修改没生效，运行时出了 Bug 还满世界找原因呢。更复杂的问题是，假如我改了 main.h 怎么办？所有包含 main.h 的源文件都需要重新编译，我得挨个找哪些源文件包含了 main.h，有的还很不明显，例如 stack.c 包含了 stack.h，而后者包含了 main.h。可见手动处理这些问题非常容易出错，那有没有自动的解决办法呢？有，就是写一个 Makefile 文件和源代码放在同一个目录下：

```
main: main.o stack.o maze.o
        gcc main.o stack.o maze.o -o main

main.o: main.c main.h stack.h maze.h
        gcc -c main.c

stack.o: stack.c stack.h main.h
        gcc -c stack.c

maze.o: maze.c maze.h main.h
        gcc -c maze.c
```

然后在这个目录下敲 make 命令编译：

```
$ make
gcc -c main.c
gcc -c stack.c
gcc -c maze.c
gcc main.o stack.o maze.o -o main
```

make 程序会自动读取当前目录下的 Makefile 文件[①]，完成相应的编译步骤。Makefile 由一组规则（Rule）组成，每条规则的格式是：

```
target ... : prerequisites ...
        command1
        command2
        ...
```

例如：

```
main: main.o stack.o maze.o
        gcc main.o stack.o maze.o -o main
```

main 是这条规则的目标（Target），main.o、stack.o 和 maze.o 是这条规则的条件（Prerequisite）。目标和条件之间的关系是：**欲更新目标，必须先更新它的所有条件；所有条件中只要有一个条件被更新了，目标也必须随之被更新。**所谓“更新”就是执行一遍规则中的命令列表，命令列表中的每条命令必须以一个 Tab 开头，注意不能用空格代替这个 Tab，Makefile 的格式不像 C 语言的缩进那么随意。对于 Makefile 中的每个以 Tab 开头的命令，make 会启动一个 Shell 进程去执行它。

对于上面这个例子，make 执行如下步骤：

1．尝试更新 Makefile 中第一条规则的目标 main，第一条规则的目标称为缺省目标，只要缺省目标更新了就算完成任务了，其他工作都是为这个目标而做的。由于我们是第一次编译，main 文件还没生成，显然需要更新，但规则说必须先更新了 main.o、stack.o 和 maze.o 这三个条件，然后才能更新 main。

2．所以 make 会进一步查找以这三个条件为目标的规则，这些目标文件也没有生成，也需要更新，所以执行相应的命令（gcc -c main.c、gcc -c stack.c 和 gcc -c maze.c）更新它们。

3．最后执行 gcc main.o stack.o maze.o -o main 更新 main。

如果没有做任何改动，再次运行 make：

```
$ make
make: `main' is up to date.
```

make 会提示缺省目标已经是最新的了，不需要执行任何命令更新它。再做个实验，如果修改了 maze.h（比如加个无关痛痒的空格）再运行 make：

① 只要符合本章所描述的语法的文件我们都叫它 Makefile，而它的文件名则不一定是 Makefile。事实上 make 程序依次查找文件名 GNUmakefile、makefile 和 Makefile，找到第一个存在的文件并执行它，一般建议使用 Makefile 做文件名。各种平台的 make 程序所支持的 Makefile 语法会有些差异，Linux 系统的 make 程序一般是 GNU make，如果你写的 Makefile 包含 GNU make 的特殊语法，可以起名为 GNUmakefile，其他系统的 make 程序如果不是 GNU make 则不会查找 GNUmakefile 这个文件名。

```
$ make
gcc -c main.c
gcc -c maze.c
gcc main.o stack.o maze.o -o main
```

make 会自动选择那些受影响的源文件重新编译，不受影响的源文件则不重新编译，这是怎么做到的呢？在这种情况下 make 的处理步骤是：

1. 检查目标 main 是否需要更新，由于它依赖于三个条件，因此要先检查 main.o、stack.o 和 maze.o 这三个条件是否需要更新。

2. make 进一步查找以这三个条件为目标的规则，然后发现 main.o 和 maze.o 需要更新，因为它们都有一个条件是 maze.h，而这个文件的修改时间比 main.o 和 maze.o 晚，所以执行相应的命令更新 main.o 和 maze.o。

3. 既然 main 的三个条件中有两个被更新过了，那么 main 也需要更新，所以执行命令 gcc main.o stack.o maze.o -o main 更新 main。

假设 Makefile 中有一条规则 A，我们总结一下 make 执行规则 A 的步骤：

1. 首先检查规则 A 的每个条件 P。

- 如果存在以 P 为目标的规则 B，则执行规则 B。在执行规则 A 的过程中要执行规则 B，这是个递归的执行过程。
- 如果找不到以 P 为目标的规则，并且文件 P 已存在，表示 P 不需要更新。
- 如果找不到以 P 为目标的规则，并且文件 P 不存在，则报错退出。

2. 在检查完规则 A 的所有条件后，检查它的目标 T，如果属于以下情况之一，表示 T 需要更新，就执行它的命令列表，执行完命令之后无论是否生成文件 T，都认为 T 被更新过。

- 文件 T 不存在。
- 文件 T 存在，但是某个条件 P 是一个文件，该文件的修改时间比 T 晚。
- 某个条件 P 被更新过（并不一定生成文件 P）。

以上步骤描述得比较抽象，请读者拿前面的例子套用这些步骤来理解 Makefile 的执行过程。通常 Makefile 都会有一个 clean 规则，用于清除编译过程中产生的二进制文件，保留源文件：

```
clean:
        @echo "cleanning project"
        -rm main *.o
        @echo "clean completed"
```

把这条规则添加到我们的 Makefile 末尾，然后执行这条规则：

```
$ make clean
cleanning project
rm main *.o
clean completed
```

在 make 的命令行中可以指定一个或多个目标，比如指定了目标 clean，则执行 Makefile 中更新目标 clean 的规则，如果在 make 的命令行中不指定任何目标，则更新 Makefile 中第一条规则的目标（缺省目标）。

和前面介绍的规则不同，clean 目标不依赖于任何条件，并且执行它的命令列表不会生成 clean 这个文件，刚才说过，只要执行了命令列表就算更新了目标，即使没有生成以目标为文件名的文件也算。在这个例子中还演示了命令前面加@和-字符的效果：如果 make 执行的命令前面加了@字符（At Sign），则不显示命令本身而只显示它的输出结果；通常 make 执行的命令如果出错（该命令的退出状态非 0）就立刻终止，不再执行后续命令，但如果命令前面加了-字符（Hyphen），即使这条命令出错，make 也会继续执行后续命令。通常 rm 命令和 mkdir 命令前面要加上-字符，因为 rm 要删除的文件可能不存在，mkdir 要创建的目录可能已存在，这两个命令都有可能出错，但这种错误其实不算什么错，应该继续执行下去。例如上面已经执行过一遍 make clean，再执行一遍就没有文件可删了，这时 rm 命令会报错，但 make 忽略这一错误，继续执行后面的 echo 命令：

```
$ make clean
cleanning project
rm main *.o
rm: cannot remove `main': No such file or directory
rm: cannot remove `*.o': No such file or directory
make: [clean] Error 1 (ignored)
clean completed
```

读者可以把命令前面的@和-去掉再试试，对比一下结果有何不同。这里还有一个问题，如果当前目录下存在一个文件叫 clean 会怎么样呢？

```
$ touch clean
$ make clean
make: `clean' is up to date.
```

如果存在 clean 这个文件，clean 目标又不依赖于任何条件，make 就认为它不需要更新了。而我们希望把 clean 当做一个特殊的名字使用，不管 clean 文件存在不存在都要更新 clean 目标，可以添加一条特殊规则，把 clean 声明成一个伪目标：

```
.PHONY: clean
```

这条规则没有命令列表。类似.PHONY 这种 make 内建的特殊目标还有很多，各有不同的用途，详见参考文献[27]的 4.8 节。在 C 语言中要求变量和函数先声明后使用，而 Makefile 不太一样，这条规则写在 clean:规则的后面也行，也能起到声明 clean 是伪目标的作用：

```
clean:
        @echo "cleanning project"
        -rm main *.o
        @echo "clean completed"

.PHONY: clean
```

当然写在前面也行。gcc 处理一个 C 程序分为预处理和编译两个阶段，类似地，make 处理 Makefile 的过程也分为两个阶段：

1．从前到后读取所有规则，建立起完整的依赖关系图，如图 21.1 所示。

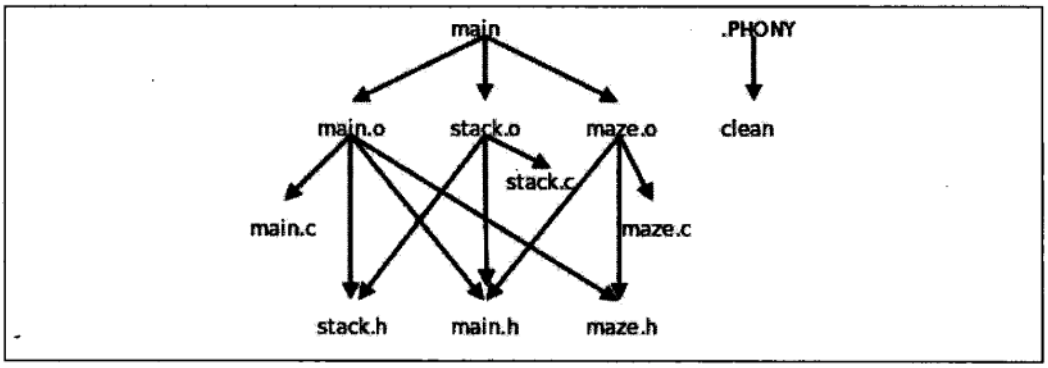


图 21.1　Makefile 的依赖关系图

2.从缺省目标或者命令行指定的目标开始，根据依赖关系图选择适当的规则执行，执行 Makefile 中的规则和执行 C 代码不一样，并不是从前到后按顺序执行，也不是所有规则都要执行一遍，例如在 make 缺省目标时不会更新 clean 目标，因为从图 21.1 可以看出，clean 目标和缺省目标 main 没有任何依赖关系。

clean 目标是一个约定俗成的名字，在所有软件项目的 Makefile 中都表示清除编译生成的文件，类似这样的约定俗成的目标名字有：

- all，执行主要的编译工作，通常用作缺省目标。
- install，执行编译后的安装工作，把可执行文件、配置文件、文档等分别拷贝到不同的安装目录。
- clean，删除编译生成的二进制文件。
- distclean，不仅删除编译生成的二进制文件，也删除其他的生成文件，比如内核源代码 make menuconfig 配置之后会生成.config 文件，一些文档源文件（比如本书的 Docbook 源文件）经过 make 之后会转换生成 HTML 或 PDF 文件，执行 make distclean 应该清除所有的生成文件，只留下源文件。

21.2　隐含规则和模式规则

上一节的 Makefile 写得中规中矩，比较繁琐，主要是为了讲清楚基本概念，其实 Makefile 有很多灵活的写法，可以写得更简洁，同时减少人为出错的可能。本节我们来看看这样一个 Makefile 还有哪些改进的余地。

一个目标依赖的所有条件不一定非得写在一条规则中，也可以拆开写，例如：

```
main.o: main.h stack.h maze.h

main.o: main.c
        gcc -c main.c
```

相当于：

```
main.o: main.c main.h stack.h maze.h
        gcc -c main.c
```

如果一个目标拆开写多条规则，其中只有一条规则允许有命令列表，其他规则应该没有命令列表，否则 make 会报警告并且采用最后一条规则的命令列表。

这样我们的例子可以改写成：

```
main: main.o stack.o maze.o
        gcc main.o stack.o maze.o -o main

main.o: main.h stack.h maze.h
stack.o: stack.h main.h
maze.o: maze.h main.h

main.o: main.c
        gcc -c main.c

stack.o: stack.c
        gcc -c stack.c

maze.o: maze.c
        gcc -c maze.c

clean:
        -rm main *.o

.PHONY: clean
```

这不是比原来更繁琐了吗？现在可以把提出来的三条规则删去，写成：

```
main: main.o stack.o maze.o
        gcc main.o stack.o maze.o -o main

main.o: main.h stack.h maze.h
stack.o: stack.h main.h
maze.o: maze.h main.h

clean:
        -rm main *.o

.PHONY: clean
```

这就比原来简单多了。可是现在 main.o、stack.o 和 maze.o 这三个目标连编译命令都没有了，怎么编译呢？试试看：

```
$ make
cc    -c -o main.o main.c
cc    -c -o stack.o stack.c
cc    -c -o maze.o maze.c
gcc main.o stack.o maze.o -o main
```

前三条编译命令是怎么来的？如果一个目标在 Makefile 中的所有规则都没有命令列表，make 会尝试在内建的隐含规则（Implicit Rule）中查找适用的规则。make

在解析 Makefile 时会把其中的规则及变量定义与 make 内建的隐含规则及变量定义融合在一起，用 make -p 命令可以查看所有这些规则和变量定义。如果在一个没有 Makefile 的目录下使用 make -p 命令，则只显示 make 内建的隐含规则及变量定义。从 make -p 命令的输出可以看到，在我们这个例子中起作用的隐含规则及变量定义有：

```
# default
OUTPUT_OPTION = -o $@

# default
CC = cc

# default
COMPILE.c = $(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c

%.o: %.c
#  commands to execute (built-in):
        $(COMPILE.c) $(OUTPUT_OPTION) $<
```

#号在 Makefile 中表示单行注释，类似于 C 语言的//注释。CC = cc 定义一个变量 CC 并给它赋值，Makefile 变量就像环境变量或者 C 语言的宏定义一样，代表一串字符（或者空字符串），并且按照惯例通常用大写字母加下划线命名。变量名加上括号和$符（Dollar Sign）写成$(CC)表示将变量 CC 的值展开[②]，在这里要展开成 cc。在 Linux 系统上 cc 是指向 gcc 的符号链接，在其他 UNIX 系统上 cc 可能指向另外一种 C 编译器。

```
$ which cc
/usr/bin/cc
$ ls -l /usr/bin/cc
lrwxrwxrwx 1 root root 20 2010-05-01 06:30 /usr/bin/cc ->
/etc/alternatives/cc
$ ls -l /etc/alternatives/cc
lrwxrwxrwx 1 root root 12 2010-05-01 06:30 /etc/alternatives/cc ->
/usr/bin/gcc
```

CFLAGS 这个变量没有定义，$(CFLAGS)展开是空，CPPFLAGS 和 TARGET_ARCH 也是如此。这样$(COMPILE.c)展开应该是 cc␣空␣空␣空␣-c，去掉所有的“空”得到 cc␣␣␣-c，注意中间留下了 4 个空格，所以%.o: %.c 规则的命令 $(COMPILE.c)␣$(OUTPUT_OPTION)␣$<展开之后是 cc␣␣␣-c␣-o␣$@␣$<，和编译命令 cc␣␣␣-c␣-o␣main.o␣main.c 已经很接近了。

$@和$<是两个特殊的变量，$@的取值是当前规则中的目标，$<的取值是当前规则中的第一个条件。%.o: %.c 是一种特殊的规则，称为模式规则（Pattern Rule）。现在回顾一下整个过程，在我们的 Makefile 中以 main.o 为目标的规则都没有命令列表，所以 make 会查找隐含规则，发现隐含规则中有这样一条模式规则适用，main.o 符合%.o 的模式，现在%就代表 main（称为 main.o 这个名字的 Stem），替换到%.c 中就是 main.c。所以这条模式规则相当于：

② 也可以写成${CC}，在本书中都采用$(CC)这种写法。

```
main.o: main.c
        cc   -c -o main.o main.c
```

随后，在处理 stack.o 目标时又用到这条模式规则，这时又相当于：

```
stack.o: stack.c
        cc   -c -o stack.o stack.c
```

maze.o 也同样处理。这三条规则可以由 make 的隐含规则推导出来，所以不必写在 Makefile 中。

先前我们写 Makefile 都是以目标为中心，一个目标依赖于若干条件，现在换个角度，以条件为中心，Makefile 还可以这么写：

```
main: main.o stack.o maze.o
        gcc main.o stack.o maze.o -o main

main.o stack.o maze.o: main.h
main.o maze.o: maze.h
main.o stack.o: stack.h

clean:
        -rm main *.o

.PHONY: clean
```

我们知道，写规则的目的是让 make 建立依赖关系图，不管怎么写，只要把所有的依赖关系都描述清楚了就行。对于多目标的规则，make 会拆成几条单目标的规则来处理，例如：

```
target1 target2: prerequisite1 prerequisite2
        command $< -o $@
```

这样一条规则相当于：

```
target1: prerequisite1 prerequisite2
        command prerequisite1 -o target1

target2: prerequisite1 prerequisite2
        command prerequisite1 -o target2
```

注意两条规则的命令列表是一样的，但$@的取值不同。

21.3 变量

这一节我们详细讨论 Makefile 中关于变量的语法规则。先看一个简单的例子：

```
all:
        @echo $(foo)

foo = Ah $(bar)
bar = Huh?
```

执行 make 命令将会打印 Ah Huh?。

1．当 make 读到 foo = Ah $(bar)时，定义 foo 的值是 Ah $(bar)，但并不立即展开$(bar)。

2．然后读到 bar = Huh?，定义 bar 的值是 Huh?。

3．至此 Makefile 从头到尾处理了一遍，建立了规则之间的依赖关系图。现在选择适当的规则来执行，这个 Makefile 中只有一条规则，目标是 all，由于文件 all 不存在，所以执行 echo $(foo)命令更新目标 all。这时要取变量 foo 的值，展开$(foo)得到 Ah $(bar)，再展开 Ah $(bar)得到 Ah Huh?，最后执行 echo Ah Huh?。

虽然在 Makefile 中 bar 的定义写在 foo = Ah $(bar)之后，而 foo 的定义写在 echo $(foo)之后，最终还是能把$(foo)展开成 Ah $(bar)，把 Ah $(bar)再展开成 Ah Huh?。关键要理解两点：

1．Makefile 并不是从前到后顺序执行的，只有理解了 Makefile 的处理步骤，才能准确分析$(foo)在何时展开，当它要展开时已经有了哪些变量定义。

2．通过=号定义一个变量，如果=号右边有需要展开的形式（例如$(bar)），并不会在定义这个变量时立即展开，而是直到这个变量取值时（即要展开这个变量本身时）才进一步展开，也叫做递归地展开。

再举个例子：

```
main.o: main.c
        $(CC) $(CFLAGS) $(CPPFLAGS) -c $<

CC = gcc
CFLAGS = -O -g
CPPFLAGS = -DNDEBUG -Iinclude
```

编译命令可以展开成 gcc -O -g -DNDEBUG -Iinclude -c main.c。通常把变量 CFLAGS 定义成一些编译选项，而把变量 CPPFLAGS 定义成一些预处理选项，这些变量定义无论写在规则的前面还是后面，运行结果都一样。

用=号定义的变量直到取值时才递归地展开，这个特性也有坏处，就是有可能写出无穷递归的定义，例如 CFLAGS = $(CFLAGS) -O，或者：

```
A = $(B)
B = $(A)
```

当然，make 有能力检测出这样的错误而不会陷入死循环。

定义变量还可以用:=号，这样的变量在定义时就立即展开:=右边，而不是等到变量取值时再展开。例如：

```
all:
        @echo "-$(foo)-"
```

```
foo := Ah $(bar)
bar := Huh?
```

当 make 读到 foo := Ah $(bar)定义时就立即把$(bar)展开，但这时 bar 还没有定义，展开为空值，因此 foo 被定义为 Ah␣，注意 Ah 和$(bar)中间的空格仍保留在定义中。接下来读到 bar := Huh?，bar 被定义为 Huh?，但已经影响不到 foo 的定义了。最终打印的结果是-Ah␣-。读者可以试试把 foo 和 bar 的定义颠倒过来看看是什么结果。

在一个变量的定义中，从=号或:=号右边的第一个非空白字符开始，直到注释或换行之前的所有字符都属于这个变量的值。例如上面的 foo := Ah $(bar)这个定义，Ah 左边的空格不算数，而 Ah 右边的空格要算数。再举个例子，如果要定义一个变量的值是个空格，可以这样：

```
nullstring :=
space := $(nullstring) # end of the line
```

在 space := $(nullstring) # end of the line 这个定义中，$(nullstring)展开为空，#号后边是注释，所以 space 的值是$(nullstring)和#之间的那个空格。写注释是为了增加可读性，如果不写注释就换行，很难看出$(nullstring)和换行之间有一个空格。

除了=号和:=号之外，还可以用?=号定义变量。例如 foo ?= $(bar)的意思是：如果 foo 没有定义过，那么?=号相当于=号，定义 foo 的值是$(bar)，但不立即展开；如果先前已经定义了 foo，则什么也不做，不会给 foo 重新赋值。

可以用+=号给变量追加值，例如：

```
objects = main.o
objects += $(foo)
```

objects 是用=号定义的，+=号仍然保持=号的特性，objects 的值是 main.o␣$(foo)（注意$(foo)前面自动添加一个空格），但不立即展开。再比如：

```
objects := main.o
objects += $(foo)
```

objects 是用:=号定义的，+=号保持:=号的特性，objects 用+=号赋值是 main.o␣$(foo)，这时 foo 还没有定义，立即展开得到 main.o␣，注意 main.o 后面的空格仍保留。

如果变量还没有定义过就直接用+=号赋值，那么+=号相当于=号，读者可以自己试试，这里就不举例了。

上一节我们用到了特殊变量$@和$<，这两个变量的特点是不需要给它们赋值，在不同的上下文中它们自动取不同的值，所以也叫自动变量。常用的自动变量有：

- $@，表示规则中的目标。

- $*，表示模式规则中的 Stem。
- $<，表示规则中的第一个条件。
- $?，表示规则中所有比目标新的条件，组成一个列表，以空格分隔。
- $^，表示规则中的所有条件，组成一个列表，以空格分隔，如果这个列表中有重复的项则消除重复的项。

例如前面写过的这条规则：

```
main: main.o stack.o maze.o
        gcc main.o stack.o maze.o -o main
```

可以改写成：

```
main: main.o stack.o maze.o
        gcc $^ -o $@
```

即使以后又往条件里添加了新的目标文件，$^也能自动包含新的目标文件名，编译命令不需要修改，因而减少了人为出错的可能。$?变量也很有用，有时候希望只对更新过的条件进行操作，例如有一个库文件 libsome.a 依赖于几个目标文件：

```
libsome.a: foo.o bar.o lose.o win.o
        ar r libsome.a $?
        ranlib libsome.a
```

只有更新过的目标文件才需要重新打包到 libsome.a 中，没更新过的目标文件原本已经在 libsome.a 中了，不必重新打包，所以这里用$?比较合适。

在上一节我们看到 make 的隐含规则数据库中用到了很多变量，有些变量没有定义（例如 CFLAGS），有些变量定义了缺省值（例如 CC），我们写 Makefile 时可以重新定义这些变量的值，也可以在缺省值的基础上追加。以下列举一些常用的变量，请读者体会这些变量的命名规律。

AR

静态库打包命令的名字，缺省值是 ar。

ARFLAGS

静态库打包命令的选项，缺省值是 rv。

AS

汇编器的名字，缺省值是 as。

ASFLAGS

汇编器的选项，没有定义。

CC

C 编译器的名字，缺省值是 cc。

CFLAGS

C 编译器的选项，没有定义。

CXX

C++编译器的名字，缺省值是 g++。

CXXFLAGS

C++编译器的选项，没有定义。

CPP

C 预处理器的名字，缺省值是$(CC) -E。

CPPFLAGS

C 预处理器的选项，没有定义。

LD

链接器的名字，缺省值是 ld。

LDFLAGS

链接器的选项，没有定义。

TARGET_ARCH

和目标平台相关的命令行选项，没有定义。

OUTPUT_OPTION

输出的命令行选项，缺省值是-o $@。

LINK.o

把.o 文件链接在一起的命令行，缺省值是$(CC) $(LDFLAGS) $(TARGET_ARCH)。

LINK.c

把.c 文件链接在一起的命令行，缺省值是$(CC) $(CFLAGS) $(CPPFLAGS) $(LDFLAGS) $(TARGET_ARCH)。

LINK.cc

把.cc 文件（C++源文件）链接在一起的命令行，缺省值是$(CXX) $(CXXFLAGS) $(CPPFLAGS) $(LDFLAGS) $(TARGET_ARCH)。

COMPILE.c

编译.c 文件的命令行，缺省值是$(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c。

COMPILE.cc

编译.cc 文件的命令行，缺省值是$(CXX) $(CXXFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c。

RM

删除命令的名字，缺省值是 rm -f。

21.4　自动处理头文件的依赖关系

现在我们的 Makefile 写成这样：

```
all: main

main: main.o stack.o maze.o
        gcc $^ -o $@

main.o: main.h stack.h maze.h
stack.o: stack.h main.h
maze.o: maze.h main.h

clean:
        -rm main *.o

.PHONY: clean
```

按照惯例，用 all 做缺省目标。还有一个问题比较麻烦，在写 main.o、stack.o 和 maze.o 这三个目标的规则时要查看源代码，找出它们依赖于哪些头文件，这很容易出错，一是因为有的头文件包含在另一个头文件中，在写规则时很容易遗漏，二是如果以后修改源代码改变了依赖关系，很可能忘记修改 Makefile 的规则。

在源代码中已经包含了目标文件和头文件之间的依赖关系，这种依赖关系是以#include 的形式描述的，如果在 Makefile 中以规则的形式再描述一遍，就存在重复信息了。手工维护重复信息很容易出错，应该想办法自动维护。以前我们讲过定义一个宏然后在代码中多处引用它可以避免硬编码，讲过写一个头文件然后包含在很多.c 文件中可以避免在这些.c 文件中重复声明，道理都是一样的：如果某信息在多处重复出现，要修改它应该只在一个地方手工修改，而在其他地方自动获得更新后的信息，这称为 DRY（Don't Repeat Yourself）原则。现在我们要想办法把源代码中的依赖关系信息抽取出来自动转换成 Makefile 中的规则。第一步，用 gcc 的-M 选项自动分析目标文件和源文件的依赖关系，以 Makefile 规则的格式输出：

```
$ gcc -M main.c
```

```
main.o: main.c /usr/include/stdio.h /usr/include/features.h \
 /usr/include/bits/predefs.h /usr/include/sys/cdefs.h \
 /usr/include/bits/wordsize.h /usr/include/gnu/stubs.h \
 /usr/include/gnu/stubs-32.h \
 /usr/lib/gcc/i486-linux-gnu/4.4.3/include/stddef.h \
 /usr/include/bits/types.h /usr/include/bits/typesizes.h \
 /usr/include/libio.h /usr/include/_G_config.h /usr/include/wchar.h \
 /usr/lib/gcc/i486-linux-gnu/4.4.3/include/stdarg.h \
 /usr/include/bits/stdio_lim.h   /usr/include/bits/sys_errlist.h
main.h \
 stack.h maze.h
```

注意在 Makefile 的规则中也可以使用类似 C 语言的续行符\。gcc -M 的输出结果不仅包括我们自己写的头文件 main.h、stack.h 和 maze.h，还包括 stdio.h 和其他系统头文件，因为我们的程序中包含了 stdio.h，而后者又包含了其他系统头文件。系统头文件通常不需要随我们的程序一起维护，所以通常不用 gcc 的-M 选项而是用-MM 选项，输出结果中只包括我们自己写的头文件：

```
$ gcc -MM *.c
main.o: main.c main.h stack.h maze.h
maze.o: maze.c maze.h main.h
stack.o: stack.c stack.h main.h
```

接下来的问题是怎么把这些规则添加到 Makefile 中，Scott McPeak 提供了一个很好的解决方案（http://scottmcpeak.com/autodepend/autodepend.html）：

```
all: main

main: main.o stack.o maze.o
        gcc $^ -o $@

clean:
        -rm main *.o *.d

.PHONY: clean

OBJS = main.o stack.o maze.o

-include $(OBJS:.o=.d)

%.o: %.c
        gcc -c $(CFLAGS) $*.c -o $*.o
        gcc -MM $(CPPFLAGS) $*.c > $*.d
        mv -f $*.d $*.d.tmp
        sed -e 's|.*:|$*.o:|' < $*.d.tmp > $*.d
        sed -e 's/.*://' -e 's/\\$$//' < $*.d.tmp | fmt -1 | \
        sed -e 's/^ *//' -e 's/$$/:/' >> $*.d
      rm -f $*.d.tmp
```

OBJS 变量的值是我们要编译生成的.o 文件的列表，$(OBJS:.o=.d)是变量值的替换语法，把 OBJS 变量中每一项的.o 后缀替换成.d 后缀，所以 include 这一句相当于：

```
-include main.d stack.d maze.d
```

类似于 C 语言的#include 预处理指示，在 Makefile 中 include main.d stack.d maze.d

表示把这三个文件包含到当前的Makefile中，这三个文件也应该符合Makefile的语法。我们在include前面加了一个-，表示如果它要包含的文件列表中有的文件不存在，则忽略不存在的文件，只包含存在的文件。如果include前面没有-，而它要包含的文件又不存在，则make会报错。

事实上，include不只是把文件包含进来这么简单，还要做一个特殊处理：对于要包含进来的每一个文件，make会把文件名当做目标来尝试更新（如果有规则匹配该目标则执行该规则下的命令，如果没有规则能匹配该目标就算了），如果make检查完所有要包含进来的文件后，其中确实有些文件被更新了，则make会从头开始执行，重新包含更新后的文件。详见参考文献[27]的3.7节。对于我们这个例子不必考虑这种复杂的情况，因为我们没有规则能匹配以.d为后缀的目标，不会更新main.d、stack.d或maze.d。

下面我们分几个不同的场景来分析当.c和.h文件的依赖关系发生变化时make如何更新规则来适应这些变化。

1. 如果你的工作目录是干净的，只有.c文件、.h文件和Makefile文件，执行make命令的结果是：

```
$ make
gcc -c  main.c -o main.o
gcc -MM  main.c > main.d
mv -f main.d main.d.tmp
sed -e 's|.*:|main.o:|' < main.d.tmp > main.d
sed -e 's/.*://' -e 's/\\$//' < main.d.tmp | fmt -1 | \
          sed -e 's/^ *//' -e 's/$/:/' >> main.d
rm -f main.d.tmp
gcc -c  stack.c -o stack.o
gcc -MM  stack.c > stack.d
mv -f stack.d stack.d.tmp
sed -e 's|.*:|stack.o:|' < stack.d.tmp > stack.d
sed -e 's/.*://' -e 's/\\$//' < stack.d.tmp | fmt -1 | \
          sed -e 's/^ *//' -e 's/$/:/' >> stack.d
rm -f stack.d.tmp
gcc -c  maze.c -o maze.o
gcc -MM  maze.c > maze.d
mv -f maze.d maze.d.tmp
sed -e 's|.*:|maze.o:|' < maze.d.tmp > maze.d
sed -e 's/.*://' -e 's/\\$//' < maze.d.tmp | fmt -1 | \
          sed -e 's/^ *//' -e 's/$/:/' >> maze.d
rm -f maze.d.tmp
gcc main.o stack.o maze.o -o main
```

由于main.d、stack.d和maze.d一个都不存在，所以include忽略它们，不包含任何文件。然后根据%.o: %.c规则，我们要更新目标main.o、stack.o和maze.o，它们的处理过程是一样的，我们以main.o为例解释一下具体的步骤：

① 执行gcc -c main.c -o main.o，由main.c编译出main.o。

② 执行gcc -MM main.c > main.d，把main.c的依赖关系写入文件main.d，其内容是main.o: main.c main.h stack.h maze.h。这条规则在本次make的处理过程中不

起作用，但下次执行 make 时会把这个 main.d 包含进来，这条规则就会起作用了。上一条命令生成 main.o，而这一条命令生成 main.d，也就是说：只要生成一个.o 文件，就要配合它生成一个.d 文件，以便下次 make 时可以根据.d 文件中的规则检查这个.o 文件需不需要更新。

③ 由 gcc -MM 生成的 main.d 还不能完全满足我们的要求，接下来的几条命令把 main.d 中的规则改成这样：

```
main.o: main.c main.h stack.h maze.h
main.c:
main.h:
stack.h:
maze.h:
```

我们稍后会分析为什么要改成这样。sed 和 fmt 命令的用法是另一个复杂的话题，在这里就不细讲了，sed 的作用是在文件中做编辑、查找、替换，fmt 的作用是段落排版。

2. 如果修改了头文件 stack.h 再重新 make，执行结果是：

```
$ make
gcc -c  main.c -o main.o
gcc -MM  main.c > main.d
mv -f main.d main.d.tmp
sed -e 's|.*:|main.o:|' < main.d.tmp > main.d
sed -e 's/.*://' -e 's/\\$//' < main.d.tmp | fmt -1 | \
          sed -e 's/^ *//' -e 's/$/:/' >> main.d
rm -f main.d.tmp
gcc -c  stack.c -o stack.o
gcc -MM  stack.c > stack.d
mv -f stack.d stack.d.tmp
sed -e 's|.*:|stack.o:|' < stack.d.tmp > stack.d
sed -e 's/.*://' -e 's/\\$//' < stack.d.tmp | fmt -1 | \
          sed -e 's/^ *//' -e 's/$/:/' >> stack.d
rm -f stack.d.tmp
gcc main.o stack.o maze.o -o main
```

由于 Makefile 中包含了 main.d 和 stack.d，也就是包含了规则 main.o: main.c main.h stack.h maze.h 和 stack.o: stack.c stack.h main.h，所以修改 stack.h 将导致 main.o 和 stack.o 被更新，但这两条规则没有命令列表，怎么更新呢？注意到它们和%.o: %.c 规则属于“一个目标拆开写多条规则”的情况，所以会执行%.o: %.c 规则的命令列表。最后，main.o 和 stack.o 被更新又导致可执行文件 main 被更新。

3. 如果添加了一个头文件 foo.h，并且在 main.c 中加了一行#include "foo.h"，再重新 make，执行结果是：

```
$ make
gcc -c  main.c -o main.o
gcc -MM  main.c > main.d
mv -f main.d main.d.tmp
sed -e 's|.*:|main.o:|' < main.d.tmp > main.d
sed -e 's/.*://' -e 's/\\$//' < main.d.tmp | fmt -1 | \
      sed -e 's/^ *//' -e 's/$/:/' >> main.d
```

```
rm -f main.d.tmp
gcc main.o stack.o maze.o -o main
```

根据规则%.o: %.c，修改main.c导致main.o被更新，在执行命令列表时重新生成了main.d，其内容为：

```
main.o: main.c main.h stack.h maze.h foo.h
main.c:
main.h:
stack.h:
maze.h:
foo.h:
```

把foo.h也包含到规则里了，下次如果修改foo.h并重新make，也会导致main.o被更新。

4．如果删除头文件foo.h再重新make，执行结果是：

```
$ make
gcc -c  main.c -o main.o
main.c:6:17: error: foo.h: No such file or directory
make: *** [main.o] Error 1
```

我们知道Makefile中包含了规则main.o: main.c main.h stack.h maze.h foo.h，但foo.h已经不存在了，make不是应该报错吗？为什么还会执行%.o: %.c的命令列表呢？

在这种情况下另外一条规则 foo.h:起作用了，像这种既没有依赖条件也没有命令列表的规则是这样处理的：如果以该目标作为文件名的文件不存在，则认为该目标需要更新，由于没有命令列表，执行该规则并不会生成文件，但会认为该目标已被更新，因此依赖该目标的其他目标也要被更新。详见参考文献[27]的4.6节。因此，foo.h不存在将导致main.o被更新。

21.5 常用的make命令行选项

-n选项只打印要执行的命令，而不会真的执行命令（这称为Dry Run），这个选项有助于我们检查Makefile写得是否正确，由于Makefile不是顺序执行的，用这个选项可以先看看命令的执行顺序，确认无误了再真正执行命令。

-C选项可以切换到另一个目录执行那个目录下的Makefile，比如我们的源代码都放在/home/akaedu/testmake目录，可以在任何目录下敲make命令进入我们的源代码目录编译，编译完成后仍退回到先前的目录：

```
$ make -C /home/akaedu/testmake
make: Entering directory `/home/akaedu/testmake'
cc    -c -o main.o main.c
cc    -c -o stack.o stack.c
cc    -c -o maze.o maze.c
gcc main.o stack.o maze.o -o main
make: Leaving directory `/home/akaedu/testmake'
```

一些规模较大的项目会把不同的模块或子系统的源代码放在不同的子目录中，每个子目录下都写一个Makefile，然后在一个总的Makefile中用make -C命令执行每个子目录下的Makefile。例如Linux内核源代码根目录下有Makefile，子目录fs、net等也有各自的Makefile，二级子目录fs/ramfs、net/ipv4等也有各自的Makefile。

在make命令行也可以用=或:=定义变量，如果这次编译我想加调试选项-g，但我不想每次编译都加-g，可以在命令行定义CFLAGS变量而不必修改Makefile：

```
$ make CFLAGS=-g
cc -g   -c -o main.o main.c
cc -g   -c -o stack.o stack.c
cc -g   -c -o maze.o maze.c
gcc main.o stack.o maze.o -o main
```

make进程中的环境变量也可以起到Makefile变量的作用。比如有这样的Makefile：

```
foo = 1
all:
        @echo $(foo)
```

如果先在Shell进程中定义一个环境变量foo，再执行make命令，这个环境变量会传给make进程，相当于在Makefile中定义了变量foo。

```
$ export foo=2
$ make
1
```

但运行结果却不是2，这是因为Makefile中已经定义了变量foo，默认情况下Makefile中的定义会覆盖环境变量的定义，如果希望环境变量的定义覆盖Makefile中的定义可以用-e选项：

```
$ make -e
2
```

在make的命令行选项中定义的变量优先级最高，会覆盖环境变量的定义和Makefile中的定义：

```
$ make foo=3
3
```

如果把foo=3写在make前面，则foo=3是Shell进程传给make进程的环境变量，而不是命令行选项，注意区分这两种写法。

```
$ foo=3 make
1
$ foo=3 make -e
3
```

刚才讲过在一些规模较大的项目中每个目录下都会有一个Makefile，一般是在上层目录的Makefile里用make -C命令执行下层目录的Makefile。我们可以在上层目录的Makefile里用export声明一些变量，这些变量会自动传给make -C命令做

环境变量。例如上层目录的 Makefile 是这样的：

```
foo = string1
bar = string2
export foo
all:
        $(MAKE) -C subdir
```

注意这里的 export 不是 Shell 命令，而是 Makefile 中的声明，因为我把 export 顶头写，开头没有 Tab 字符。在子目录 subdir 中有这样的 Makefile：

```
all:
        @echo $(foo) $(bar)
```

在上层目录中执行 make 的结果是：

```
$ make
make -C subdir
make[1]: Entering directory `/home/akaedu/testmake/subdir'
string1
make[1]: Leaving directory `/home/akaedu/testmake/subdir'
```

有一个例外，在命令行选项中定义的变量不需要 export 就可以传给 Makefile 里的 make -C 命令：

```
$ make bar=str2
make -C subdir
make[1]: Entering directory `/home/akaedu/testmake/subdir'
string1 str2
make[1]: Leaving directory `/home/akaedu/testmake/subdir'
```

第 22 章

指针

22.1 指针的基本概念

在第 12 章讲过，堆栈有栈顶指针，队列有头指针和尾指针，这些概念中的“指针”本质上是一个整数，是数组的索引，通过指针访问数组中的某个元素。在图 19.3 中我们又看到另外一种指针的概念，把一个变量所在的内存单元的地址保存在另外一个内存单元中，保存地址的这个内存单元称为指针（Pointer），访问变量要通过指针间接寻址，这种指针在 C 语言中可以用一个指针类型的变量表示，例如某程序中定义了以下全局变量：

```
int i;
int *pi = &i;
char c;
char *pc = &c;
```

这几个变量的内存布局如图 22.1 所示，在初学阶段经常要借助于这样的图来理解指针。

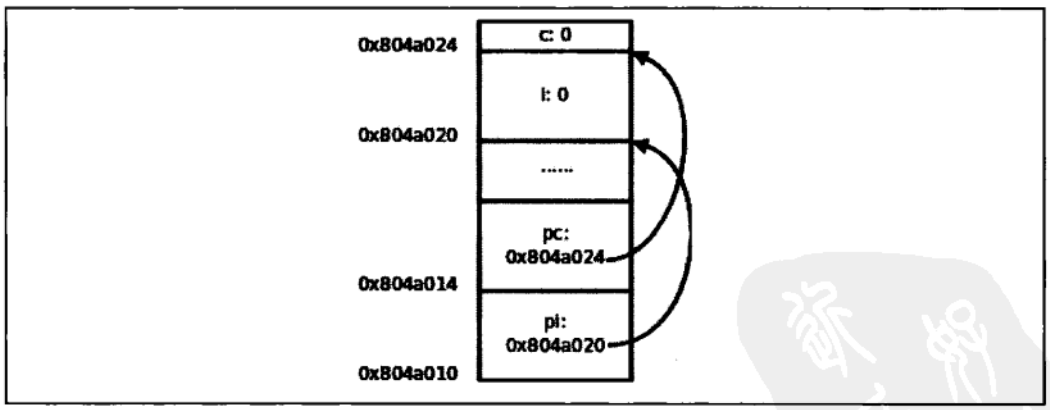


图 22.1 指针的基本概念

这里的&是取地址运算符（Address Operator），&i 表示取变量 i 的地址，int *pi = &i;表示定义一个指向 int 型的指针变量 pi，并用 i 的地址来初始化 pi。我们讲过全局变量只能用常量表达式来初始化，如果定义 int p = i;就错了，因为 i 不是常量表达式。然而用 i 的地址来初始化一个指针却没有错，因为 i 的地址是在编译链接时确定的，不需要到运行时才知道，&i 是常量表达式。

后面两行代码定义了一个字符型变量 c 和一个指向 c 的字符型指针 pc，注意 pi 和 pc 虽然是不同类型的指针变量，但它们的内存单元都占 4 个字节，因为要保存 32 位的虚拟地址，同理，在 64 位平台上指针变量都占 8 个字节。

如果把多个数组放在一起声明，每个数组名后面都要有[]括号：int a[5], b[5];。同样道理，如果把多个指针变量放在一起声明，每个变量名前面都要有*号。例如：

```
int *p, *q;
```

如果写成 int* p, q;就错了，这是声明了一个整型指针 p 和一个整型变量 q，这样写很容易看错，比较好的写法是*号和前面的类型 int 之间留空格，和后面的变量名写在一起，写成 int *p, q;就不容易看错了，很明显这是定义了一个指针和一个整型变量。

如果要让 pi 指向另一个整型变量 j，可以重新对 pi 赋值：

```
pi = &j;
```

现在要通过指针 pi 间接寻址到变量 j，把变量 j 的值增加 10，可以写成：

```
*pi = *pi + 10;
```

这里的*号是指针间接寻址运算符（Indirection Operator），*pi 表示取指针 pi 所指向的变量的值，也称为 Dereference 操作，指针有时称为变量的引用（Reference），所以根据指针找到变量称为 Dereference。

我们知道[]括号用在声明中和用在表达式中有不同的含义，[]括号用在声明中表示声明一个数组，用在表达式中是取下标运算符。同样道理，*号用在声明中表示声明一个指针类型，用在表达式中是间接寻址运算符。

*和&互为逆运算。&运算符的操作数必须是左值，因为只有左值才表示一个内存单元，才会有地址，运算结果是指针类型。*运算符的操作数必须是指针类型，运算结果可以做左值。所以，如果表达式 E 可以做左值，*&E 和 E 等价，如果表达式 E 是指针类型，&*E 和 E 等价。

指针之间可以相互赋值，也可以用一个指针初始化另一个指针，例如：

```
int *ptri = pi;
```

或者：

```
int *ptri;
ptri = pi;
```

表示 **pi 指向哪就让 ptri 也指向哪**，本质上就是把变量 pi 所保存的地址值赋给变量 ptri。

用一个指针给另一个指针赋值时要注意，两个指针必须是同一类型的。在我们的

例子中，pi是int *型的，pc是char *型的，pi = pc;这样赋值就是错误的。但是可以先强制类型转换然后赋值，如图22.2所示。

```
pi = (int *)pc;
```

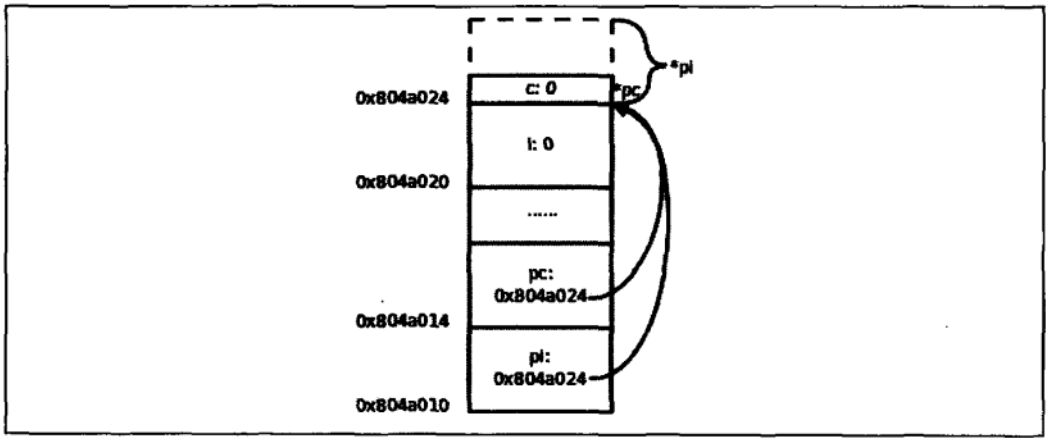


图22.2 把char *指针的值赋给int *指针

现在pi指向的地址和pc一样，但是通过*pc只能访问到一个字节，而通过*pi可以访问到4个字节，后3个字节已经不属于变量c了，除非你很确定变量c的一个字节和后面3个字节组合而成的int值是有意义的，否则就不应该给pi这么赋值。因此使用指针要特别小心，很容易将指针指向错误的地址，访问这样的地址可能导致段错误，可能读到无意义的值，也可能意外改写了某些数据，使得程序在随后的运行中出错。有一种情况需要特别注意，定义一个指针类型的局部变量而没有初始化：

```
int main(void)
{
        int *p;
        ...
        *p = 0;
        ...
}
```

我们知道，在栈上分配的变量初始值是不确定的，也就是说指针p所指向的内存地址是不确定的，后面用*p访问不确定的地址会导致不确定的后果，可能引发段错误，也可能意外改写了数据而导致程序在随后的运行中出错，而且后一种情况很难找到错误原因。像这种指向不确定地址的指针称为“野指针”（Unbound Pointer），为避免出现野指针，在定义指针变量时就应该明确地给它赋初值，或者把它初始化成NULL：

```
int main(void)
{
        int *p = NULL;
        ...
        *p = 0;
        ...
}
```

NULL 在 C 标准库的头文件 stddef.h 中定义：

```
#define NULL ((void *)0)
```

指针也是一种标量类型，可以用()运算符做强制类型转换，其他标量类型可以转成指针类型，指针类型也可以转成其他标量类型，比如在上面的定义中把整型的 0 强制转换成 void *指针，这个指针指向 0 地址，称为空指针。操作系统不会把任何数据保存在地址 0 及其附近，也不会把地址 0～0xfff 的页面映射到物理内存，所以任何对 0 地址的访问（比如上面的*p = 0;）一定会引发段错误，这种错误就像放在眼前的炸弹一样很容易找到，相比之下，由野指针引起的错误就像埋下地雷一样，更难发现和排除，这次走过去没事，下次走过去就有事。

由于指针是标量类型，因此可以做逻辑与、或、非运算的操作数和 if、for、while 的控制表达式，例如 if (p) 语句 A else 语句 B，如果指针 p 非空则执行语句 A，如果是空指针则执行语句 B。

讲到这里就该讲一下 void *类型了。在编程时经常需要一种通用指针，可以转换成任意其他类型的指针，任意其他类型的指针也可以转换成通用指针，最初 C 语言没有 void *类型，就把 char *当做通用指针，需要转换时就用()运算符强制转换，ANSI 在把 C 语言标准化时引入了 void *类型，规定 void *指针与其他类型的指针之间可以隐式转换，而不必用()运算符强制转换。注意，只能定义 void *类型的指针，而不能定义 void 类型的变量，因为 void *指针和别的指针一样都占 4 个字节，而如果定义 void 型变量（即类型不确定的变量），编译器不知道该给这个变量分配几个字节。同样道理，void *指针不能直接 Dereference，而必须先转换成别的类型的指针再做 Dereference。void *指针常用于函数传参和传返回值，下一章讲函数接口时再分析 void *指针的作用。

习题

1. 思考一下，为什么要用 typedef 定义类型名而不用#define 定义类型名？比如：

```
typedef int *pint_t;
#define pint_t int *
```

这两种定义用起来有区别吗？

22.2 指针类型的参数和返回值

首先看以下程序：

例 22.1 指针参数和返回值

```
#include <stdio.h>

int *swap(int *px, int *py)
{
```

```
        int temp;
        temp = *px;
        *px = *py;
        *py = temp;
        return px;
}

int main(void)
{
        int i = 10, j = 20;
        int *p = swap(&i, &j);
        printf("now i=%d j=%d *p=%d\n", i, j, *p);
        return 0;
}
```

我们知道，调用函数的传参过程相当于定义形参变量并且用实参的值来初始化，swap(&i, &j)这个调用相当于：

```
int *px = &i;
int *py = &j;
```

所以px和py分别指向main函数的局部变量i和j，在swap函数中读写*px和*py其实是读写main函数的局部变量i和j。尽管swap函数在它的作用域中访问不到i和j这两个变量名，却可以通过指针间接寻址到这两个变量，并且将它们的值做了交换。

上面的例子还演示了函数返回值是指针的情况，return px;语句相当于定义一个临时变量并且用px初始化：

```
int *tmp = px;
```

然后临时变量tmp的值成为表达式swap(&i, &j)的值，然后在main函数中又把这个值赋给了p，相当于：

```
int *p = tmp;
```

最后的结果是swap函数的px指向哪就让main函数的p指向哪。我们知道px指向i，所以p也指向i。

通过函数返回值传指针有一种常见的错误，如下面的代码所示：

```
int *foo(void)
{
        int a;
        ...
        return &a;
}

int main(void)
{
        int *p = foo();
        ...
}
```

foo 函数的返回值是一个指向局部变量 a 的指针，可是 foo 函数返回之后它的栈帧就要释放掉，要这个指针还有什么用呢？这其实也是一种野指针。上面的代码错得比较明显，编译器会报警告 warning: function returns address of local variable，有时候错得不那么明显，例如：

```
int *foo(void)
{
        int a;
        int *pa = &a;
        ...
        return pa;
}
```

编译器是不会报警告的，这就需要程序员自己多加小心了。

习题

1．对照本节的描述，像图 22.1 那样画图理解函数的调用和返回过程。在下一章我们会看到更复杂的参数和返回值形式，在初学阶段对每个程序都要画图理解它的运行过程，只要基本概念清晰，无论多复杂的形式都应该能正确分析。

2．现在回头看第 3.3 节的习题 1，那个程序应该怎么改？

22.3　指针与数组

先看一个例子，有如下代码：

```
int a[10];
int *pa = &a[0];
pa++;
```

一开始指针 pa 里保存的是数组元素 a[0]的地址，注意后缀运算符的优先级高于单目运算符，所以&a[0]是取 a[0]的地址而不是取 a 的地址。然后 pa++让 pa 指向下一个数组元素 a[1]，由于 pa 是 int *指针，一个 int 型元素占 4 个字节，所以 pa++使 pa 里保存的地址值加 4 而不是加 1。

下面画图理解。从图 22.1 可以看出来，地址的具体数值其实无关紧要，关键是要说明地址之间的关系（数组 a 的每个元素占 4 个字节的内存单元，a[1]紧挨在 a[0]之后）以及指针与变量之间的关系（指针里保存的是变量的地址），现在我们换一种画法，省略地址的具体数值，用方框表示内存单元，用箭头表示指针和变量之间的关系，如图 22.3 所示。

既然指针可以用++运算符，当然也可以用+和-运算符，pa+2 这个表达式也是有意义的，如图 22.3 所示，pa 指向 a[1]，那么 pa+2 指向 a[3]。事实上，E1[E2]这种写法和(*((E1)+(E2)))是等价的，*(pa+2)也可以写成 pa[2]，pa 可以像数组名一样使用。

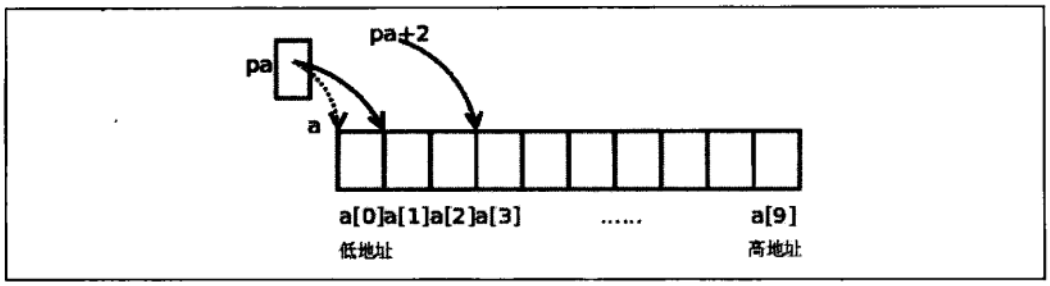


图 22.3 指针与数组

其实对于*和[]运算符来说数组类型和指针是统一的，在第 8.1 节讲过数组类型做右值时自动转换成指向首元素的指针，a 做右值时自动转换成指向 a[0]的 int *指针，所以 a[2]也可以写成*(a+2)，本质上都是通过指针间接寻址访问元素。由于(*((E1)+(E2)))显然可以写成(*((E2)+(E1)))，所以 E1[E2]也可以写成 E2[E1]，这意味着 2[a]、2[pa]这种写法也是对的，但一般不这么写。另外，由于 a 做右值使用时和&a[0]是一个意思，所以 int *pa = &a[0];通常写成更简洁的形式 int *pa = a;。

在第 8.1 节还讲过 C 语言允许数组下标是负数，现在你该明白为什么这样规定了。在上面的例子中，表达式 pa[-1]（或者写成*(pa-1)）是合法的，它和 a[0]表示同一个元素。

现在猜一下，两个指针变量做比较运算（>>= <<= == !=）表示什么意义？两个指针变量做减法运算又表示什么意义？

根据什么来猜？根据第 3.3 节讲过的 Rule of Least Surprise 原则。你理解了指针和常数相加减的概念，再根据以往使用比较运算的经验，就应该猜到 pa + 2 > pa，pa - 1 == a，所以指针之间的比较运算比的是地址，C 语言正是这样规定的，不过 C 语言的规定更为严谨，只有指向同一个数组中元素的指针之间相互比较才有意义，也只有指向数组元素的指针加或减一个常数才有意义，否则都是 Undefined。那么两个指针相减表示什么？pa - a 等于几？因为 pa - 1 == a，所以 pa - a 显然应该等于 1，而 a - pa 应该等于-1，指针相减表示两个指针之间相差的元素个数，同样只有指向同一个数组中元素的指针之间相减才有意义，相减的结果是 ptrdiff_t 类型，这个类型名在 stddef.h 中定义，是一种有符号整型。

两个指针相加表示什么？想不出来能有什么意义，因此 C 语言也规定两个指针不能相加。假如 C 语言为指针相加也规定了一种意义，那就相当 Surprise 了，不符合一般的经验。无论是设计编程语言还是设计函数接口或人机界面都是这个道理，应该尽量让用户根据以往的经验知识就能推断出该系统的基本用法。

数组类型做右值时和指针的语法是统一的，但如果把数组类型做左值使用，和指针就有区别了。例如++pa 是合法的，但++a 就不合法，pa = a + 1 是合法的，但 a = pa + 1 就不合法。数组类型做左值时表示整个数组的存储空间，++a 相当于 a += 1，+=左右两边的操作数类型不符，不能给整个数组加上 1，就像不能给整个结构体加上 1 一样，因为它们不是标量类型，同样道理 a = pa + 1 也是左右两边类型不符，=号左边是数组类型，而=号右边是指针类型。数组类型做左值支持&运算

符，所以&a是合法的，这个表达式的类型是指向数组的指针，我们将在第22.7节介绍这种类型。另外要注意，sizeof a这个表达式中数组名a不是做右值，计算结果是整个数组的存储空间大小，而sizeof pa求一个指针pa的存储空间大小，计算结果应该是4个字节。

在第8.1节讲过，在函数原型中，如果参数写成数组的形式，则该参数实际上是指针类型。例如：

```
void func(int a[10])
{
        ...
}
```

其实等价于：

```
void func(int *a)
{
        ...
}
```

第一种形式方括号中的数字也可以不写，仍然是等价的：

```
void func(int a[])
{
        ...
}
```

参数写成指针形式还是数组形式对编译器来说没区别，这个参数都是指针类型，之所以规定两种形式是为了给读代码的人提供有用的信息，如果这个参数指向一个元素，通常写成指针的形式，如果这个参数指向一串元素中的首元素，则经常写成数组的形式。

习题

1. 定义一个二维数组，例如int a[][5] = {23, 43, 4, 5, 7, 45, 42, 12, 11, 66, 55, 47, 33, 89, 1};，编写程序将里面的元素排序。

2. 表达式*p++应该怎么理解？

3. 请写出下面程序的运行结果，然后上机运行看你写得对不对，通过辨析sizeof的用法理解本节的基本概念。

```
#include <stdio.h>

void foo(char buf3[20])
{
        printf("%u\n", sizeof(buf3));
}

int main(void)
{
```

```
    char *buf = "hello world";
    char buf2[20] = "hello world";

    printf("%u\n%u\n%u\n", sizeof(buf), sizeof(buf2),
          sizeof("hello world"));
    foo("hello world");

    return 0;
}
```

22.4 指针与const限定符

const限定符和指针结合起来常见的情况有以下几种。

- 指针类型声明的*号前面加const。例如：

```
const int *a;
int const *a;
```

这两种写法一样，a是一个指向const int型的指针，a所指向的内存单元不可改写，所以(*a)++是不允许的，但指针a本身可以改写，即a可以指向别的内存单元，a++是允许的。

- 指针类型声明的*号后面加const。例如：

```
int *const a;
```

a是一个指向int型的const指针，*a是可以改写的，但a本身不允许改写。

- 指针类型声明的*号前后都加const。例如：

```
int const *const a;
const int *const a;
```

这两种写法一样，a是一个指向const int型的const指针，*a和a本身都不允许改写。

在赋值、初始化或函数调用传参过程中，假设要把指针p赋给指针q，应注意const限定符的语义作用：

- 如果p指向的类型不带有const限定，而q指向的类型带有const限定，则可以把p赋给q，例如：

```
char c = 'a';
const char *q = &c;
```

表达式&c是char *型的，而q是const char *型的，可以赋值，赋值后*q不允许改写，即不能通过指针q来改写变量c的值。

- 如果p指向的类型带有const限定，而q指向的类型不带有const限定，

则不能把 p 赋给 q，例如对下面的代码编译器会报警告：

```
const char c = 'a';
char *q = &c;
```

表达式&c 是 const char *型的，而 q 是 char *型的，假如允许赋值，则赋值后就可以通过指针 q 来改写变量 c 的值，等于绕过了变量 c 的 const 限定这道防线，这是很危险的，如果变量 c 被分配在.rodata 段，改写它会导致段错误，因此编译器不允许这样赋值。

- 如果 p 和 q 指向的类型都带有 const 限定，或者都不带有 const 限定，则可以把 p 赋给 q，这是理所当然的。

事实上，在指针赋值过程中 volatile 和 restrict 限定符也有和 const 限定符类似的作用，用一句话概括就是：**要想把指针 p 赋给指针 q，q 指向的类型应该比 p 指向的类型限定得更严格，或至少是同样严格，而不能比 p 指向的类型限定得更宽松。**

即使不用 const 限定符也能写出功能正确的程序，但良好的编程习惯应该尽量使用 const 限定符，理由是：

1．const 给读代码的人传达非常有用的信息。比如一个函数的形参是 const char *指针，你在调用这个函数时就可以放心地传给它 char *或 const char *指针，而不必担心指针所指的内存单元被改写，因为该函数无法通过它的形参改写你传给它的内存单元。

2．把程序中不该变的变量都加上 const 限定符，可以依靠编译器检查程序的 Bug，防止运行时意外改写了不该变的数据。

3．const 对编译器优化是一个有用的提示，编译器也许会把 const 变量优化成常量。

在第 8.4 节提到，字符串字面值和数组类型相似，做右值使用时自动转换成指向首元素的指针，字符串中的元素是 char 型的，因此指向首元素的指针应该是 char *型。

如果改写字符串字面值中的元素会怎么样呢？对此 C 标准没有明确规定（Undefined）。在第 8.4 节讲过，如果在代码中直接通过下标改写字符串字面值中的元素，gcc 会报错，那么这样行不行呢：

```
char *p = "abcd";
p[1] = 'B';
```

这样可以绕过编译器的检查，在编译时不报错，但在运行时却出现段错误。在第 18.3 节我们看到，gcc 把这种字符串字面值分配在.rodata 段，在运行时.rodata 段加载到 Text Segment，操作系统会保护 Text Segment 不被改写，因此改写它会导致段错误。综上所述，**按照 C 标准，字符串字面值做右值时自动转换成 char *指针，但在 gcc 的实现中字符串字面值是只读的，因此字符串字面值做右值时应**

看作 const char *指针。

我们知道printf函数原型的第一个参数是const char *型，可以把char *或const char *指针传给它，所以下面这些调用都是合法的：

```
const char *p = "abcd";
const char str1[5] = "abcd";
char str2[5] = "abcd";
printf(p);
printf(str1);
printf(str2);
printf("abcd");
```

注意上面的 printf(p)和 printf(str2)调用在编译时会报警告 warning: format not a string literal and no format arguments，编译器认为用 printf(p)这种形式打印一个字符串是危险的，因为字符串中可能包含%号而被 printf 当成转换说明，printf 并不知道后面没有传其他参数，照样会从栈帧上取参数（第 23.6 节会详细解释怎样从栈帧上取可变参数），从而造成非法内存访问。比较保险的写法是 printf("%s", p)。

22.5 指针与结构体

首先定义一个结构体类型，然后定义这种类型的变量和指针：

```
struct unit {
        char c;
        int num;
};
struct unit u;
struct unit *p = &u;
```

要通过指针 p 访问结构体成员可以写成(*p).c 和(*p).num，为了书写方便，C 语言提供了->运算符，也可以写成 p->c 和 p->num。在第 7.1 节讲过，由.运算符组成的表达式能不能做左值取决于.运算符左边的操作数能不能做左值。->运算符则不同，由->运算符组成的表达式一定能做左值，想一想为什么。

22.6 指向指针的指针与指针数组

先前我们用指针指向 int 或 char 型，但指针也可以指向复合类型，比如指向另外一个指针类型的变量，这称为指向指针的指针。

```
int i;
int *pi = &i;
int **ppi = &pi;
```

这样定义之后，表达式*ppi 取 pi 的值，表达式**ppi 取 i 的值。请读者自己画图理解 i、pi、ppi 这三个变量之间的关系。

很自然地，也可以定义指向“指向指针的指针”的指针，但是很少用到：

```
int ***p;
```

数组中的每个元素也可以是指针类型。例如定义一个数组 a 由 10 个元素组成，每个元素都是 int *指针：

```
int *a[10];
```

这称为指针数组。int *a[10];和 int **pa;之间的关系类似于 int a[10];和 int *pa;之间的关系：a 是由一种元素组成的数组，pa 则是指向这种元素的指针。所以，如果 pa 指向 a 的首元素：

```
int *a[10];
int **pa = &a[0];
```

则 pa[0]和 a[0]取的是同一个元素，唯一比原来复杂的地方在于元素的类型是 int *指针，而不是基本类型。

我们知道 main 函数的标准原型应该是 int main(int argc, char *argv[]);。argc 是命令行参数（或者叫命令行选项）的个数。而 argv 是一个指向指针的指针，为什么不是指针数组呢？因为前面讲过，函数原型中的[]表示指针而不表示数组，等价于 char **argv。那为什么要写成 char *argv[]而不写成 char **argv 呢？这样写给读代码的人提供了有用的信息，argv 不是指向单个指针，而是指向一个指针数组的首元素。数组中每个元素都是 char *指针，指向一个命令行参数字符串。

例 22.2 打印命令行参数

```
#include <stdio.h>

int main(int argc, char *argv[])
{
        int i;
        for(i = 0; i < argc; i++)
                printf("argv[%d]=%s\n", i, argv[i]);
        return 0;
}
```

编译执行：

```
$ gcc main.c
$ ./a.out a b c
argv[0]=./a.out
argv[1]=a
argv[2]=b
argv[3]=c
$ ln -s a.out printargv
$ ./printargv d e
argv[0]=./printargv
argv[1]=d
argv[2]=e
```

注意程序名也算一个命令行参数，所以执行./a.out a b c 这个命令时，argc 是 4，argv 如图 22.4 所示。

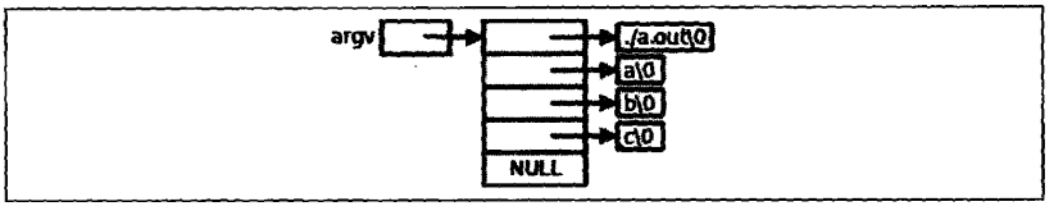


图 22.4 argv

由于 argv[4]是 NULL，我们也可以这样循环遍历 argv：

```
for (i=0; argv[i] != NULL; i++)
```

NULL 标识着 argv 的结尾，这个循环碰到 NULL 就结束，因而不会访问越界，这种用法很形象地称为 Sentinel，NULL 就像一个哨兵守卫着数组的边界。

在这个例子中我们还看到，如果给程序建立符号链接，然后通过符号链接运行这个程序，就可以得到不同的 argv[0]。通常，程序会根据不同的命令行参数做不同的事情，例如 ls -l 和 ls -R 打印不同格式的文件列表，有些程序会根据不同的 argv[0]做不同的事情，例如专门针对嵌入式系统的开源项目 Busybox，将各种 Linux 命令裁剪后集于一身，编译成一个可执行文件 busybox，安装时将 busybox 程序拷到嵌入式系统的/bin 目录下，同时在/bin、/sbin、/usr/bin、/usr/sbin 等目录下创建很多指向/bin/busybox 的符号链接，命名为 cp、ls、mv、ifconfig 等，不管执行哪个命令其实最终都在执行/bin/busybox，它会根据 argv[0]来扮演不同的命令。

最后补充介绍一下用 gdb 调试时如何加命令行参数。用 gdb 调试上面的代码：

```
$ gcc main.c -g
$ gdb a.out
GNU gdb (GDB) 7.1-ubuntu
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/
gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show
copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/akaedu/a.out...done.
(gdb) run a b c
Starting program: /home/akaedu/a.out a b c
argv[0]=/home/akaedu/a.out
argv[1]=a
argv[2]=b
argv[3]=c

Program exited normally.
(gdb) start d e f
Temporary breakpoint 1 at 0x80483ed: file main.c, line 6.
Starting program: /home/akaedu/a.out d e f

Temporary breakpoint 1, main (argc=4, argv=0xbffff424) at main.c:6
```

```
6               for(i = 0; i < argc; i++)
(gdb) c
Continuing.
argv[0]=/home/akaedu/a.out
argv[1]=d
argv[2]=e
argv[3]=f

Program exited normally.
(gdb) set args h i j
(gdb) r
Starting program: /home/akaedu/a.out h i j
argv[0]=/home/akaedu/a.out
argv[1]=h
argv[2]=i
argv[3]=j

Program exited normally.
```

可以在 run 或 start 命令后面加命令行参数，也可以用 set args 命令设置命令行参数之后再用 run 或 start 运行程序。

习题

1. 想想以下声明中的 const 分别起什么作用？编写程序验证你的猜测。

```
const char **p;
char *const *p;
char **const p;
```

2. 有这样一段代码：

```
void foo(const char **p) {}

int main(int argc, char **argv)
{
        foo(argv);
        return 0;
}
```

在调用 foo 函数时把 char **类型的实参 argv 传给 const char **类型的形参 p。这段代码在编译时会报警告：

```
$ gcc main.c
main.c: In function 'main':
main.c:5: warning: passing argument 1 of 'foo' from incompatible
pointer type
main.c:1: note: expected 'const char **' but argument is of type
'char **'
```

我们知道 char *指针可以赋给 const char *指针，为什么 char **指针不能赋给 const char **指针？怎样修改 foo 函数的接口才能使 foo(argv)这个调用合法？

3. 把例 8.4 改用指针数组实现，比较多维字符数组和字符串指针数组的区别。

22.7 指向数组的指针与多维数组

指针可以指向复合类型，上一节讲了指向指针的指针，这一节学习指向数组的指针。以下定义一个指向数组的指针，这种数组由10个int元素组成：

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

和上一节指针数组的定义int *a[10];相比，仅仅多了一个()括号。如何记住和区分这两种定义呢？我们可以认为[]括号比*号有更高的优先级（和运算符类似，后缀比前缀优先级高），如果a先和*号结合则表示a是一个指针，如果a先和[]括号结合则表示a是一个数组。int *a[10];这个定义可以拆成两句：

```
typedef int *t;
t a[10];
```

t代表int *类型，a则是由这种类型的元素组成的数组。int (*a)[10];这个定义也可以拆成两句：

```
typedef int t[10];
t *a;
```

t代表由10个int组成的数组类型，a则是指向这种类型的指针。

现在看指向数组的指针如何使用：

```
int a[10];
int (*pa)[10] = &a;
```

a是一个数组，在&a这个表达式中，数组类型做左值，取整个数组的首地址赋给指针pa。注意，&a[0]表示数组a的首元素的首地址，而&a表示数组a的首地址，虽然这两个地址的数值相同，但这两个表达式的类型是两种不同的指针类型，前者的类型是int *，而后者的类型是int (*)[10]。

*pa就表示pa所指向的数组a，*pa做右值时也会自动转换成指向数组首元素的指针，所以取数组的a[0]元素也可以用表达式(*pa)[0]。注意到*pa可以写成pa[0]，所以(*pa)[0]这个表达式可以改写成pa[0][0]，pa就像一个二维数组的名字。下面分析一下指向数组的指针和二维数组是什么关系。

int a[5][10];和int (*pa)[10];之间的关系同样类似于int a[10];和int *pa;之间的关系：a是由一种元素组成的数组，pa则是指向这种元素的指针。所以，如果pa指向a的首元素：

```
int a[5][10];
int (*pa)[10] = &a[0];
```

则pa[0]和a[0]取的是同一个元素，唯一比原来复杂的地方在于元素的类型是int [10]数组，而不是基本类型。这样，我们可以把pa当成二维数组名来使用，pa[1][2]和a[1][2]取的也是同一个元素，而且pa比a用起来更灵活，数组类型不支持赋值、

自增等运算，而指针可以支持，pa++使 pa 跳过二维数组的一行（40 个字节），指向 a[1]的首地址。

习题

1. 定义以下变量：

```
char a[4][3][2] = {{{'a', 'b'}, {'c', 'd'}, {'e', 'f'}},
                   {{'g', 'h'}, {'i', 'j'}, {'k', 'l'}},
                   {{'m', 'n'}, {'o', 'p'}, {'q', 'r'}},
                   {{'s', 't'}, {'u', 'v'}, {'w', 'x'}}};

char (*pa)[2] = &a[1][0];
char (*ppa)[3][2] = &a[1];
```

要想通过指针 pa 或 ppa 访问数组 a 中的'r'元素，表达式应该怎么写？

22.8 函数类型和函数指针类型

在 C 语言中，函数也是一种类型，可以定义指向函数的指针。我们知道，指针变量的内存单元里存放一个地址值，而函数指针的内存单元里存放的就是函数的入口地址（位于.text 段）。下面看一个简单的例子：

例 22.3 函数指针

```
#include <stdio.h>

void say_hello(const char *str)
{
        printf("Hello %s\n", str);
}

int main(void)
{
        void (*f)(const char *) = say_hello;
        f("Guys");
        return 0;
}
```

分析一下变量 f 的类型声明 void (*f)(const char *)，f 首先跟*号结合在一起，因此是一个指针。(*f)外面是一个函数原型的格式，参数是 const char *，返回值是 void，所以 f 是指向这种函数的指针。而 say_hello 函数的参数是 const char *，返回值是 void，正好是这种函数，因此 f 可以指向 say_hello 函数。注意，表达式 say_hello 是一种函数类型，函数类型和数组类型相似，做右值使用时自动转换成函数指针类型，因此可以直接赋给 f，也可以写成 void (*f)(const char *) = &say_hello;，把函数 say_hello 先取地址再赋给 f，就不需要自动类型转换了。

数组取下标运算符[]要求操作数是指针类型，a[1]等价于*(a+1)，如果 a 是数组类型则要自动转换成指向首元素的指针类型。同样道理，函数调用运算符()要求操

作数是函数指针类型，所以 f("Guys")是最直接的写法，而 say_hello("Guys")或(*f)("Guys")则是把函数类型自动转换成函数指针类型然后做函数调用。

下面再举几个例子区分函数类型和函数指针类型。首先定义函数类型 F：

```
typedef int F(void);
```

这种类型的函数不带参数，返回值是 int。那么可以这样声明 f 和 g：

```
F f, g;
```

相当于声明：

```
int f(void);
int g(void);
```

下面这个函数声明是错误的：

```
F h(void);
```

因为一个函数可以返回 void 类型、标量类型、结构体或联合体，但不能返回函数类型，也不能返回数组类型。而下面这个函数声明是正确的：

```
F *e(void);
```

函数 e 返回一个 F *类型的函数指针，函数指针是标量类型，可以做函数的返回值。如果给 e 多套几层括号仍然表示同样的意思：

```
F *((e))(void);
```

但如果把*号也套在括号里就不一样了：

```
int (*fp)(void);
```

这样声明的 fp 是一个函数指针而不是函数。也可以这样声明：

```
F *fp;
```

现在给这种函数指针起一个类型名叫 FP：

```
typedef int (*FP)(void);
```

假设有一个地址 0x12345678 正好是这样一个函数的首地址，我们把它强制转换成一个函数指针并调用它，以下两种写法皆可：

```
((FP)0x12345678)();
(*(FP)0x12345678)();
```

不用类型名 FP 也可以写出强制转换运算符：

```
((int (*)(void))0x12345678)();
```

```
(*(int (*)(void))0x12345678)();
```

下面这段程序（感谢 xinwu 网友提供）对于辨析函数类型和函数指针类型也非常有帮助：

```
#include <stdio.h>

void hello(void)
{}

int main(void)
{
        void (*hello_ptr)(void);

        printf("hello: %p\n", hello); /* function type as rvalue */
        printf("&hello: %p\n", &hello); /* function type as lvalue */
        printf("*hello: %p\n", *hello); /* function type as rvalue */

        hello_ptr = hello;
        printf("hello_ptr = hello;\n");

        printf("hello_ptr: %p\n", hello_ptr); /* value of a function
        pointer */
        printf("&hello_ptr: %p\n", &hello_ptr); /* address of a
        function pointer */
        printf("*hello_ptr: %p\n", *hello_ptr); /* dereference a f
        unction pointer */

        return 0;
}
```

通过函数指针调用函数和通过函数名直接调用函数相比有什么好处呢？我们研究一个例子。回顾第 7.3 节的习题 1，由于结构体中多了一个类型字段，需要重新实现 real_part、img_part、magnitude、angle 这些函数，你当时是怎么实现的？大概是这样吧：

```
double real_part(struct complex_struct z)
{
        if (z.t == RECTANGULAR)
                return z.a;
        else
                return z.a * cos(z.b);
}
```

现在类型字段有两种取值，RECTANGULAR 和 POLAR，每个函数都要用 if ... else ...分别处理两种情况。如果类型字段有三种取值呢？每个函数都要有 if ... else if ... else ...，或者 switch ... case ....。这样维护代码是不够理想的，现在我用函数指针给出一种实现：

```
double rect_real_part(struct complex_struct z)
{
        return z.a;
}

double rect_img_part(struct complex_struct z)
```

```
{
        return z.b;
}

double rect_magnitude(struct complex_struct z)
{
        return sqrt(z.a * z.a + z.b * z.b);
}

double rect_angle(struct complex_struct z)
{
        double PI = acos(-1.0);

        if (z.a > 0)
                return atan(z.b / z.a);
        else
                return atan(z.b / z.a) + PI;
}

double pol_real_part(struct complex_struct z)
{
        return z.a * cos(z.b);
}

double pol_img_part(struct complex_struct z)
{
        return z.a * sin(z.b);
}

double pol_magnitude(struct complex_struct z)
{
        return z.a;
}

double pol_angle(struct complex_struct z)
{
        return z.b;
}

double (*real_part_tbl[])(struct complex_struct) = { rect_real_
part, pol_real_part };
double (*img_part_tbl[])(struct complex_struct) = { rect_img_part,
pol_img_part };
double      (*magnitude_tbl[])(struct      complex_struct)      =
{ rect_magnitude, pol_magnitude };
double  (*angle_tbl[])(struct  complex_struct)  =  {  rect_angle,
pol_angle };

#define real_part(z) real_part_tbl[z.t](z)
#define img_part(z) img_part_tbl[z.t](z)
#define magnitude(z) magnitude_tbl[z.t](z)
#define angle(z) angle_tbl[z.t](z)
```

当调用 real_part(z)时，用类型字段 z.t 做索引，从指针数组 real_part_tbl 中取出相应的函数指针来调用，也可以达到 if ... else ...的效果。相比之下这种实现更好，每个函数都只做一件事情，而不必用 if ... else ...兼顾好几件事情，比如 rect_real_part 和 pol_real_part 各做各的，互相独立，而不必把它们的代码都耦合到一个函数中。“低耦合，高内聚”（Low Coupling, High Cohesion）是程序设计

的一条基本原则，这样可以更好地复用现有代码，更容易维护。如果类型字段 z.t 又多了一种取值，只需要添加一组新的函数，修改函数指针数组，原有的函数仍然可以不加改动地复用。

另外，这种实现用数据（函数指针数组）取代了代码（if ... else ...），和例 8.4 有异曲同工之处，也是应用了 Data-driven Programming 的思想。

22.9　不完全类型和复杂声明

现在 C 语言的语法基本讲完了，我们完整地总结一下 C 语言的各种类型。图 22.5 出自参考文献[4]。

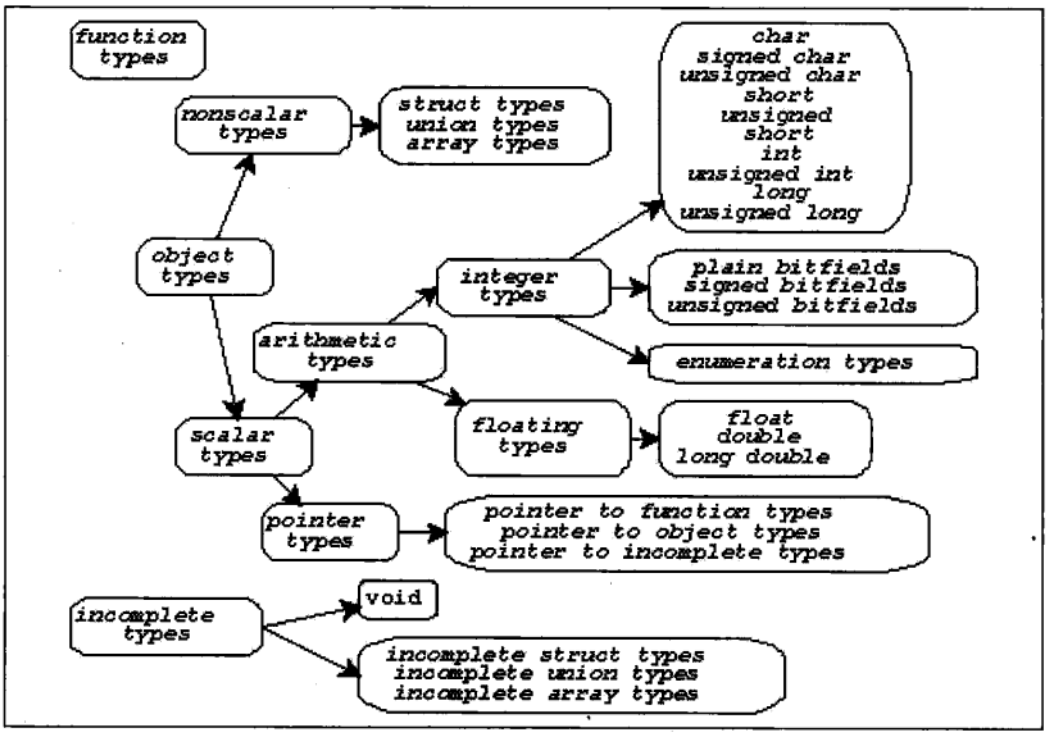


图 22.5　C 语言类型总结

C 语言的类型分为函数类型、对象类型和不完全类型三大类。对象类型又分为标量类型和非标量类型。不完全类型是暂时没有完全定义好的类型，编译器不知道这种类型该占几个字节的存储空间，例如：

```
struct s;
union u;
char str[];
```

在同一个编译单元里，具有不完全类型的变量可以通过多次声明组合成一个完全类型，比如将数组 str 先后声明两次：

```
extern char str[];
extern char str[10];
```

这两次声明可能来自两个不同的头文件，这两个头文件都被包含到同一个编译单元里。当编译器碰到第一个声明时，认为 str 的类型是不完全类型，碰到第二个声明时就组合成完全类型了，如果编译器处理到编译单元的末尾仍然无法将 str 的类型组合成一个完全类型就会报错。

再说说不完全的结构体类型，看一个例子：

```
struct s {
        struct t *pt;
};

struct t {
        struct s *ps;
};
```

struct s 和 struct t 各有一个指针成员指向另一种类型。编译器从前到后依次处理，当看到 struct s { struct t* pt; };时，认为 struct t 是一个不完全类型，pt 是一个指向不完全类型的指针，尽管如此，这个指针却是完全类型，因为不管什么指针都占 4 个字节存储空间，这一点很明确。然后编译器又看到 struct t { struct s *ps; };，这时 struct t 有了完整的定义，就组合成一个完全类型了，pt 的类型就组合成一个指向完全类型的指针。由于 struct s 在前面有完整的定义，所以 struct s *ps;也定义了一个指向完全类型的指针。

这样的类型定义是错误的：

```
struct s {
        struct t ot;
};

struct t {
        struct s os;
};
```

编译器看到 struct s { struct t ot; };时，认为 struct t 是一个不完全类型，无法定义成员 ot，因为不知道它该占几个字节。所以结构体中可以递归地定义指针成员，但不能递归地定义变量成员，你可以设想一下，假如允许递归地定义变量成员，struct s 中有一个 struct t，struct t 中又有一个 struct s，struct s 又中有一个 struct t，这就成了一个无穷递归的定义。

以上是两个结构体相互引用构成的递归定义，一个结构体引用自身也可以构成递归定义：

```
struct s {
        char data[6];
        struct s* next;
};
```

当编译器处理到第一行 struct s {时，认为 struct s 是一个不完全类型，当处理到第三行 struct s *next;时，认为 next 是一个指向不完全类型的指针，当处理到第四行};

时，struct s 成了一个完全类型，next 也成了一个指向完全类型的指针。类似这样的结构体是很多种数据结构的基本组成单元，如链表、二叉树等，我们将在第 25 章详细介绍。图 22.6 示意了由几个 struct s 结构体组成的链表，这些结构体称为链表的节点（Node）。

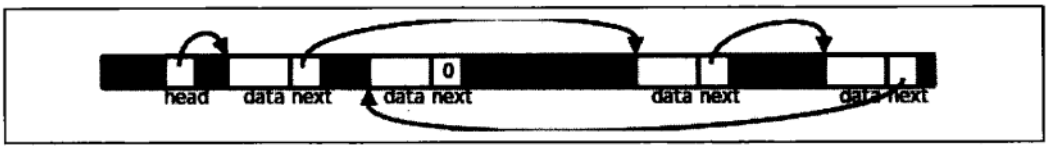


图 22.6 链表

head 指针是链表的头指针，指向第一个节点，每个节点的 next 指针域指向下一个节点，最后一个节点的 next 指针域是空指针，在图中用 0 表示。

可以想象得到，如果把指针和数组、函数、结构体层层组合起来可以构成非常复杂的类型，下面看几个复杂声明。

```
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

这个声明来自 signal(2)。sighandler_t 是一个函数指针，它所指向的函数带一个 int 型参数，返回值是 void。signal 是一个函数，它带两个参数，一个 int 参数，一个 sighandler_t 类型的参数，返回值也是 sighandler_t 类型。如果把这两行合成一行写，就是：

```
void (*signal(int signum, void (*handler)(int)))(int);
```

在分析复杂声明时，要借助 typedef 把复杂声明分解成几种基本形式：

- T *p;，p 是指向 T 类型的指针。
- T a[];，a 是由 T 类型的元素组成的数组。但有一个例外，在第 22.3 节讲过，如果 a 是函数的形参，则相当于 T *a;。
- T1 f(T2, T3...);，f 是一个函数，参数类型是 T2、T3 等，返回值类型是 T1。

我们分解一下这个复杂声明：

```
int (*(*fp)(void *))[10];
```

1. fp 和*号括在一起，说明 fp 是一个指针，指向 T1 类型：

```
typedef int (*T1(void *))[10];
T1 *fp;
```

2. T1 应该是一个函数类型，参数是 void *，返回值是 T2 类型：

```
typedef int (*T2)[10];
typedef T2 T1(void *);
T1 *fp;
```

3．T2和*号括在一起，应该也是个指针，指向T3类型：

```
typedef int T3[10];
typedef T3 *T2;
typedef T2 T1(void *);
T1 *fp;
```

显然，T3是一个int数组，由10个元素组成。分解完毕。参考文献[3]的5.12节有一个非常经典的例子，用递归下降法解析复杂声明，把C语言声明翻译成自然语言描述，建议读者去学习一下。

习题

1．分析以下复杂声明：

```
char (*(*x(void))[3])(void);
```

第 23 章

函数接口

我们在第 11.6 节讲过，函数的调用者和实现者之间订立了一个契约，在调用函数之前，调用者要为实现者提供某些条件，在函数返回时，实现者要对调用者尽到某些义务。如何描述这个契约呢？首先靠函数接口来描述，即函数名，参数，返回值，只要函数和参数的名字起得合理，参数和返回值的类型定得准确，这个函数该怎么用程序员单看函数接口就能猜出八九分了。然而函数接口并不能表达函数的全部语义，这时文档就起了重要的补充作用，函数的文档该写什么，怎么写，Man Page 为我们做了很好的榜样。

函数接口一旦和指针结合起来就变得异常灵活，有五花八门的用法，但是万变不离其宗，只要像图 22.1 那样画图分析，指针的任何用法都能分析清楚，所以，如果上一章你真正学明白了，本章不用学也能自己领悟出来。

23.1 本章的预备知识

这一节介绍本章的范例代码要用的几个 C 标准库函数。我们先体会一下这几个函数的接口是怎么设计的，Man Page 是怎么写的。其他常用的 C 标准库函数将在下一章介绍。

23.1.1 strcpy 与 strncpy

从现在开始我们的程序中要用到很多库函数，在学习每个库函数时一定要看 Man Page。Man Page 随时都在我们手边，想查什么只要敲一个命令就能查，然而很多初学者就是不喜欢看 Man Page，宁可满世界去查书、查资料，也不愿意看 Man Page。据我分析原因有三：

1．英文不好。那还是先学好了英文再学编程吧，否则即使你把这本书都学透了也一样无法胜任开发工作，因为你没有进一步学习的能力。

2．Man Page 的语言不够友好。Man Page 不像本书这样由浅入深地讲解，而是平铺直叙，不过看习惯了就好了，每个 Man Page 都不长，多看几遍自然可以抓住重点，理清头绪。本节分析一个例子，帮助读者把握 Man Page 的语言特点。

3．Man Page 通常没有例子。描述一个函数怎么用，一靠接口，二靠文档，而不

是靠例子。函数的用法无非是本章所总结的几种模式，只要把本章学透了，你就不需要每个函数都得有个例子教你怎么用了。

总之，Man Page是一定要看的，一开始看不懂硬着头皮也要看，为了逼迫读者去看Man Page而不是查书，本书不会像参考文献[3]那样把库函数总结成一个附录放在书后面。现在我们来分析strcpy(3)。

```
STRCPY(3)        Linux Programmer's Manual          STRCPY(3)

NAME
       strcpy, strncpy - copy a string

SYNOPSIS
#include <string.h>

char *strcpy(char *dest, const char *src);

char *strncpy(char *dest, const char *src, size_t n);
```

这个Man Page描述了两个函数，strcpy和strncpy，敲命令man strcpy或者man strncpy都可以看到这个Man Page。这两个函数的作用是把一个字符串拷贝给另一个字符串。**SYNOPSIS**部分给出了这两个函数的原型，以及要用这些函数需要包含哪些头文件。参数dest、src和n都加了下划线，有时候并不想从头到尾阅读整个Man Page，而是想查一下某个参数的含义，通过下划线和参数名就能很快找到你关心的部分。

dest表示Destination，src表示Source，看名字就能猜到是把src所指向的字符串拷贝到dest所指向的内存空间。这一点从两个参数的类型也能看出来，dest是char *型的，而src是const char *型的，说明src所指向的内存空间在函数中只能读不能改写，而dest所指向的内存空间在函数中是要改写的，改写的目的是当函数返回后调用者可以读取改写的结果。因此可以猜到strcpy函数是这样用的：

```
char buf[10];
strcpy(buf, "hello");
printf(buf);
```

至于strncpy的参数n是干什么用的，单从函数接口猜不出来，就需要看下面的文档。

```
DESCRIPTION
    The  strcpy()  function  copies the string pointed to by src,
including
    the terminating null byte ('\0'), to the buffer  pointed  to  by  dest.
    The  strings  may  not overlap, and the destination string dest must be
    large enough to receive the copy.

    The strncpy() function is similar, except that at most n bytes  of  src
    are  copied.  Warning: If there is no null byte among the first n
bytes
    of src, the string placed in dest will not be null terminated.

    If the length of src is less than n, strncpy() pads  the  remainder  of
```

```
dest with null bytes.

A simple implementation of strncpy() might be:

    char*
    strncpy(char *dest, const char *src, size_t n){
        size_t i;

        for (i = 0 ; i < n && src[i] != '\0' ; i++)
            dest[i] = src[i];
        for ( ; i < n ; i++)
            dest[i] = '\0';

        return dest;
    }
```

在文档中强调了 strcpy 在拷贝字符串时会把结尾的'\0'也拷到 dest 中，因此保证了 dest 是以 Null 结尾的字符串。但这样做存在一个问题，strcpy 只知道 src 字符串的首地址，不知道长度，它会一直拷贝到'\0'为止，因此 dest 所指向的内存空间要足够大，否则有可能写越界，例如：

```
char buf[10];
strcpy(buf, "hello world");
```

如果不保证 src 所指向的内存空间以'\0'结尾，也有可能读越界，例如：

```
char buf[10] = "abcdefghij", str[4] = "hell";
strcpy(buf, str);
```

因为 strcpy 函数的实现者通过函数接口无法得知 src 字符串的长度和 dest 内存空间的大小，所以“确保不会写越界”应该是调用者的责任，调用者提供的 dest 参数应该指向足够大的内存空间，“确保不会读越界”也是调用者的责任，调用者提供的 src 参数指向的内存空间应该确保以'\0'结尾。

此外，文档中还强调了 src 和 dest 所指向的内存空间不能有重叠，例如这样调用是不允许的：

```
char buf[10] = "hello";
strcpy(buf, buf+1);
```

凡是有指针参数的 C 标准库函数基本上都有这条要求，也有个别函数例外，下一章会讲到这样的函数。strncpy 的参数 n 指定最多从 src 中拷贝 n 个字节到 dest 中，换句话说，如果拷贝到'\0'就结束，如果拷贝到 n 个字节还没有碰到'\0'，那么也结束。如果调用者不能确定 src 字符串的长度，可以规定一个适当的 n 值，以确保读写不会越界，通常让 n 的值等于 dest 所指向的内存空间的大小：

```
char buf[10];
strncpy(buf, "hello world", sizeof(buf));
```

这样做也存在一个问题，文档中特别用 **Warning** 指出，这意味着 dest 有可能不是以'\0'结尾的。例如上面的调用，虽然把"hello world"截断为 10 个字符拷贝到 buf

中，但 buf 不是以'\0'结尾的，如果再 printf(buf)就会读越界。如果你需要确保 dest 以'\0'结尾，可以这么调用：

```
char buf[10];
strncpy(buf, "hello world", sizeof(buf));
buf[sizeof(buf)-1] = '\0';
```

strncpy 还有一个特性，如果 src 字符串全部拷完了不足 n 个字节，那么还差多少个字节就补多少个'\0'，但是正如上面所述，这并不保证 dest 一定以'\0'结尾，当 src 字符串的长度大于 n 时，不但不补多余的'\0'，连字符串结尾的'\0'也没有。strcpy(3)的文档已经相当友好了，为了帮助理解，还给出一个 strncpy 的简单实现。

```
RETURN VALUE
       The  strcpy()  and strncpy() functions return a pointer
       to the destination string dest.
CONFORMING TO
       SVr4, 4.3BSD, C89, C99.

NOTES
       Some programmers consider strncpy() to be inefficient
       and error  prone.
       If  the  programmer knows (i.e., includes code to test!)
       that the size
       of dest is greater than the length of src, then strcpy()
       can be used.

       If there is no terminating null byte in the first n
       characters of  src,
   strncpy()   produces   an  unterminated  string  in  dest.
Programmers often
       prevent this mistake by forcing termination as follows:

           strncpy(buf, str, n);
           if (n > 0)
               buf[n - 1]= '\0';
```

函数的 Man Page 都会有一个 **RETURN VALUE** 部分专门讲返回值，这两个函数的返回值都是 dest 指针。可是为什么要返回 dest 指针呢？dest 指针本来就是调用者传过去的，再返回一遍 dest 指针并没有提供任何有用的信息。之所以这么规定是为了把函数调用当做一个指针类型的表达式使用，比如 printf("%s\n", strcpy(buf, "hello"))，一举两得，如果 strcpy 的返回值是 void 就没有这么方便了。

CONFORMING TO 部分描述了这个函数是遵照哪些标准实现的。strcpy 和 strncpy 是 C 标准库函数，当然遵照 C99 标准。在第 18.2 节讲过 libc 中的有些函数属于 POSIX 标准但并不属于 C 标准，例如_exit(2)。

NOTES 部分给出一些提示信息。这里指出如何确保 strncpy 的 dest 以'\0'结尾，和我们上面给出的代码类似，但这段代码比较谨慎，由于 n 是个变量，在执行 buf[n - 1]= '\0';之前先检查一下 n 是否大于 0，如果 n 不大于 0，buf[n - 1]就访问越界了，

所以要避免。

```
    BUGS
          If the destination string of a strcpy() is not large enough,
          then  any-
          thing  might  happen.  Overflowing  fixed-length  string
          buffers is a
          favorite cracker technique for taking complete control of the
          machine.
          Any  time  a  program  reads  or copies data into a buffer, the
          program
          first needs to check that there's enough space.  This may  be
          unneces-
          sary  if you can show that overflow is impossible, but be
          careful: pro-
          grams can get changed over time, in ways that may make  the
          impossible possible.

    SEE ALSO
    bcopy(3),   memccpy(3),   memcpy(3),   memmove(3),   strdup(3),
strpcpy(3),
    wcscpy(3), wcsncpy(3)

    COLOPHON
          This page is part of release 3.23 of the Linux  man-pages
          project.  A
          description  of  the project, and information about reporting
          bugs, can
          be found at http://www.kernel.org/doc/man-pages/.

    GNU                        2009-06-01                      STRCPY(3)
```

BUGS 部分说明了使用这些函数可能引起的 Bug，这部分一定要仔细看。用 strcpy 比用 strncpy 更加不安全，如果在调用 strcpy 之前不仔细检查 src 字符串的长度就有可能写越界，这是一个很常见的错误，例如：

```
    void foo(char *str)
    {
            char buf[10];
            strcpy(buf, str);
            ...
    }
```

str 所指向的字符串有可能超过 10 个字符而导致写越界，在第 10.4 节我们看到过，这种写越界有可能当时不出错，而在函数返回时出现段错误，原因是写越界覆盖了保存在栈帧上的返回地址，函数返回时跳转到非法地址，因而出错。像 buf 这种由调用者分配并传给 strcpy 函数访问的一段内存通常称为缓冲区（Buffer），缓冲区写越界的错误称为缓冲区溢出（Buffer Overflow）。如果只是出现段错误那还不算严重，更严重的是缓冲区溢出 Bug 经常被恶意用户利用，使函数返回时跳转到一个事先设计好的地址，执行一段事先设计好的指令，如果设计得巧妙甚至可以启动一个 Shell，然后随心所欲执行任何命令，可想而知，如果一个用 root 权限执行的程序存在这样的 Bug，被攻陷了，后果将很严重。至于怎样巧妙设计和攻陷一个存在缓冲区溢出问题的程序，感兴趣的读者可以查阅参考

文献[28]。

习题

1. 自己实现一个 strcpy 函数，尽可能简洁，按照本书的编码风格你能用三行代码写出函数体吗？

2. 编一个函数，输入一个字符串，要求做一个新字符串，把其中所有的一个或多个连续空白字符都压缩成一个空格。这里所说的空白字符包括空格、'\t'、'\n'、'\r'。例如原来的字符串是：

```
This Content hoho      is ok
      ok?

      file system
uttered words   ok ok     ?
end.
```

压缩空白之后就是：

```
This Content hoho is ok ok? file system uttered words ok ok ? end.
```

实现该功能的函数接口要求符合下述规范：

```
char *shrink_space(char *dest, const char *src, size_t n);
```

各项参数和返回值的含义和 strncpy 类似。完成之后，为自己实现的函数写一个 Man Page。

23.1.2 malloc 与 free

程序中需要动态分配一块内存时怎么办呢？可以像上一节那样定义一个缓冲区数组。这种方法不够灵活，C89 要求定义的数组是固定长度的，而程序往往在运行时才知道要动态分配多大的内存，例如：

```
void foo(char *str, int n)
{
        char buf[?];
        strncpy(buf, str, n);
        ...
}
```

n 是由参数传进来的，事先不知道是多少，那么 buf 该定义多大呢？在第 8.1 节讲过 C99 引入 VLA 特性，可以定义 char buf[n+1] = {0};，这样可以确保 buf 是以'\0'结尾的。但即使用 VLA 仍然不够灵活，VLA 是在栈上分配的，函数返回时就要释放，如果我们希望动态分配一块全局的内存空间，在各函数中都可以访问，该怎么办呢？由于全局数组无法定义成 VLA，所以仍然不能满足要求。

在第 19.5 节提过，每个进程都有一个堆空间，C 标准库函数 malloc 可以在堆空间动态分配内存，它的底层通过 brk 系统调用向操作系统申请内存。动态分配的内存用完之后可以用 free 释放，更准确地说是归还给了 malloc，下次调用 malloc 时这块内存可以再次分配出来。本节学习这两个函数的用法和工作原理。

```
#include <stdlib.h>

void *malloc(size_t size);
返回值：成功返回所分配内存空间的首地址，出错返回 NULL

void free(void *ptr);
```

malloc 的参数 size 表示要分配的字节数，如果分配失败（可能是因为系统内存耗尽）则返回 NULL。由于 malloc 函数不知道用户拿到这块内存要存放什么类型的数据，所以返回通用指针 void *，用户程序可以转换成其他类型的指针再访问这块内存。malloc 函数保证它返回的指针所指向的地址满足系统的对齐要求，例如在 32 位平台上返回的指针一定对齐到 4 字节边界，以保证用户程序把它转换成任何类型的指针都能用。

动态分配的内存用完之后可以调 free 函数释放掉，传给 free 的参数正是先前 malloc 返回的内存块首地址。举例如下：

例 23.1　malloc 和 free

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct {
        int number;
        char *msg;
} unit_t;

int main(void)
{
        unit_t *p = malloc(sizeof(unit_t));

        if (p == NULL) {
                printf("out of memory\n");
                exit(1);
        }
        p->number = 3;
        p->msg = malloc(20);
        strcpy(p->msg, "Hello world!");
        printf("number: %d\nmsg: %s\n", p->number, p->msg);
        free(p->msg);
        free(p);
        p = NULL;

        return 0;
}
```

这个程序要注意以下几点：

- unit_t *p = malloc(sizeof(unit_t));这一句，等号右边是 void *类型，等号左边是 unit_t *类型，编译器会做隐式类型转换，我们讲过 void *类型和任何指针类型之间可以相互隐式转换。
- 虽然内存耗尽是很不常见的错误，但写程序要规范，malloc 之后应该判断是否成功。以后要学习的大部分系统函数都有成功的返回值和失败的返回值，每次调用系统函数之后都应该判断是否成功。
- free(p);之后，p 所指的内存空间是归还了，但是 p 的值并没有变，因为从 free 的函数接口来看根本没法改变 p 的值，p 现在指向的内存空间已经不属于用户程序，换句话说，p 成了野指针，为避免出现野指针，我们应该在 free(p);之后手动置 p = NULL;。
- 应该先 free(p->msg)，再 free(p)，顺序不能颠倒。如果先 free(p)，p 成了野指针，就不能再访问 p->msg 了。

上面的例子只有一个简单的顺序控制流程，分配内存，赋值，打印，释放内存，退出程序。这种情况下即使不用 free 释放内存也可以，因为进程退出时它占用的所有内存都会释放，也就是归还给了操作系统。但如果一个程序长年累月运行（例如网络服务器程序），并且在循环或递归中调用 malloc 分配内存，则必须有 free 与之配对，分配一次就要释放一次，否则每次循环都分配内存，分配完了又不释放，就会慢慢耗尽系统内存，这种错误称为内存泄漏（Memory Leak）。另外，malloc 返回的指针一定要保存好，只有把它传给 free 才能释放这块内存，如果这个指针丢失了，就没有办法 free 这块内存了，也会造成内存泄漏。例如：

```
void foo(void)
{
        char *p = malloc(10);
        ...
}
```

foo 函数返回时要释放局部变量 p 的内存空间，它所指向的内存地址就丢失了，这 10 个字节也就没法释放了。内存泄漏的 Bug 很难找到，因为它并不像访问越界一样导致程序运行错误，少量内存泄漏并不影响程序的正确运行，大量的内存泄漏会导致物理内存紧缺，换页频繁，不仅影响当前进程，而且会把整个系统都拖得很慢。

关于 malloc 和 free 还有一些特殊情况。malloc(0)这种调用也是合法的，也会返回一个非 NULL 的指针，这个指针也可以传给 free 释放，但是不能通过这个指针访问内存。free(NULL)也是合法的，不做任何事情，但是 free 一个野指针是不合法的，例如先调用 malloc 返回一个指针 p，然后连着调用两次 free(p);，则后一次调用会产生运行时错误。

参考文献[3]的 8.7 节给出了 malloc 和 free 的简单实现，基于环形链表。目前读者还没有学习链表，看那段代码会有点困难，我再做一些简化，如图 23.1 所示，目的是让读者理解 malloc 和 free 的工作原理。libc 的实现比这要复杂得多，但基本工作原理也是如此。读者只要理解了基本工作原理，就很容易分析在使用 malloc

和 free 时遇到的各种 Bug。

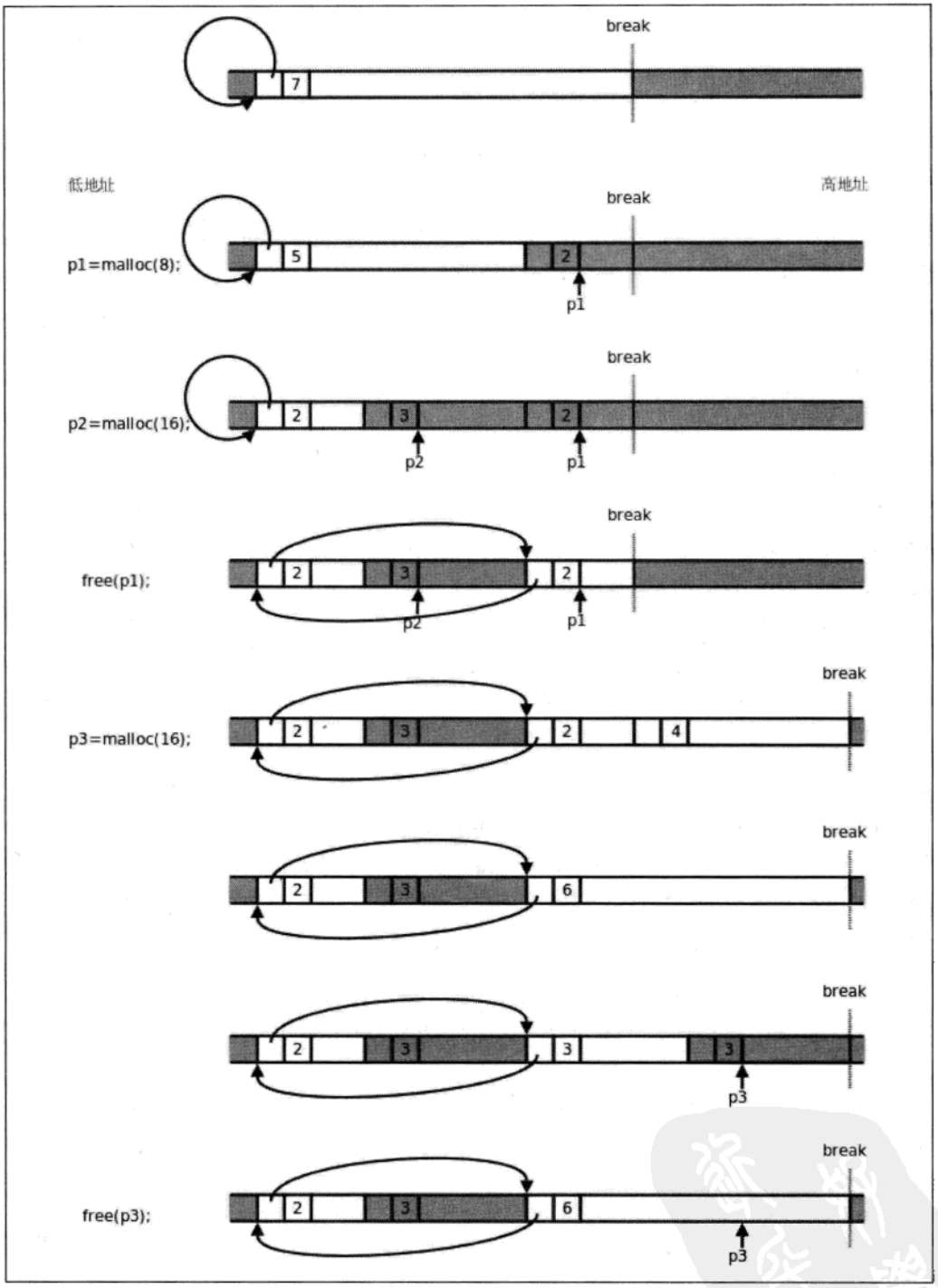


图 23.1　简单的 malloc 和 free 实现

图 23.1 中白色背景的框表示 malloc 管理的空闲内存块，深色背景框不归 malloc 管，可能是已经分配给用户程序的内存块，也可能还没有映射到当前进程的地址空间，我们讲过 Break 以上不属于当前进程的地址空间，需要通过 brk 系统调用

向内核申请，由内核分配物理内存映射到进程地址空间，并抬高 Break。每个内存块开头都有一个头节点，里面有一个指针字段和一个长度字段，指针字段把所有空闲块的头节点串在一起，组成一个环形链表，长度字段记录着头节点和后面的内存块加起来一共有多长，以 8 字节为单位（也就是以头节点的长度为单位）。

1. 一开始堆空间由一个空闲块组成，长度为 7×8=56 字节，除头节点之外的长度为 48 字节。

2. 调用 malloc 分配 8 个字节，要在这个空闲块的末尾截出 16 个字节，其中新的头节点占了 8 个字节；另外 8 个字节返回给用户使用，注意返回的指针 p1 指向头节点后面的内存块首地址。

3. 再次调用 malloc 分配 16 个字节，又在空闲块的末尾截出 24 个字节，步骤和上一步类似。

4. 调用 free 释放 p1 所指向的内存块，内存块（包括头节点在内）归还给了 malloc，现在 malloc 管理着两块不连续的内存，用环形链表串起来。注意这时 p1 成了野指针，指向不属于用户程序的内存空间。由于 p1 所指向的内存地址在 Break 之下，是属于当前进程的，所以访问 p1 时不会出段错误，但在访问 p1 时这段内存可能已经被 malloc 再次分配出去了，可能会读到不确定的值，或者意外改写数据。另外注意，此时如果通过 p2 向右写越界，也不会出段错误，但是会覆盖右边的头节点，从而破坏 malloc 管理的环形链表，malloc 就无法从一个空闲块的指针字段找到下一个空闲块了，找到哪去都不一定，全乱套了。

5. 调用 malloc 分配 16 个字节，现在虽然有两个空闲块，各有 8 个字节可分配，但这两块是不连续的，malloc 只好通过 brk 系统调用抬高 Break，获得新的内存空间。在参考文献[3]的实现中每次调用 sbrk 函数向内核申请 8KB 内存，Linux 系统的 sbrk 函数也是通过 brk 系统调用实现的，这里为了画图方便，我们假设每次调用 sbrk 申请 32 个字节，建立一个新的空闲块。

6. 新申请的空闲块和前一个空闲块连续，可以合并成一个。在能合并时要尽量合并，以免空闲块越割越小，无法满足大的内存分配请求。

7. 在合并后的这个空闲块末尾截出 24 个字节，新的头节点占 8 个字节，另外 16 个字节返回给用户。

8. 调用 free(p3)释放这个内存块，由于它和前一个空闲块连续，又重新合并成一个空闲块。注意，Break 只能抬高而不能降低，从内核申请到的内存以后都归 malloc 管了，即使调用 free 也不会还给内核。

习题

1. 编写一个小程序不停地调 malloc，让它耗尽系统内存。观察一下，分配了多少内存之后才会出现分配失败？内存耗尽之后会怎么样？会不会死机？

23.2　传入参数与传出参数

如果函数接口有指针参数，既可以把指针所指向的数据传给函数使用（称为传入参数，如表 23.1 所示），也可以由函数填充指针所指的内存空间，传回给调用者使用（称为传出参数，如表 23.2 所示），例如 strcpy 的 src 参数是传入参数，dest 参数是传出参数。有些函数的指针参数同时充当这两种角色，如 select(2)的 fd_set *参数，既是传入参数又是传出参数，这称为 Value-result 参数，如表 23.3 所示。

表 23.1　传入参数示例：void func(const unit_t *p);

调用者	实现者
分配 p 所指的内存空间 在 p 所指的内存空间中保存数据 调用函数 由于有 const 限定符，调用者可以确信 p 所指的内存空间不会被改变	规定指针参数的类型 unit_t * 读取 p 所指的内存空间

想一想，如果有函数接口 void func(const int p);，这里的 const 写或不写对调用者来说有区别吗？

表 23.2　传出参数示例：void func(unit_t *p);

调用者	实现者
分配 p 所指的内存空间 调用函数 读取 p 所指的内存空间	规定指针参数的类型 unit_t * 在 p 所指的内存空间中保存数据

表 23.3　Value-result 参数示例：void func(unit_t *p);

调用者	实现者
分配 p 所指的内存空间 在 p 所指的内存空间中保存数据 调用函数 读取 p 所指的内存空间	规定指针参数的类型 unit_t * 读取 p 所指的内存空间 改写 p 所指的内存空间

由于传出参数和 Value-result 参数的函数接口完全相同，应该在文档中说明是哪种参数。以下是一个传出参数的完整例子：

例 23.2　传出参数

```
/* populator.h */
#ifndef POPULATOR_H
#define POPULATOR_H

typedef struct {
        int number;
        char msg[20];
} unit_t;
```

```
void set_unit(unit_t *);

#endif
/* populator.c */
#include <string.h>
#include "populator.h"

void set_unit(unit_t *p)
{
        if (p == NULL)
                return; /* ignore NULL parameter */
        p->number = 3;
        strcpy(p->msg, "Hello World!");
}
/* main.c */
#include <stdio.h>
#include "populator.h"

int main(void)
{
        unit_t u;

        set_unit(&u);
        printf("number: %d\nmsg: %s\n", u.number, u.msg);
        return 0;
}
```

本章的例子都是由多个源文件编译链接成一个程序，请读者自己练习写Makefile。很多系统函数对于指针参数是NULL的情况有特殊规定：如果传入参数是NULL表示取缺省值，例如pthread_create(3)的pthread_attr_t *参数，也可能表示不做特别处理，例如free(3)的参数；如果传出参数是NULL表示调用者不需要传出值，例如time(2)的参数。这些特殊规定应该在文档中写清楚。

23.3 两层指针的参数

两层指针也是指针，同样可以表示传入参数、传出参数或者Value-result参数，只不过该参数所指的内存空间应该解释成指针变量。用两层指针做传出参数的系统函数也很常见，比如pthread_join(3)的void **参数。下面看一个简单的例子。

例23.3 两层指针做传出参数

```
/* redirect_ptr.h */
#ifndef REDIRECT_PTR_H
#define REDIRECT_PTR_H

void get_a_day(const char **);

#endif
```

想一想，这里的参数指针是const char **，有const限定符，却不是传入参数而是传出参数，为什么？如果是传入参数应该怎么表示？

```
/* redirect_ptr.c */
#include "redirect_ptr.h"

static const char *msg[] = {"Sunday", "Monday", "Tuesday", "Wednesday",
                            "Thursday", "Friday", "Saturday"};

void get_a_day(const char **pp)
{
        static int i = 0;
        *pp = msg[i%7];
        i++;
}
/* main.c */
#include <stdio.h>
#include "redirect_ptr.h"

int main(void)
{
        const char *firstday = NULL;
        const char *secondday = NULL;
        get_a_day(&firstday);
        get_a_day(&secondday);
        printf("%s\t%s\n", firstday, secondday);
        return 0;
}
```

两层指针作为传出参数还有一种特别的用法，可以在函数中分配内存，调用者通过传出参数取得指向该内存的指针，比如 getaddrinfo(3)的 struct addrinfo **参数。一般来说，实现一个分配内存的函数就要实现一个释放内存的函数，所以 getaddrinfo(3)有一个对应的 freeaddrinfo(3)函数。

表 23.4　通过参数分配内存示例：void alloc_unit(unit_t **pp); void free_unit(unit_t *p);

分配 pp 所指的指针变量的空间 调用 alloc_unit 分配内存 读取 pp 所指的指针变量，通过后者使用 alloc_unit 分配的内存 调用 free_unit 释放内存	规定指针参数的类型 unit_t ** alloc_unit 分配 unit_t 的内存并初始化，为 pp 所指的指针变量赋值 free_unit 释放在 alloc_unit 中分配的内存

例 23.4　通过两层指针参数分配内存

```
/* para_allocator.h */
#ifndef PARA_ALLOCATOR_H
#define PARA_ALLOCATOR_H

typedef struct {
        int number;
        char *msg;
} unit_t;

void alloc_unit(unit_t **);
void free_unit(unit_t *);

#endif
```

```
/* para_allocator.c */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "para_allocator.h"

void alloc_unit(unit_t **pp)
{
        unit_t *p = malloc(sizeof(unit_t));
        if (p == NULL) {
                printf("out of memory\n");
                exit(1);
        }
        p->number = 3;
        p->msg = malloc(20);
        strcpy(p->msg, "Hello World!");
        *pp = p;
}

void free_unit(unit_t *p)
{
        free(p->msg);
        free(p);
}
/* main.c */
#include <stdio.h>
#include "para_allocator.h"

int main(void)
{
        unit_t *p = NULL;

        alloc_unit(&p);
        printf("number: %d\nmsg: %s\n", p->number, p->msg);
        free_unit(p);
        p = NULL;
        return 0;
}
```

思考一下，为什么在 main 函数中不能直接调用 free(p)释放内存，而要调用 free_unit(p)？为什么一层指针的函数接口 void alloc_unit(unit_t *p);不能分配内存，而一定要用两层指针的函数接口？

总结一下，两层指针参数如果是传出的，可以有两种情况：第一种情况，传出的指针指向静态内存（比如上面的例子），或者指向已分配的动态内存（比如指向某个链表的节点）；第二种情况是在函数中动态分配内存，然后传出的指针指向这块内存空间，这种情况下调用者应该在使用内存之后调用释放内存的函数，调用者的责任是请求分配和释放内存，实现者的责任是完成分配和释放内存的操作。由于这两种情况的函数接口相同，应该在函数的文档中说明是哪一种情况。

23.4 返回值是指针的情况

返回值显然是传出的而不是传入的，如果返回值传出的是指针，和上一节通过参数传出指针类似，也分为两种情况：第一种是传出指向静态内存或已分配的动态

内存的指针，例如 localtime(3)和 inet_ntoa(3)，第二种是在函数中动态分配内存并传出指向这块内存的指针，例如 malloc(3)和 strdup(3)，这种情况通常还要实现一个释放内存的函数，所以有和 malloc(3)对应的 free(3)函数。由于这两种情况的函数接口相同，应该在函数的文档中说明是哪一种情况。

表 23.5　返回指向已分配内存的指针示例：unit_t *func(void);

调用者	实现者
调用函数	规定返回值指针的类型 unit_t *
将返回值保存下来以备后用	返回一个指针

以下是一个完整的例子。

例 23.5　返回指向已分配内存的指针

```
    /* ret_ptr.h */
    #ifndef RET_PTR_H
    #define RET_PTR_H

    char *get_a_day(int idx);

    #endif
    /* ret_ptr.c */
    #include <string.h>
    #include "ret_ptr.h"

    static const char *msg[] = {"Sunday", "Monday", "Tuesday",
"Wednesday",
                                "Thursday", "Friday", "Saturday"};

    char *get_a_day(int idx)
    {
            static char buf[20];
            strcpy(buf, msg[idx]);
            return buf;
    }
    /* main.c */
    #include <stdio.h>
    #include "ret_ptr.h"

    int main(void)
    {
            printf("%s %s\n", get_a_day(0), get_a_day(1));
            return 0;
    }
```

这个程序的运行结果是 Sunday Monday 吗？请读者自己分析一下。

表 23.6　动态分配内存并返回指针示例：unit_t *alloc_unit(void); void free_unit(unit_t *p);

调用者	实现者
调用 alloc_unit 分配内存	规定返回值指针的类型 unit_t *
将返回值保存下来以备后用	alloc_unit 分配内存并返回指向该内存的指针
调用 free_unit 释放内存	free_unit 释放由 alloc_unit 分配的内存

以下是一个完整的例子。

例 23.6 动态分配内存并返回指针

```
/* ret_allocator.h */
#ifndef RET_ALLOCATOR_H
#define RET_ALLOCATOR_H

typedef struct {
        int number;
        char *msg;
} unit_t;

unit_t *alloc_unit(void);
void free_unit(unit_t *);

#endif
/* ret_allocator.c */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "ret_allocator.h"

unit_t *alloc_unit(void)
{
        unit_t *p = malloc(sizeof(unit_t));
        if (p == NULL) {
                printf("out of memory\n");
                exit(1);
        }
        p->number = 3;
        p->msg = malloc(20);
        strcpy(p->msg, "Hello world!");
        return p;
}

void free_unit(unit_t *p)
{
        free(p->msg);
        free(p);
}
/* main.c */
#include <stdio.h>
#include "ret_allocator.h"

int main(void)
{
        unit_t *p = alloc_unit();

        printf("number: %d\nmsg: %s\n", p->number, p->msg);
        free_unit(p);
        p = NULL;
        return 0;
}
```

思考一下，通过参数分配内存需要两层的指针，而通过返回值分配内存就只需要返回一层的指针，为什么？

习题

1. 跟指针参数或指针返回值有关的函数接口模式肯定不止本章介绍的这几种。比如由调用者分配一段内存，通过指针参数传给实现者，这也算一种模式。请读者自己总结其他模式，并写出示例程序。

23.5　回调函数

如果参数是一个函数指针，调用者可以传递一个函数的地址给实现者，即调用者提供一个函数但自己不去调用它，而是让实现者去调用它，这称为回调函数（Callback Function）。例如 qsort(3)和 bsearch(3)。

表 23.7　回调函数示例：void func(void (*f)(void *), void *p);

调用者	实现者
提供一个回调函数，再提供一个准备传给回调函数的参数。 把回调函数传给参数 f，把准备传给回调函数的参数按 void *类型传给参数 p	在适当的时候根据调用者传来的函数指针 f 调用回调函数，将调用者传来的参数 p 转交给回调函数，即调用 f(p)

以下是一个简单的例子。实现了一个 repeat_three_times 函数，可以把调用者传来的任何回调函数重复执行三次。

例 23.7　回调函数

```
/* para_callback.h */
#ifndef PARA_CALLBACK_H
#define PARA_CALLBACK_H

typedef void (*callback_t)(void *);
void repeat_three_times(callback_t, void *);

#endif
/* para_callback.c */
#include "para_callback.h"

void repeat_three_times(callback_t f, void *para)
{
        f(para);
        f(para);
        f(para);
}
/* main.c */
#include <stdio.h>
#include "para_callback.h"

static void say_hello(void *str)
{
        printf("Hello %s\n", (const char *)str);
}
```

```
static void count_numbers(void *num)
{
        int i;
        for (i = 1; i <= (int)num; i++)
                printf("%d ", i);
        putchar('\n');
}

int main(void)
{
        repeat_three_times(say_hello, "Guys");
        repeat_three_times(count_numbers, (void *)4);
        return 0;
}
```

回顾一下前面几节的例子，参数类型都是由实现者规定的。而本例中回调函数的参数按什么类型解释由调用者规定，对于实现者来说就是一个void *指针，实现者只负责将这个指针转交给回调函数，而不关心它到底指向什么数据类型。调用者知道自己传的参数是char *型和int型的，那么在自己提供的回调函数中就应该知道参数要转换成char *型和int型来解释。

回调函数的一个典型应用就是实现类似C++的泛型算法（Generics Algorithm）。下面实现的max函数可以在任意一组对象中找出最大值，可以是一组int、一组char或一组结构体，但是实现者并不知道怎样去比较两个对象的大小，需要调用者再提供一个做比较操作的回调函数。

例23.8 泛型算法

```
/* generics.h */
#ifndef GENERICS_H
#define GENERICS_H

#include <stddef.h>

typedef int (*cmp_t)(const void *, const void *);
void *max(const void *base, size_t nmemb, size_t size, cmp_t cmp);

#endif
/* generics.c */
#include "generics.h"

void *max(const void *base, size_t nmemb, size_t size, cmp_t cmp)
{
        const char *_base = base;
        const char *temp = _base;
        size_t i;
        for (i = 1; i < nmemb; i++)
                if (cmp(temp, _base + size * i) < 0)
                        temp = _base + size * i;
        return (void *)temp;
}
/* main.c */
#include <stdio.h>
#include "generics.h"
```

```
typedef struct {
        const char *name;
        int score;
} student_t;

static int cmp_student(const void *a, const void *b)
{
        if (((student_t *)a)->score > ((student_t *)b)->score)
                return 1;
        else if (((student_t *)a)->score == ((student_t *)b)->
        score)
                return 0;
        else
                return -1;
}

int main(void)
{
        student_t list[4] = {{"Tom", 68}, {"Jerry", 72},
                             {"Moby", 60}, {"Kirby", 89}};
    student_t *pmax = max(list, sizeof(list)/sizeof(student_t),
                             sizeof(student_t), cmp_student);
    printf("%s gets the highest score %d\n", pmax->name, pmax-> score);
    return 0;
}
```

传给max函数的参数是对象数组的基地址、共有多少个对象以及每个对象的大小，max函数可以算出每个对象的首地址，但并不知道对象到底是什么类型，也不必关心它是什么类型，只要把每个对象的首地址转交给比较函数cmp，然后根据比较结果做相应操作即可，cmp是调用者提供的回调函数，调用者当然知道对象是什么类型以及如何比较。

以上举例的回调函数都是被同步调用的，调用者调用max函数，max函数则调用cmp函数，相当于调用者间接调用了自己提供的回调函数。除此之外，异步调用也是回调函数的一种典型用法，调用者首先将回调函数传给实现者，实现者记住这个函数，这称为**注册**一个回调函数，然后当某个事件发生时实现者再调用先前注册过的函数，比如sigaction(2)注册一个信号处理函数，当信号产生时由操作系统调用该函数进行处理，再比如pthread_create(3)注册一个线程函数，当发生调度时操作系统切换到新注册的线程函数中运行，在GUI编程中异步回调函数更是有普遍的应用，例如为某个按钮注册一个回调函数，当用户单击按钮时调用它。

以下是一个代码框架。

```
/* registry.h */
#ifndef REGISTRY_H
#define REGISTRY_H

typedef void (*registry_t)(void);
void register_func(registry_t);

#endif
/* registry.c */
#include <unistd.h>
#include "registry.h"
```

```
static registry_t func;

void register_func(registry_t f)
{
        func = f;
}

static void on_some_event(void)
{
        ...
        func();
        ...
}
```

既然参数可以是函数指针，返回值同样也可以是函数指针，因此可以有 func()() 这样的调用。返回函数指针的函数在C语言中很少见，而在一些函数式编程语言（例如LISP）中返回函数的函数很常见，基本思想是把函数也当做一种数据来操作，可以输入、输出和参与运算，操作函数的函数称为高阶函数（High-order Function）。

习题

1. 请仿照本节的例23.8自己实现C标准库的qsort(3)和bsearch(3)函数。

23.6 可变参数

到目前为止我们只见过一个带有可变参数的函数printf：

```
int printf(const char *format, ...);
```

在下一章还会见到更多这样的函数。现在我们实现一个简单的myprintf函数：

例23.9 用可变参数实现简单的printf函数

```
#include <stdio.h>
#include <stdarg.h>

void myprintf(const char *format, ...)
{
        va_list ap;
        char c;

        va_start(ap, format);
        while (c = *format++) {
                switch(c) {
                case 'c': {
                        /* char is promoted to int when passed
                        through '...' */
                        char ch = va_arg(ap, int);
                        putchar(ch);
                        break;
                }
```

```
                case 's': {
                        char *p = va_arg(ap, char *);
                        fputs(p, stdout);
                        break;
                }
                default:
                        putchar(c);
                }
        }
        va_end(ap);
}

int main(void)
{
        char c = '1';
        myprintf("c\ts\n", c, "hello");
        return 0;
}
```

要处理可变参数，需要用到 C 标准库的 va_list 类型和 va_start、va_arg、va_end 宏，这些定义在 stdarg.h 头文件中。这些宏是如何取出可变参数的呢？我们首先对照反汇编分析在调用 myprintf 函数时这些参数的内存布局，如图 23.2 所示。

```
      myprintf("c\ts\n", c, "hello");
80484d8:      0f be 44 24 1f          movsbl 0x1f(%esp),%eax
80484dd:      c7 44 24 08 c0 85 04    movl   $0x80485c0,0x8(%esp)
80484e4:      08
80484e5:      89 44 24 04             mov    %eax,0x4(%esp)
80484e9:      c7 04 24 c6 85 04 08    movl   $0x80485c6,(%esp)
80484f0:      e8 4f ff ff ff          call   8048444 <myprintf>
```

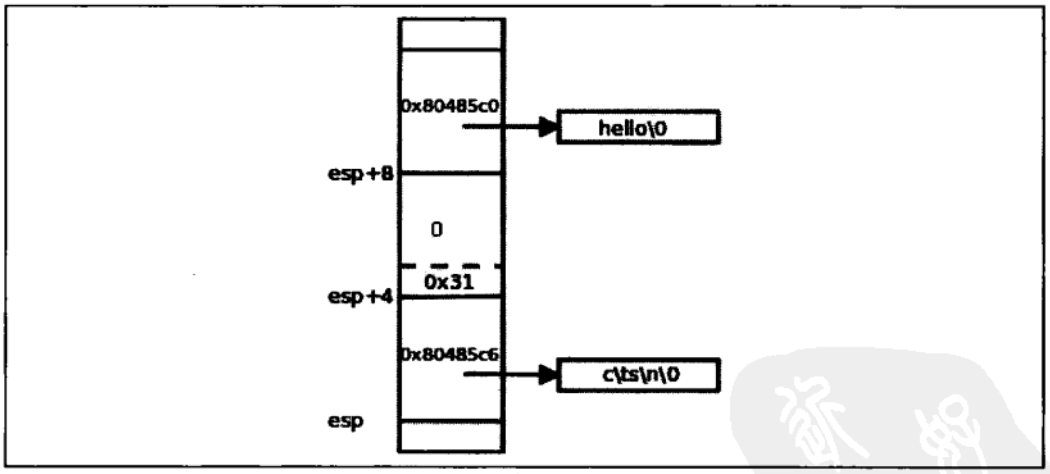


图 23.2 myprintf 函数的参数布局

这些参数是从右向左依次压栈的，所以第一个参数靠近栈顶，第三个参数靠近栈底。这些参数在内存中是连续存放的，每个参数都对齐到 4 字节边界。第一个和第三个参数都是指针类型，各占 4 个字节。第二个参数是 char 型，只占一个字节，但我们在第 14.3.1 节讲过 Default Argument Promotion 规则，这个参数要做 Integer Promotion，由 char 型提升为 int 型要做符号扩展，所以用 movsbl 指令。

现在给出一个 stdarg.h 的简单实现，这个实现出自参考文献[5]第 10 章：

例 23.10 stdarg.h 的一种实现

```
/* stdarg.h standard header */
#ifndef _STDARG
#define _STDARG

/* type definitions */
typedef char *va_list;
/* macros */
#define va_arg(ap, T) \
        (* (T *)(((ap) += _Bnd(T, 3U)) - _Bnd(T, 3U)))
#define va_end(ap) (void)0
#define va_start(ap, A) \
        (void)((ap) = (char *)&(A) + _Bnd(A, 3U))
#define _Bnd(X, bnd) (sizeof (X) + (bnd) & ~(bnd))
#endif
```

这个头文件中的内部宏定义_Bnd(X, bnd)将类型或变量 X 的长度对齐到 bnd+1 字节的整数倍，例如_Bnd(char, 3U)的值是 4，_Bnd(int, 3U)也是 4。这其实就是第 2.5 节习题 1 的整数除法取 Ceiling，只不过现在用位运算来做，而不是用+ - * /来做。

在 myprintf 中定义的 va_list ap;其实是一个指针，va_start(ap, format)使 ap 指向 format 参数的下一个参数，也就是指向图 23.2 中 esp+4 的位置。然后 va_arg(ap, int)把第二个参数的值按 int 型取出来，同时使 ap 指向第三个参数，也就是指向图 23.2 中 esp+8 的位置。然后 va_arg(ap, char *)把第三个参数的值按 char *型取出来，同时使 ap 指向更高的地址。va_end(ap)在我们的简单实现中不起任何作用，在有些实现中可能会把 ap 改写成无效值，C 标准要求在同一个函数中 va_start 和 va_end 要配对。

如果把 myprintf 中的 char ch = va_arg(ap, int);改成 char ch = va_arg(ap, char);，用我们这个 stdarg.h 的简单实现是没有问题的。但如果改用 libc 提供的 stdarg.h，在编译时会报错：

```
$ gcc main.c
main.c: In function 'myprintf':
main.c:14: warning: 'char' is promoted to 'int' when passed through
'...'
main.c:14: note: (so you should pass 'int' not 'char' to 'va_arg')
main.c:14: note: if this code is reached, the program will abort
$ ./a.out
Illegal instruction
```

因此要求 char 型的可变参数必须按 int 型来取，在调用函数时 char 型的可变参数已经提升为 int 型了。

从 myprintf 的例子可以理解 printf 的实现原理，printf 函数根据第一个参数（格式化字符串）来确定后面有几个参数、分别是什么类型。确保参数的类型、个数与格式化字符串的描述相匹配是调用者的责任，实现者只管按格式化字符串的描述

从栈上取参数，如果调用者传递的参数类型或个数不正确，实现者从栈上取到的参数就是错的。

还有一种办法可以确定可变参数的个数，就是在参数列表的末尾传一个 Sentinel，例如 NULL。execl(3)就采用这种方法确定参数的个数。下面实现一个 printlist 函数，可以打印若干个传入的字符串。

例 23.11　根据 Sentinel 判断可变参数的个数

```
#include <stdio.h>
#include <stdarg.h>

void printlist(int begin, ...)
{
        va_list ap;
        char *p;

        va_start(ap, begin);
        p = va_arg(ap, char *);

        while (p != NULL) {
                fputs(p, stdout);
                putchar('\n');
                p = va_arg(ap, char*);
        }
        va_end(ap);
}

int main(void)
{
        printlist(0, "hello", "world", "foo", "bar", NULL);
        return 0;
}
```

其实 printlist 函数的第一个参数 begin 并没有用到，但是这个参数必须写，因为 C 语言规定可变参数列表的...前面至少要定义一个有名字的参数，要把参数列表中最后一个有名字的参数提供给 va_start，这样 va_start 才能找到可变参数在栈上的位置。实现者应该在文档中说明参数列表必须以 NULL 结尾，如果调用者不遵守这个约定，函数就会出错。

习题

1．实现一个功能更完整的 myprintf，能够处理%c、%s、%d（对应参数是 int 型，以十进制打印）、%o（对应参数是 unsigned int 型，以八进制打印）、%x（对应参数是 unsigned int 型，以十六进制打印）、%f（对应参数是 double 型，打印到小数点后 6 位）、%%（打印一个%号）等转换说明，在实现中不许调用 printf(3)这个 Man Page 中描述的任何函数。

第 24 章
C 标准库

在前面的各章中我们已经见过 C 标准库的一些用法，总结如下：

- 我们最常用的是包含 stdio.h，使用其中声明的 printf 函数，这个函数在 libc 中实现，程序在运行时要动态链接 libc 共享库。
- 在例 3.1 中用到了 math.h 中声明的 sin 和 log 函数，使用这些函数需要动态链接 libm 共享库。
- 在第 8.2 节用到了 stdlib.h 中声明的 rand 函数，还提到这个头文件中定义的 RAND_MAX 常量，在例 8.5 中用到了 stdlib.h 中声明的 srand 函数和 time.h 中声明的 time 函数。使用这些函数需要动态链接 libc 共享库。
- 在第 18.2 节讲了 stdlib.h 中声明的 exit 函数，使用这个函数需要动态链接 libc 共享库。
- 在第 11.6 节用到了 assert.h 中定义的 assert 宏，在第 20.4 节我们看到了这个宏的一种实现，它的实现需要调用 stdio.h 和 stdlib.h 中声明的函数，所以使用这个宏也需要动态链接 libc 共享库。
- 在第 15.2.4 节提到 size_t 类型在 stddef.h 中定义，还提到 stdint.h 中定义一些类型名，在第 22 章提到 NULL 指针和 ptrdiff_t 类型也在 stddef.h 中定义。
- 在第 23.1 节介绍了 stdlib.h 中声明的 malloc 和 free 函数以及 string.h 中声明的 strcpy 和 strncpy 函数，使用这些函数需要动态链接 libc 共享库。
- 在第 23.6 节介绍了 stdarg.h 中定义的 va_list 类型和 va_arg、va_start、va_end 等宏定义，并给出了一种实现，这些宏定义的实现并没有调用库函数，所以不依赖于某个共享库，这一点和 assert 不同。

总结一下，Linux 平台提供的 C 标准库包括：

- 一组头文件，定义了很多类型和宏，声明了很多库函数和全局变量。这些头文件放在哪些目录下取决于不同的 Linux 发行版和编译器版本，在我的系统上，stdarg.h 和 stddef.h 位于/usr/lib/gcc/i486-linux-gnu/4.4.3/include 目录下，stdio.h、stdlib.h、time.h、math.h、assert.h 位于/usr/include 目录下。C99 标准定义的头文件有 24 个，本书只介绍其中最基本、最常用的几个。

- 一组库文件，提供了库函数和全局变量的定义。大多数库函数在 libc 共享库中，有些库函数在另外的共享库中，例如数学函数在 libm 中。在第 19.4 节讲过，通常 libc 共享库是/lib/libc.so.6，而我的系统启用了 hwcap 机制，libc 共享库是/lib/tls/i686/cmov/libc.so.6。

本章集中介绍一些最基本和最常用的库函数（也包括一些不属于 C 标准但在 UNIX 平台上很常用的函数），写这一章是为了介绍字符串操作和文件操作的基本概念和方法，而不是为了写一本 C 标准库函数的参考手册，Man Page 已经是一本很好的手册了，读者学完这一章之后在开发时应该查阅 Man Page，而不是把我这一章当参考手册来翻，所以本章不会面面俱到介绍所有的库函数。很多技术书的作者给自己的书太多定位，既想写成一本入门教程，又想写成一本参考手册，我觉得这样不好，读者过于依赖技术书就失去了看真正的手册的能力。

24.1　字符串操作函数

程序按功能划分可分为数值计算、符号处理和 I/O 操作三类，符号处理程序占相当大的比例，符号处理程序无处不在，编译器、浏览器、Office 套件等程序的主要功能都是符号处理。无论多复杂的符号处理都是由各种基本的字符串操作组成的，本节介绍如何用 C 的库函数做字符串赋初值、取长度、拷贝、连接、比较、搜索、分割等基本操作。

24.1.1　给字符串赋初值

```
#include <string.h>

void *memset(void *s, int c, size_t n);
返回值：s 指向哪，返回的指针就指向哪
```

memset 函数给一段内存（从 s 所指的地址开始的 n 个字节）赋初值，把每个字节都填充为 c 的值。通常调用 memset 时传给 c 的值是 0，把一块内存区清零。例如定义 char buf[10];，如果它是全局变量或静态变量则自动初始化为 0（位于.bss 段），如果它是函数的局部变量则初值不确定，可以用 memset(buf, 0, 10)清零，由 malloc 分配的内存初值也是不确定的，也可以用 memset 清零。

为什么规定参数 c 的类型是 int 呢？C 标准说 memset 函数要把参数 c 转换成 unsigned char 型再填充到每个字节中，那为什么不直接规定参数 c 是 unsigned char 型呢？这个问题很难说清楚，参考文献[6]也没有解释，我认为主要是历史原因。网上有几种说法，我列在这里供读者自己判断，如果你不是像我一样 Paranoid，没兴趣追究这个，可以直接跳到下一小节。

一种说法认为和 Integer Promotion 有关。在 Old Style C 的时代调用函数不必声明函数原型，由于编译器不知道参数类型，char 型参数都要提升为 int 型，Integer

Promotion 规则就是这么来的。现在的 C 语言虽然可以声明函数原型，但仍然保留了 Old Style C 的语法和 Integer Promotion 规则，以前的函数接口也保留了下来，所以本来应该是 char 型的参数都写成 int 型，而我们自己定义新的函数接口就不必遵循这个惯例了。

另一种说法认为，规定参数为 int 型是为了传字符常量方便，因为字符常量也是 int 型的，比如调用 memset(buf, 'A', 10)。C 标准库中像这样的函数有很多，比如稍后要讲的 strchr、strrchr、fputc 等，再比如 ctype.h 中声明的字符处理函数：

```
#include <ctype.h>

int isalnum(int c);
int isalpha(int c);
int isascii(int c);
int isblank(int c);
int iscntrl(int c);
int isdigit(int c);
int isgraph(int c);
int islower(int c);
int isprint(int c);
int ispunct(int c);
int isspace(int c);
int isupper(int c);
int isxdigit(int c);
返回值：如果 c 属于该字符类则返回非零，否则返回零

int toupper(int c);
int tolower(int c);
返回值：如果能转换就返回转换之后的字母，如果不能转换就返回 c 本身
```

这些函数都带一个字符参数，参数类型都是 int，第一组函数判断某个字符是否属于某个字符类，第二组函数做大小写字母的相互转换，比如调用 isalpha('+')判断'+'是不是字母，调用 toupper('a')把小写字母'a'转成大写。另外，C 标准规定传给这些函数的参数可以是 EOF，EOF 是一个特殊的值，它的类型是 int 而不是 char，如果转换成 char 型会丢失信息（在第 24.2.5 节详细解释），所以参数类型必须是 int。

24.1.2 取字符串的长度

```
#include <string.h>

size_t strlen(const char *s);
返回值：字符串的长度
```

strlen 函数返回 s 所指的字符串的长度。该函数从 s 所指的第一个字符开始找'\0'字符，一旦找到就返回，返回的长度不包括'\0'字符在内。例如定义 char buf[] = "hello";，则 strlen(buf)的值是 5。注意，如果定义 char buf[5] = "hello";，则调用 strlen(buf)是危险的，会造成数组访问越界。

24.1.3　拷贝字符串

在第 23.1 节中介绍了 strcpy 和 strncpy 函数，拷贝以 Null 结尾的字符串，strncpy 还带一个参数指定最多拷贝多少个字节，另外注意 strncpy 并不保证目标缓冲区以'\0'结尾。现在介绍 memcpy 和 memmove 函数，准确地说，这两个函数不是拷贝字符串，而是拷贝固定的字节数。

```
#include <string.h>

void *memcpy(void *dest, const void *src, size_t n);
void *memmove(void *dest, const void *src, size_t n);
返回值：dest 指向哪，返回的指针就指向哪
```

memcpy 函数从 src 所指的内存地址拷贝 n 个字节到 dest 所指的内存地址，和 strncpy 不同，memcpy 并不是遇到'\0'就结束，而是一定会拷贝完 n 个字节。我们可以得出这几个库函数的命名规律，以 str 开头的函数操作以 Null 结尾的字符串，而以 mem 开头的函数则不关心'\0'字符，或者说这些函数只是把参数看作 n 个字节，并不看作字符串，因此参数的指针类型是 void *而非 char *。

memmove 也是从 src 所指的内存地址拷贝 n 个字节到 dest 所指的内存地址，虽然叫 move 但其实也是拷贝而非移动。但是和 memcpy 有一点不同，memcpy 的两个参数 src 和 dest 所指的内存区间如果重叠则无法保证正确拷贝，而 memmove 却可以正确拷贝。假设定义了一个数组 char buf[20] = "hello world\n";，如果想把其中的字符串往后移动一个字节（变成"hhello world\n"），调用 memcpy 是无法保证正确拷贝的：

例 24.1　错误的 memcpy 调用

```
#include <stdio.h>
#include <string.h>

int main(void)
{
        char buf[20] = "hello world\n";
        memcpy(buf + 1, buf, 13);
        printf("%s", buf);
        return 0;
}
```

在有些系统上可能跑出 hhhllooworrd 这样的结果。如果把代码中的 memcpy 改成 memmove 则可以保证正确拷贝。memmove 可以这样实现：

```
void *memmove(void *dest, const void *src, size_t n)
{
        char temp[n];
        int i;
        char *d = dest;
        const char *s = src;

        for (i = 0; i < n; i++)
```

```
                temp[i] = s[i];
        for (i = 0; i < n; i++)
                d[i] = temp[i];

        return dest;
}
```

借助于一个临时缓冲区temp，即使src和dest所指的内存区间有重叠也能正确拷贝。思考一下，如果不借助于临时缓冲区能不能正确处理重叠内存区间的拷贝？

用memcpy为什么会得到hhhllooworrd这个奇怪的结果呢？根据这个结果猜测的一种可能的实现是：

```
void *memcpy(void *dest, const void *src, size_t n)
{
        char *d = dest;
        const char *s = src;
        int *di;
        const int *si;
        int r = n % 4;
        while (r--)
                *d++ = *s++;
        di = (int *)d;
        si = (const int *)s;
        n /= 4;
        while (n--)
                *di++ = *si++;

        return dest;
}
```

在32位的x86平台上，每次拷贝1个字节需要一条指令，每次拷贝4个字节也只需要一条指令，为了提高拷贝的效率，我们先处理完零头然后4个字节4个字节地拷贝。注意这个实现并不正确，把void *指针转成int *指针来访问应该考虑对齐的问题，请读者自己实现一个更完善的版本。

C99的restrict关键字

我们来看一个和memcpy/memmove类似的问题。下面的函数将两个数组中对应的元素相加，结果保存在第三个数组中。

```
    void vector_add(const double *x, const double *y, double
*result)
    {
            int i;
            for (i = 0; i < 64; ++i)
                    result[i] = x[i] + y[i];
    }
```

如果这个函数要在多处理器的计算机上运行，编译器可以做这样的优化：把这一个循环拆成两个循环，一个处理器计算i值从0到31的循环，另一个处理器计算i值从32到63的循环，这样两个处理器可以同时工作，使计算时间缩短一半。但是这样的编译优化能保证得出正确结果吗？假

如 result 和 x 所指的内存区间是重叠的，result[0]其实是 x[1]，result[i]其实是 x[i+1]，这两个处理器就不能各干各的事情了，因为第二个处理器的计算过程依赖于第一个处理器的最终计算结果，这种情况下编译优化的结果是错的。这样看来编译器是不敢随便做优化了，那么多处理器提供的并行性就无法利用，岂不可惜？为此，C99 引入 restrict 关键字，如果程序员把上面的函数声明为 void vector_add(const double *restrict x, const double *restrict y, double *restrict result)，就是告诉编译器可以放心地对这个函数做优化，由程序员负责保证这些指针所指的内存区间互不重叠。

由于 restrict 是 C99 引入的新关键字，目前 Linux 的 Man Page 还没有更新，所以都没有 restrict 关键字，本书的函数原型都取自 Man Page，所以也都没有 restrict 关键字。但在 C99 标准中库函数的原型都在必要的地方加了 restrict 关键字，在 C99 中 memcpy 的原型是 void *memcpy(void * restrict s1, const void * restrict s2, size_t n);，就是告诉调用者，这个函数的实现可能会做些优化，编译器也可能会做些优化，传进来的指针不允许指向重叠的内存区间，否则结果可能是错的，而 memmove 的原型是 void *memmove(void *s1, const void *s2, size_t n);，没有 restrict 关键字，说明传给这个函数的指针允许指向重叠的内存区间。在 restrict 关键字出现之前都是用自然语言描述哪些函数的参数不允许指向重叠的内存区间，例如在 C89 标准的库函数一章开头提到，本章描述的所有函数，除非特别说明，都不应该接收两个指针参数指向重叠的内存区间，例如调用 sprintf 时传进来的格式化字符串和结果字符串重叠，诸如此类的调用都是非法的。本书也遵循这一惯例，除了像 memmove 这样的函数特别说明之外，一般都不允许两个指针参数指向重叠的内存区间。

关于 restrict 关键字更详细的解释可以查阅参考文献[29]。

字符串的拷贝也可以用 strdup(3)函数，这个函数属于 POSIX 标准但并不属于 C 标准。

```
#include <string.h>

char *strdup(const char *s);
返回值：指向新分配的字符串
```

这个函数调用 malloc 动态分配内存，把字符串 s 拷贝到新分配的内存中然后返回。用这个函数省去了事先为新字符串分配内存的麻烦，但是用完之后要记得调用 free 释放新字符串的内存。

24.1.4 连接字符串

```
#include <string.h>

char *strcat(char *dest, const char *src);
char *strncat(char *dest, const char *src, size_t n);
返回值：dest 指向哪，返回的指针就指向哪
```

strcat把src所指的字符串连接到dest所指的字符串后面，例如：

```
char d[10] = "foo";
char s[10] = "bar";
strcat(d, s);
printf("%s %s\n", d, s);
```

调用strcat函数后，缓冲区s的内容没变，缓冲区d中保存着字符串"foobar"，注意原来"foo"后面的'\0'被连接上来的字符串"bar"覆盖掉了，"bar"后面的'\0'仍保留。

strcat和strcpy有同样的问题，调用者必须确保dest缓冲区足够大，否则会导致缓冲区溢出错误。strncat函数通过参数n指定一个长度，就可以避免缓冲区溢出错误。注意这个参数n的含义和strncpy的参数n不同，它并不是缓冲区dest的长度，而是表示最多从src缓冲区中取n个字符（不包括结尾的'\0'）连接到dest后面。如果src中前n个字符没有出现'\0'，则取前n个字符再加一个'\0'连接到dest后面，所以strncat总是保证dest缓冲区以'\0'结尾，这一点又和strncpy不同，strncpy并不保证dest缓冲区以'\0'结尾。例如：

```
void openfoorc(const char *homedir)
{
        char path[100] = {0};

        strncpy(path, homedir, sizeof(path)-1);
        strncat(path, "/.foorc", sizeof(path)-strlen(path)-1);
        /* open path */
        ...
}
```

24.1.5 比较字符串

```
#include <string.h>

int memcmp(const void *s1, const void *s2, size_t n);
int strcmp(const char *s1, const char *s2);
int strncmp(const char *s1, const char *s2, size_t n);
返回值：负值表示s1小于s2，0表示s1等于s2，正值表示s1大于s2
```

memcmp从前到后逐个比较缓冲区s1和s2的前n个字节（不管里面有没有'\0'），如果s1和s2的前n个字节全都一样就返回0，如果遇到不一样的字节，s1的字节比s2小就返回负值，s1的字节比s2大就返回正值。

strcmp把s1和s2当做字符串比较，从前到后逐个比较每个字符，结束时有两种可能：

1．如果两个字符串完全相同，同时遇到'\0'字符，那么返回0。

2．如果在比较过程中遇到不同的字符，那么把两个字符相比较返回正值或负值。如果在一个字符串遇到'\0'，比另一个字符串先结束，也属于这种情况，这时把'\0'和另一个字符串中对应的字符相比较返回正值或负值。

按照上面的比较准则，"ABC"比"abc"小，"ABCD"比"ABC"大，"123A9"比"123B2"小。

strncmp 的比较结束条件是：要么比较完 n 个字符结束（类似于 memcmp），要么当遇到不同的字符或者当一个字符串比另一个先遇到'\0'时结束（类似于 strcmp）。例如，strncmp("ABCD", "ABC", 3)的返回值是 0，而 strncmp("ABCD", "ABC", 4)的返回值是正值。

```
#include <strings.h>

int strcasecmp(const char *s1, const char *s2);
int strncasecmp(const char *s1, const char *s2, size_t n);
返回值：负值表示 s1 小于 s2，0 表示 s1 等于 s2，正值表示 s1 大于 s2
```

这两个函数和 strcmp/strncmp 类似，但在比较过程中忽略大小写，'A'和'a'认为是相等的。这两个函数不属于 C 标准，是 POSIX 标准定义的。

24.1.6　搜索字符串

```
#include <string.h>

char *strchr(const char *s, int c);
char *strrchr(const char *s, int c);
返回值：如果找到字符 c，返回字符串 s 中指向字符 c 的指针，如果找不到就返回 NULL
```

strchr 在字符串 s 中从左到右查找字符 c，找到它第一次出现的位置就返回，返回的指针就指向这个位置，如果找不到字符 c 就返回 NULL。strrchr 和 strchr 类似，但是返回字符 c 最后一次出现的位置，函数名中间多了一个字母 r 可以理解为 Reverse。

```
#include <string.h>

char *strstr(const char *haystack, const char *needle);
返回值：如果找到子串，返回值指向子串的开头，如果找不到就返回 NULL
```

strstr 在一个长字符串中从前到后找一个子串（Substring），找到子串第一次出现的位置就返回，返回值指向子串的开头，如果找不到就返回 NULL。这两个参数名很形象，在干草堆 haystack 中找一根针 needle，按中文的说法叫大海捞针，显然 haystack 是长字符串，needle 是要找的子串。

搜索子串有一个显而易见的算法，可以用两层循环，外层循环把 haystack 中每个字符的位置依次假定为子串的开头，内层循环从这个位置开始逐个比较 haystack 和 needle 的每个字符是否相同。想想这个算法最多需要做多少次比较？其实有比这个算法高效得多的算法，有兴趣的读者可以查阅参考文献[16]第 32 章。

24.1.7　分割字符串

很多文件格式或协议格式中会规定一些分隔符，例如/etc/passwd 文件中保存着系

统的账号信息：

```
$ cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
...
```

每条记录占一行，也就是说记录之间的分隔符是换行符，每条记录又由若干个字段组成，这些字段包括用户名、密码、用户 id、组 id、个人信息、主目录、登录 Shell，字段之间的分隔符是:号。解析这样的字符串需要根据分隔符把字符串分割成几段，C 标准库提供的 strtok 函数可以很方便地完成分割字符串的操作。tok 是 Token 的缩写，分割出来的每一段字符串称为一个 Token。

```
#include <string.h>

char *strtok(char *str, const char *delim);
char *strtok_r(char *str, const char *delim, char **saveptr);
返回值：每次调用依次返回字符串 str 中的一个 Token，如果到达 str 末尾就返回 NULL
```

参数 str 是待分割的字符串，delim 是分隔符，可以指定一个或多个分隔符，strtok 遇到其中任何一个分隔符就会分割字符串，看下面的例子。

例 24.2 strtok

```
#include <stdio.h>
#include <string.h>

int main(void)
{
        char str[] = "root:x::0:root:/root:/bin/bash:";
        char *token;

        token = strtok(str, ":");
        printf("%s\n", token);
        while ( (token = strtok(NULL, ":")) != NULL)
                printf("%s\n", token);

        return 0;
}
```

执行结果是：

```
$ ./a.out
root
x
0
root
/root
/bin/bash
```

结合这个例子我们看看 strtok 是怎样分割字符串的：冒号是分隔符，把"root:x::0:root:/root:/bin/bash:"这个字符串分隔成"root"、"x"、""、"0"、"root"、"/root"、

"/bin/bash"、""等几个 Token，但是空字符串的 Token 直接忽略而不返回。第一次调用时要传字符串的首地址给 strtok 的第一个参数，以后每次调用只要传 NULL 给第一个参数就可以了，strtok 函数自己会记住上次处理到字符串的什么位置（显然这是通过 strtok 函数里的一个静态指针变量记住的）。

用 gdb 跟踪这个程序，会发现 str 字符串被 strtok 不断修改，每次调用 strtok 把 str 中的一个分隔符改成'\0'，分割出一个小字符串，并返回这个小字符串的首地址。

```
(gdb) start
Temporary breakpoint 1 at 0x804847d: file main.c, line 5.
Starting program: /home/akaedu/a.out

Temporary breakpoint 1, main () at main.c:5
5   {
(gdb) n
6               char str[] = "root:x::0:root:/root:/bin/bash:";
(gdb) （直接回车）
9               token = strtok(str, ":");
(gdb) display str
1: str = "root:x::0:root:/root:/bin/bash:"
(gdb) n
10              printf("%s\n", token);
1: str = "root\000x::0:root:/root:/bin/bash:"
(gdb) （直接回车）
root
11              while ( (token = strtok(NULL, ":")) != NULL)
1: str = "root\000x::0:root:/root:/bin/bash:"
(gdb) （直接回车）
12                      printf("%s\n", token);
1: str = "root\000x\000:0:root:/root:/bin/bash:"
(gdb) （直接回车）
x
11              while ( (token = strtok(NULL, ":")) != NULL)
1: str = "root\000x\000:0:root:/root:/bin/bash:"
```

刚才提到在 strtok 函数中应该有一个静态指针变量记住上次处理到字符串的什么位置，所以不必每次调用都把字符串的当前处理位置传给 strtok。但在函数中使用静态变量是不好的，这样的函数是不可重入的（可重入性的概念请查阅参考文献[31]的 10.6 节），所以 POSIX 标准定义了一个不使用静态变量的 strtok_r 函数（这个函数不属于 C 标准），调用者需要自己分配一个指针变量来维护字符串的当前处理位置，每次调用时要传这个指针变量的地址给 strtok_r 的第三个参数，告诉 strtok_r 从哪里开始处理，strtok_r 返回时再把新的处理位置写回这个指针变量中（这是一个 Value-result 参数）。strtok_r 末尾的 r 表示可重入（Reentrant）。关于 strtok_r 的用法 Man Page 上有一个很好的例子：

例 24.3　strtok_r

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[])
```

```
{
        char *str1, *str2, *token, *subtoken;
        char *saveptr1, *saveptr2;
        int j;

        if (argc != 4) {
                fprintf(stderr, "Usage: %s string delim subdelim \n",
                        argv[0]);
                exit(EXIT_FAILURE);
        }

        for (j = 1, str1 = argv[1]; ; j++, str1 = NULL) {
            token = strtok_r(str1, argv[2], &saveptr1);
            if (token == NULL)
                    break;
            printf("%d: %s\n", j, token);

            for (str2 = token; ; str2 = NULL) {
                    subtoken = strtok_r(str2, argv[3], &saveptr2);
                    if (subtoken == NULL)
                            break;
                    printf(" --> %s\n", subtoken);
            }
        }

        exit(EXIT_SUCCESS);
}
```

```
$ ./a.out 'a/bbb///cc;xxx:yyy:' ':;' '/'
1: a/bbb///cc
 --> a
 --> bbb
 --> cc
2: xxx
 --> xxx
3: yyy
 --> yyy
```

a/bbb///cc;xxx:yyy:这个字符串有两级分隔符，一级分隔符是:号或;号，把这个字符串分割成 a/bbb///cc、xxx、yyy 三个子串，只有第一个子串中有二级分隔符，二级分隔符是/斜线，第一个子串被进一步分割成 a、bbb、cc 三个二级子串。由于 strtok_r 不使用静态变量，而是要求调用者自己保存字符串的当前处理位置，所以这个例子可以在分割一级子串的过程中穿插着分割二级子串。建议读者用 gdb 的 display 命令跟踪 argv[1]、saveptr1 和 saveptr2，以理解 strtok_r 的工作方式。

Man Page 的 BUGS 部分指出了用 strtok 和 strtok_r 函数需要注意的问题：

- 这两个函数要改写字符串以达到分割的效果。
- 这两个函数不能用于分割字符串字面值，因为试图改写.rodata 段会产生段错误。
- 在做了分割之后，字符串中的分隔符就丢失了，就被'\0'覆盖了。
- strtok 函数使用了静态变量，它不是线程安全的，必要时应该用可重入的 strtok_r 函数，线程安全的概念请查阅参考文献[31]的 12.5 节。

习题

1. 出于练习的目的，strtok 和 strtok_r 函数非常值得自己动手实现一遍，在这个过程中不仅可以更深刻地理解这两个函数的工作原理，也为理解“可重入性”和“线程安全”这两个重要概念打下基础。

2. 解析 URL 中的路径和查询字符串。动态网页的 URL 末尾通常带有查询，例如：

http://www.google.cn/search?complete=1&hl=zh-CN&ie=GB2312&q=linux&meta=
http://www.baidu.com/s?wd=linux&cl=3

比如上面第一个例子，http://www.google.cn/search 是路径部分，?号后面的 complete=1&hl=zh-CN&ie=GB2312&q=linux&meta= 是查询字符串，由五个“key=value”形式的键值对组成，以&隔开，值可以是空字符串，比如这个例子中的键 meta 对应的值是空字符串。

现在要求实现一个函数，传入一个带查询字符串的 URL，首先检查输入格式的合法性，然后对 URL 进行切分，将路径部分和各键值对分别传出，请仔细设计函数接口以便传出这些字符串。如果函数中有动态分配内存的操作，还要另外实现一个释放内存的函数。完成之后，为自己设计的函数写一个 Man Page。

24.2　标准 I/O 库函数

24.2.1　文件的基本概念

我们已经多次用到了文件，例如源文件、目标文件、可执行文件、库文件等，现在学习如何用 C 标准库对文件进行读写操作。本节介绍的大部分函数在头文件 stdio.h 中声明，称为标准 I/O 库函数。

文件可分为文本文件（Text File）和二进制文件（Binary File）两种，源文件是文本文件，而目标文件、可执行文件和库文件是二进制文件。文本文件是用来保存字符的，文件中的字节都是字符的某种编码（例如 ASCII 或 UTF-8），用 cat 命令可以查看文本文件的内容，用 vi 可以编辑文本文件；而二进制文件不是用来保存字符的，文件中的字节表示其他含义，例如可执行文件中有些字节表示指令，有些字节表示各 Section 在文件中的位置，有些字节表示各 Segment 的加载地址。

在第 17.5.1 节讲过用 hexdump 命令查看二进制文件，现在我们再介绍一种二进制文件查看工具 od。用 vi 编辑一个文件 textfile，在其中输入“5678”然后保存退出，用 ls -l 命令可以看到它的长度是 5：

```
$ ls -l textfile
-rw-r--r-- 1 akaedu akaedu 5 2010-03-20 10:58 textfile
```

“5678”四个字符各占一个字节，vi会自动在文件末尾加一个换行符，所以文件长度是5。很多程序要求文本文件的每一行末尾都要有换行符，最后一行也不例外，如果一个源文件的最后一行末尾没有换行符，用gcc编译会报错，所以vi要在最后一行末尾加换行符。用od命令查看该文件的内容：

```
$ od testfile
0000000 033065 034067 000012
0000005
$ od -tx1 -tc -Ax textfile
000000 35 36 37 38 0a
         5  6  7  8 \n
000005
```

od命令默认以八进制数显示文件中的字节，左边的文件地址默认也是八进制的，od是octal dump的缩写。-tx1选项要求以十六进制数显示文件中的字节，并且一个字节一组，-tc选项要求以字符形式显示文件中的ASCII码，-Ax选项要求以十六进制数显示左边的文件地址。

我们看到这个文件中保存了5个字符，每个字符占一个字节，以ASCII码保存，ASCII码的取值范围是0～127，所以文件中每个字节只用到低7位，最高位都是0。是不是只包含ASCII码的文件就叫文本文件呢？其实文本文件是一个模糊的概念，通常我们说的文本文件是指用vi可以编辑的文件，例如/etc目录下的各种配置文件，这些文件中只包含ASCII码的可见字符，而不包含Null字符等不可见字符，也不包含最高位是1的非ASCII码字节。从广义上来说，只要是专门保存字符的文件都算文本文件，包含不可见字符的也算，采用其他字符编码（例如UTF-8编码）的也算。

24.2.2 fopen/fclose

在操作文件之前要用fopen打开文件，操作完毕要用fclose关闭文件。打开文件就是在操作系统中分配一些资源用于保存该文件的状态信息，并得到该文件的标识，以后用户程序就可以用这个标识对文件做各种操作，关闭文件则释放文件在操作系统中占用的资源，使文件的标识失效，用户程序就无法再操作这个文件了。

```
#include <stdio.h>

FILE *fopen(const char *path, const char *mode);
返回值：成功返回文件指针，出错返回NULL并设置errno
```

path是文件的路径名，mode表示打开方式。如果文件打开成功，就返回一个FILE *文件指针来标识这个文件。以后调用其他函数对文件做读写操作都要提供这个指针，以指明是对哪个文件进行操作。

FILE是C标准库中定义的结构体类型，其中包含该文件在内核中的标识[①]、用户空间I/O缓冲区和当前读写位置等信息。但调用者不必知道FILE结构体都有哪些

① 即文件描述符，读者可以查阅参考文献[31]的3.2节。

成员，调用者只是把文件指针在库函数接口之间传来传去，而文件指针所指的FILE结构体的成员在库函数内部维护，调用者不能直接访问这些成员，这也是封装思想的一种应用。像FILE *这样的指针称为不透明指针（Opaque Pointer）或句柄（Handle），FILE *指针就像一个把手（Handle），抓住这个把手就可以打开门或抽屉，但用户只能抓这个把手，而不能直接抓门或抽屉。

现在总结一下我们讲过的封装（Encapsulation）：在第4.2节讲过把一组语句封装成一个函数，这是最简单的封装；在第19.2节讲过用static关键字封装模块的内部变量和函数；现在我们讲到用不透明指针封装一个类型的内部表示。封装是为了隔离，为了使一个模块的改动不会波及其他模块，从而保证整个系统的复杂性是可以控制的。

下面说说参数path和mode，path可以是相对路径也可以是绝对路径，mode表示打开方式是读还是写。比如 fp = fopen("/tmp/file2", "w");表示打开绝对路径/tmp/file2，只做写操作，path也可以是相对路径，比如fp = fopen("file.a", "r");表示在当前工作目录下打开文件file.a，只做读操作，再比如fp = fopen("../a.out", "r");只读打开当前工作目录上一层目录下的a.out，fp = fopen("Desktop/file3", "w");只写打开当前工作目录下子目录Desktop中的file3。

mode参数是一个字符串，由rwab+五个字符组合而成。r表示读，w表示写，a表示追加（Append），即在文件末尾追加数据使文件的尺寸增大。b表示二进制模式，不写b则表示文本模式。

为什么打开方式要区分文本模式和二进制模式呢？主要是因为换行符的问题，建议读者仔细看看Wikipedia的Newline词条，一个换行符原来可以这么复杂。我们知道Windows系统的文本文件的换行符是\r\n（ASCII码0x0d 0x0a），如果以文本模式打开，则从文件中读到的\r\n会自动转换，看起来像是一个\n字符，而写入文件的\n自动转换成\r\n保存，如果以二进制模式打开则不会做这种转换。UNIX系统的文本文件的换行符是\n，不管以文本模式还是二进制模式打开都一样，所以在UNIX系统上这两种模式没区别，本书示例代码的mode参数都省略b，对文本模式和二进制模式不加区分，但要注意这样的代码对于非UNIX操作系统是不可移植的。

rwa+四个字符有以下6种合法的组合：

"r"

只读，文件必须已存在。

"w"

只写，如果文件不存在则创建，如果文件已存在则把文件长度截断（Truncate）为0字节再重新写，也就是替换掉原来的文件内容。

"a"

只能在文件末尾追加数据，如果文件不存在则创建。

"r+"

允许读和写，文件必须已存在。

"w+"

允许读和写，如果文件不存在则创建，如果文件已存在则把文件长度截断为 0 字节再重新写。

"a+"

允许读和追加数据，如果文件不存在则创建。

想一想，如果要打开一个文件做写操作，覆盖文件开头 1KB 的内容，而后面的内容保持不变，应该以哪种模式打开？

在打开一个文件时如果出错，fopen 将返回 NULL 并设置 errno，errno 稍后介绍。在程序中应该做出错处理，通常这样写：

```
if ( (fp = fopen("/tmp/file1", "r")) == NULL) {
        printf("error open file /tmp/file1!\n");
        exit(1);
}
```

比如/tmp/file1 这个文件不存在，而 r 打开方式又不会创建这个文件，fopen 就会出错返回。

再说说 fclose 函数。

```
#include <stdio.h>

int fclose(FILE *fp);
返回值：成功返回 0，出错返回 EOF 并设置 errno
```

把文件指针传给 fclose 可以关闭它所标识的文件，关闭之后该文件指针就无效了，不能再使用了。如果 fclose 调用出错（比如传给它一个无效的文件指针）则返回 EOF 并设置 errno，errno 稍后介绍，EOF 在 stdio.h 中定义：

```
/* End of file character.
   Some things throughout the library rely on this being -1.  */
#ifndef EOF
# define EOF (-1)
#endif
```

它的值是-1。fopen 调用应该和 fclose 调用配对，打开文件操作完之后一定要记得关闭。如果不调用 fclose，在进程退出时内核会自动关闭该进程打开的所有文件，但不能因此就忽略 fclose 调用，如果写一个长年累月运行而不退出的程序（比如

网络服务器程序），打开的文件都不关闭，堆积得越来越多，就会占用越来越多的系统资源。

24.2.3 stdin/stdout/stderr

我们经常用 printf 打印到屏幕，也用过 scanf 读键盘输入，这些也属于 I/O 操作，但不是对文件做 I/O 操作而是对终端设备做 I/O 操作。所谓终端（Terminal）是指人机交互的设备，也就是可以接收用户输入并输出信息给用户的设备。在计算机刚诞生的年代，终端是电传打字机和打印机，现在的终端通常是键盘和显示器。终端设备和文件一样也需要先打开后操作，终端设备也有对应的路径名，/dev/tty 就表示和当前进程相关联的终端设备（称为进程的控制终端），/dev/tty 不是一个普通文件，它不表示磁盘上的一组数据，而是表示一个设备。用 ls 命令查看这个文件：

```
$ ls -l /dev/tty
crw-rw-rw- 1 root dialout 5, 0 2010-03-20 19:31 /dev/tty
```

开头的 c 表示文件类型是字符设备。中间的“5, 0”是它的设备号，主设备号 5，次设备号 0，主设备号标识内核中的一个设备驱动程序，次设备号标识该设备驱动程序管理的一个设备。内核通过设备号找到相应的驱动程序，完成对该设备的操作。我们知道常规文件的这一列应该显示文件长度，而设备文件的这一列显示设备号，这表明设备文件没有“文件长度”的属性，设备文件在磁盘上不保存数据，对设备文件做读写操作并不是读写磁盘上的数据，而是在读写设备。

UNIX 的传统是 Everything is a file，键盘、显示器、串口、磁盘等设备在/dev 目录下都有一个特殊的设备文件与之对应，这些设备文件也可以像普通文件一样打开、读、写和关闭，使用的函数接口是相同的。本书中不严格区分“文件”和“设备”这两个概念，遇到“文件”这个词，读者可以根据上下文理解它是指普通文件还是设备，如果需要强调是保存磁盘数据的普通文件，本书会用“常规文件”（Regular File）这个词。

我们还没有打开过终端设备，为什么就可以用 printf 和 scanf 来读写呢？因为程序启动时会自动打开终端设备[②]，并且用三个 FILE *指针 stdin、stdout 和 stderr 指向这个设备，这三个文件指针是 libc 中定义的全局变量，在 stdio.h 中声明，printf 向 stdout 写，而 scanf 从 stdin 读，用户程序也可以直接使用这三个文件指针。stdin、stdout 和 stderr 的打开方式都是可读可写的，但通常 stdin 只用于读操作，称为标准输入（Standard Input），stdout 只用于写操作，称为标准输出（Standard Output），stderr 也只用于写操作，称为标准错误输出（Standard Error），通常程序的运行结果打印到标准输出，而错误提示（例如 gcc 报的警告和错误）打印到标准错误输出，所以 fopen 的错误处理写成这样更符合惯例：

② 更准确地说是这样：在 Shell 下敲命令启动一个新进程，新进程会继承 Shell 进程已经打开的文件，所以新进程在启动时就已经打开了终端设备。

```
if ( (fp = fopen("/tmp/file1", "r")) == NULL) {
        fputs("Error open file /tmp/file1\n", stderr);
        exit(1);
}
```

fputs 函数稍后详细介绍。不管是打印到标准输出还是打印到标准错误输出效果是一样的，都是打印到终端设备（也就是屏幕），那为什么还要分成标准输出和标准错误输出呢？我们可以在命令行用重定向把标准输出和标准错误输出分开，例如：

```
$ ./a.out > errlog.txt
```

这样把标准输出重定向到一个常规文件，而标准错误输出仍然对应终端设备，就可以把正常的输出结果和错误提示分开，而不是混在一起打印到屏幕。

24.2.4 errno 与 perror/strerror 函数

很多系统函数错误返回时将错误原因记录在 libc 定义的全局变量 errno 中[③]，每种错误原因对应一个错误码，请查阅 errno(3)的 Man Page 了解各种错误码，errno 在头文件 errno.h 中声明，是一个整型变量，所有错误码都是正整数。

如果在程序中打印错误信息时直接打印 errno 的值，不能很直观地看出是什么错误。比较好的办法是用 perror 将 errno 解释成字符串再打印。

```
#include <stdio.h>

void perror(const char *s);
```

perror 函数将错误信息打印到标准错误输出，首先打印参数 s 所指的字符串，然后打印:号，然后根据当前 errno 的值打印错误原因。虽然 perror 函数要读取 errno 并解释成字符串，但使用 perror 函数不必包含 errno.h，只需包含声明 perror 函数的 stdio.h，因为我们讲过 C 标准库的头文件是相互独立的。例如：

例 24.4 perror

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
        FILE *fp = fopen("abcde", "r");
        if (fp == NULL) {
                perror("Open file abcde");
                exit(1);
        }
        return 0;
}
```

③ 其实这个说法并不准确，在多线程的程序中 errno 是一个宏定义，访问 errno 看起来像是访问一个全局变量，其实是每个线程各自访问各自的一块内存空间，互不干扰，请查阅参考文献[31]的 1.7 节。

如果文件 abcde 不存在，fopen 返回 NULL 并设置 errno 为 ENOENT（fopen(3)的 **ERRORS** 部分描述了这个函数可能产生的错误码），紧接着 perror 函数读取 errno 的值，将 ENOENT 解释成字符串 No such file or directory 并打印，最后打印的结果是 Open file abcde: No such file or directory。注意，如果把上面的程序改成这样：

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

int main(void)
{
        FILE *fp = fopen("abcde", "r");
        if (fp == NULL) {
                perror("Open file abcde");
                printf("errno: %d\n", errno);
                exit(1);
        }
        return 0;
}
```

则 printf 打印的错误号可能是 perror 产生的而不是 fopen 产生的。errno 是一个全局变量，很多系统函数都有可能改变它，errno(3)指出，调用一个系统函数时不管成功不成功都有可能改变 errno，但只有不成功时 errno 的值才是有意义的。所以即使 perror 调用是成功的，它也有可能改变 errno，后面 printf 打出来的 errno 的值可能就不再反映 fopen 的错误原因了。同样道理，把上面的代码改成先 printf 再 perror 也是错误的，perror 打印的错误原因可能是 printf 产生的而不是 fopen 产生的：

```
FILE *fp = fopen("abcde", "r");
if (fp == NULL) {
        printf("errno: %d\n", errno);
        perror("Open file abcde");
        exit(1);
}
```

所以一个系统函数错误返回后应该马上检查 errno，在检查 errno 之前不能再调用其他系统函数。如果需要在先后两个系统函数中检查同一个错误码，可以这样：

```
FILE *fp = fopen("abcde", "r");
if (fp == NULL) {
        int err = errno
        printf("errno: %d\n", errno);
        errno = err;
        perror("Open file abcde");
        exit(1);
}
```

strerror 函数可以根据错误号返回错误原因字符串。

```
#include <string.h>

char *strerror(int errnum);
返回值：错误码 errnum 所对应的字符串
```

这个函数返回指向静态内存的指针，用完之后不需要释放内存。strerror 函数有时候用起来比 perror 方便，可以自己控制输出格式，例如：

```
    fprintf(stderr, "%s, line %d: %s\n", __FILE__, __LINE__,
strerror(errno));
```

fprintf 函数类似于 printf，但可以输出到自己指定的文件，而不是固定打印到标准输出。另外，有时候错误码并不保存在 errno 中，例如 pthread 库函数的错误码都是通过返回值返回，不改变 errno，显然这种情况用 strerror 比 perror 方便。

习题

1．在系统头文件中找到各种错误码的宏定义。

2．做几个小练习，看看 fopen 出错有哪些常见的原因。

打开一个没有访问权限的文件。

```
    fp = fopen("/etc/shadow", "r");
    if (fp == NULL) {
            perror("Open /etc/shadow");
            exit(1);
    }
```

fopen 也可以打开一个目录，传给 fopen 的第一个参数目录名末尾可以加/也可以不加/，但只允许以只读方式打开。试试如果以可写的方式打开一个存在的目录会怎么样呢？

```
    fp = fopen("/home/akaedu/", "r+");
    if (fp == NULL) {
            perror("Open /home/akaedu");
            exit(1);
    }
```

请读者自己设计几个实验，看看你还能测试出哪些错误原因？

24.2.5 以字节为单位的 I/O 函数

fgetc 函数从指定的文件中读一个字节，getchar 从标准输入读一个字节，调用 getchar()相当于调用 fgetc(stdin)。

```
    #include <stdio.h>

    int fgetc(FILE *stream);
    int getchar(void);
    返回值：成功返回读到的字节，出错或者已经到达文件末尾则返回 EOF
```

对于 fgetc 函数有以下几点说明：

- 要用 fgetc 函数读一个文件，该文件的打开方式必须是可读的。
- 系统对于每个打开的文件都记录着当前读写位置（Position Indicator）。当文件打开时，读写位置在文件开头，每调用一次 fgetc，读写位置向后移动一个字节，因此可以连续多次调用 fgetc 函数依次读取多个字节。
- fgetc 成功时返回读到一个字节，本来应该是 unsigned char 型的，但由于函数原型中返回值是 int 型，所以这个字节要转换成 int 型再返回，那为什么要规定返回值是 int 型呢？因为 fgetc 有一个特殊返回值 EOF，如果调用 fgetc 出错，或者在调用 fgetc 时读写位置已经到达文件末尾，则返回 EOF，即-1，用 int 型表示-1 应该是 0xffffffff，如果读到字节 0xff，由 unsigned char 型转换成 int 型是 0x000000ff，只有规定返回值是 int 型才能把这两种情况区分开。如果规定返回值是 unsigned char 型，那么当返回值是 0xff 时无法区分到底是 EOF 还是字节 0xff。

  如果需要保存 fgetc 的返回值，一定要保存在 int 型变量中，如果写成 unsigned char c = fgetc(fp);，那么根据 c 的值也无法区分 EOF 和 0xff 字节。注意，fgetc 读到文件末尾时返回 EOF，只是用这个返回值表示已读到文件末尾，并不是说每个文件末尾都有一个特殊的字节是 EOF（根据上面的分析，EOF 并不是一个字节）。

fputc 函数向指定的文件中写一个字节，putchar 向标准输出写一个字节，调用 putchar(c)相当于调用 fputc(c, stdout)。

```
#include <stdio.h>

int fputc(int c, FILE *stream);
int putchar(int c);
返回值：成功返回写入的字节，出错返回 EOF
```

对于 fputc 函数也要说明几点：

- 要用 fputc 函数写一个文件，该文件的打开方式必须是可写的(包括追加)。
- 每调用一次 fputc，读写位置向后移动一个字节，因此可以连续多次调用 fputc 函数依次写入多个字节。如果文件是以追加方式打开的，每次调用 fputc 总是先把读写位置移到文件末尾然后把要写入的字节追加到后面。以后要介绍的 I/O 函数也都是这样更新读写位置的，不再赘述。

下面的例子演示了这四个函数的用法，从键盘读入一串字符写到一个文件中，再从这个文件中读出这些字符打印到屏幕上。

例 24.5 用 fputc/fgetc 读写文件和终端

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
        FILE *fp;
```

```
        int ch;

        if ( (fp = fopen("file2", "w+")) == NULL) {
                perror("Open file file2\n");
                exit(1);
        }
        while ( (ch = getchar()) != EOF)
                fputc(ch, fp);
        rewind(fp);
        while ( (ch = fgetc(fp)) != EOF)
                putchar(ch);
        fclose(fp);
        return 0;
}
```

从终端设备读有点特殊。当调用 getchar()或 fgetc(stdin)时，如果用户没有输入字符，getchar 函数就阻塞等待。所谓阻塞（Block）是指这个函数调用不返回，也就不能执行后面的代码，这个进程阻塞了，操作系统可以调度别的进程执行。从终端设备读还有一个特点，用户输入一般字符并不会使 getchar 函数返回，仍然阻塞着，只有当用户输入回车或者文件结束标志时 getchar 才返回[④]。这个程序的执行过程分析如下：

```
$ ./a.out
hello（输入 hello 并回车，这时第一次调用 getchar 返回，读取字符 h 存到文件中，然后连续调用 getchar 五次，读取 ello 和换行符存到文件中，第 7 次调用 getchar 又阻塞了）
hey（输入 hey 并回车，第 7 次调用 getchar 返回，读取字符 h 存到文件中，然后连续调用 getchar 三次，读取 ey 和换行符存到文件中，第 11 次调用 getchar 又阻塞了）
（这时输入 Ctrl-D，第 11 次调用 getchar 返回 EOF，跳出循环，进入下一个循环，回到文件开头，把文件内容一个字节一个字节读出来打印，直到文件结束）
hello
hey
```

从终端设备输入时有两种方法表示文件结束，一种方法是在某一行开头输入 Ctrl-D（如果光标不在一行开头则需要连续输入两次 Ctrl-D），另一种方法是利用 Shell 的 Heredoc 语法：

```
$ ./a.out <<END
> hello
> hey
> END
hello
hey
```

<<END 表示从下一行开始是标准输入，直到某一行开头出现 END 时结束。<<后面的结束符可以任意指定，不一定非得是 END，只要和输入的内容能区分开就行。

在上面的程序中，第一个 while 循环结束时 fp 所指的文件的读写位置在文件末尾，然后调用 rewind 函数把读写位置移到文件开头，再进入第二个 while 循环从头读

④ 这个特性取决于终端的工作模式，终端可以配置成一次输入一行的模式，也可以配置成一次输入一个字符的模式，默认是一次输入一行的模式（本书的实验都是在这种模式下做的），关于终端的配置请查阅参考文献[31]的第 18 章。

取文件内容并打印。

习题

1．虽然我说 getchar 函数会一直阻塞到用户输入回车才返回，但例 24.5 的代码和运行结果并没有提供证据支持我的说法，如果看成每敲一个键 getchar 就返回一次，也能解释程序的运行结果。请写一个小程序证明 getchar 确实是一直阻塞到用户输入回车才返回。

2．编写一个简单的文件复制程序。

```
$ ./mycp dir1/fileA dir2/fileB
```

运行这个命令可以把 dir1/fileA 文件拷贝成 dir2/fileB 文件。

```
$ ./mycp /home/akaedu/fileA dir2
```

如果 dir2 是一个目录名，运行这个命令可以把/home/akaedu/fileA 文件拷贝到 dir2 目录下，成为 dir2/fileA 文件。注意做好各种出错处理。

24.2.6　操作读写位置的函数

我们在例 24.5 中看到 rewind 函数可以把读写位置移到文件开头，本节介绍另外两个操作读写位置的函数，fseek 可以任意改变读写位置，ftell 可以返回当前读写位置。

```
#include <stdio.h>

int fseek(FILE *stream, long offset, int whence);
返回值：成功返回 0，出错返回-1 并设置 errno

long ftell(FILE *stream);
返回值：成功返回当前读写位置，出错返回-1 并设置 errno

void rewind(FILE *stream);
```

任意改变读写位置的操作称为 Seek，C 标准 I/O 库的 fseek、fsetpos 和 rewind 函数可以做 Seek 操作。fsetpos/fgetpos 函数本书不做详细介绍。fseek 的 whence 和 offset 参数共同决定了读写位置移到何处，whence 参数的含义如下：

SEEK_SET

　　从文件开头移动 offset 个字节

SEEK_CUR

　　从当前读写位置移动 offset 个字节

SEEK_END

　　从文件末尾移动 offset 个字节

offset 可正可负，负值表示向前（向文件开头的方向）移动，正值表示向后（向文件末尾的方向）移动。如果向前移动的字节数超过了文件开头则出错返回。如果向后移动的字节数超过了文件末尾，再次写入时将增大文件尺寸，从原来的文件末尾到fseek移动之后的读写位置之间的字节都是0，下面做个实验证实这一点。在第24.2.1节我们创建过一个文件textfile，其中有五个字节，“5678”加一个换行符，现在我们拿这个文件做实验。

例24.6 fseek

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
        FILE* fp;
        if ( (fp = fopen("textfile","r+")) == NULL) {
                perror("Open file textfile");
                exit(1);
        }
        if (fseek(fp, 10L, SEEK_SET) != 0) {
                perror("Seek file textfile");
                exit(1);
        }
        fputc('K', fp);
        fclose(fp);
        return 0;
}
```

运行这个程序，然后查看文件textfile的内容：

```
$ ./a.out
$ od -tx1 -tc -Ax textfile
000000 35 36 37 38 0a 00 00 00 00 00 4b
         5  6  7  8 \n \0 \0 \0 \0 \0  K
00000b
```

fseek(fp, 10L, SEEK_SET)将读写位置从文件开头（文件地址0的位置）向后移动10个字节，移到文件地址10的位置，然后在该位置写入一个字符K，这样文件就变长了，文件地址5～9的字节自动用0填充。

读写位置可以用一个long型的值表示，这个值可以调用ftell得到，也可以传给fseek，UNIX系统的文件模型比较简单，我们可以认为这个值等同于文件地址，但在其他操作系统上则不一定。我们讲过Windows的文本文件如果以文本模式打开，换行符在磁盘上存的是\r\n，而在程序中看到的却是\n，在磁盘上存的字节数和在程序中看到的不一致，其他操作系统的文件模型也有一些特殊规定，所以C标准关于fseek函数有很多奇怪的规定（至少在UNIX程序员看起来是很奇怪的），如果要编写可移植的代码就得考虑这些问题：

1．对于以文本模式打开的文件，whence参数只有取SEEK_SET是有意义的，并且传给offset参数的值要么是0，要么是先前对同一个文件调用ftell得到的返回值，不能像上面的例子那样任意指定一个10L。

2．对于以二进制模式打开的文件，fseek 函数有可能不支持 whence 参数取 SEEK_END 的情况。

最后还有一点要注意，常规文件都可以做 Seek 操作，而设备文件有很多是不支持 Seek 操作的，只允许顺序读写，比如对终端设备调用 fseek 会出错返回。

24.2.7　以字符串为单位的 I/O 函数

fgets 从指定的文件中读一行字符到调用者提供的缓冲区中，gets 从标准输入读一行字符到调用者提供的缓冲区中。

```
#include <stdio.h>

char *fgets(char *s, int size, FILE *stream);
char *gets(char *s);
返回值：成功时 s 指向哪返回的指针就指向哪，出错或者已经到达文件末尾则返回 NULL
```

gets 函数无需解释，Man Page 的 **BUGS** 部分已经说得很清楚了：Never use gets()。gets 函数的存在只是为了兼容以前的程序，我们写的代码都不应该调用这个函数。gets 函数的接口设计得很有问题，就像 strcpy 一样，用户提供缓冲区首地址，却不能指定缓冲区大小，很可能导致缓冲区溢出错误。这个函数比 strcpy 更加危险，strcpy 的输入和输出都来自程序内部，只要程序员小心一点就可以避免出问题，而 gets 读取的输入直接来自程序外部，用户可以输入任意长的字符串，程序员无法避免 gets 函数导致的缓冲区溢出错误，所以唯一的办法就是不用它。

现在说说 fgets 函数，参数 s 是缓冲区首地址，size 是缓冲区长度，该函数从 stream 所指的文件中读取以'\n'结尾的一行（包括'\n'在内）存到缓冲区 s 中，并且在该行末尾加上一个'\0'组成完整的字符串。

如果文件中的一行太长，fgets 从文件中读了 size-1 个字符还没读到'\n'，就把已经读到的 size-1 个字符再加上一个'\0'存入缓冲区并返回，文件中剩下的半行可以在下次调用 fgets 时继续读，如图 24.1 所示。

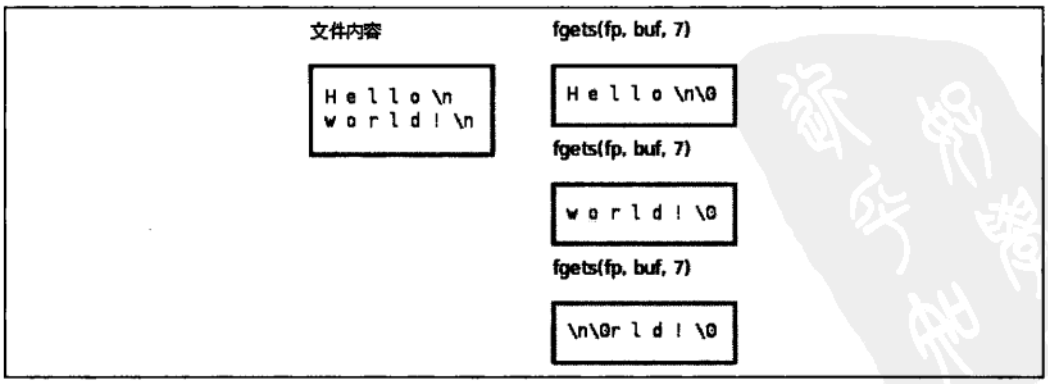


图 24.1　fgets 用法举例

如果 fgets 读到文件末尾的最后几个字符，不管是不是以'\n'结尾都加上一个'\0'存

入缓冲区并返回。下次再调用 fgets 时读写位置已经到达文件末尾，则返回 NULL。

注意，对于 fgets 来说，'\n'是一个特别的字符，而'\0'并无任何特别之处，如果读到'\0'就当做普通字符读入。如果文件中存在'\0'字符（或者说字节 0），调用 fgets 之后就无法判断缓冲区中的'\0'究竟是从文件读上来的还是由 fgets 自动添加的，所以 fgets 只适合读文本文件而不适合读二进制文件，并且文本文件中的所有字符都应该是可见字符，不能有'\0'。

注意在 Man Page 的函数原型中 FILE *指针参数通常起名叫 stream，因为标准 I/O 库操作的文件有另一个名称叫做流（Stream），现在简单介绍一下这个名词的历史背景。最早的操作系统并没有设备抽象机制，磁带驱动器、磁盘驱动器、行式打印机、打孔卡片等设备的操作方式各不相同，每种设备的打开方法都不一样，读写方法也不一样，有的设备中的数据按记录存储，每次只能读写一条记录，有的设备需要一边读写一边发各种控制命令，程序员需要记住所有这些细节。而 UNIX 系统向前迈进了一大步，把所有设备都抽象成“文件”的概念，文件由一串字节组成，用一个路径名标识，不管操作什么设备都用同样的函数打开，用同样的函数读写，并且每次可以读写任意的字节数。比如磁盘上的数据是按扇区（Sector）来组织的，每次操作磁盘驱动器可以读写一个扇区，但是 UNIX 系统屏蔽了这些底层细节，用户程序把磁盘文件看作数据流，每次可以读写任意的字节数，比如用 fgets 读一行，这一行可长可短，也可以跨扇区边界，如果用户程序调用一次 fgets 读三个扇区的数据，UNIX 系统底层做三次读操作来完成用户的一次 fgets 调用。

fputs 向指定的文件中写入一个字符串，puts 向标准输出写入一个字符串。

```
#include <stdio.h>

int fputs(const char *s, FILE *stream);
int puts(const char *s);
返回值：成功返回一个非负整数，出错返回 EOF
```

缓冲区 s 中保存的是以 Null 结尾的字符串，fputs 将该字符串写入文件 stream，但并不写入结尾的'\0'。与 fgets 不同的是，fputs 并不关心字符串中的'\n'字符，字符串中可以有'\n'也可以没有'\n'。puts 将字符串 s 写到标准输出（不包括结尾的'\0'），然后自动写一个'\n'到标准输出。

习题

1. 用 fgets/fputs 写一个拷贝文件的程序，根据本节对 fgets 函数的分析，这个程序应该只能拷贝文本文件，试试用它拷贝二进制文件会出什么问题。

24.2.8 以记录为单位的 I/O 函数

```
#include <stdio.h>

size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE
```

```
*stream);
```

返回值：读或写的记录数，成功时返回的记录数等于 nmemb，出错或读到文件末尾时返回的记录数小于 nmemb，也可能返回 0

fread 和 fwrite 用于读写记录，这里的记录是指一串固定长度的字节，比如一个 int、一个结构体或者一个定长数组。参数 size 指出每条记录的长度，而参数 nmemb 指出要读或写多少条这样的记录，这些记录在 ptr 所指的内存空间中连续存放，共占 size * nmemb 个字节。fread 从文件 stream 中读出 size * nmemb 个字节保存到 ptr 所指的内存空间，而 fwrite 把 ptr 所指的内存空间里的 size * nmemb 个字节写到文件 stream 中。

nmemb 是用户程序请求读或写的记录数，但系统不一定能完成这样的请求，fread/fwrite 返回的记录数是实际完成读写的记录数，有可能小于 nmemb。例如调用 fread 时指定 nmemb 为 2，而当前读写位置距文件末尾只有一条记录的长度，这种情况下只读一条记录，返回 1。如果当前读写位置已经到达文件末尾，则调用 fread 返回 0。如果在读写文件的过程中出错了，fread/fwrite 的返回值也可能小于 nmemb 指定的值，比如刚写完一条记录，在写下一条记录时出错了，则 fwrite 返回 1。下面的例子由两个程序组成，一个程序把结构体保存到文件中，另一个程序从文件中读出这个结构体。

例 24.7　fread/fwrite

```
/* writerec.c */
#include <stdio.h>
#include <stdlib.h>

struct record {
        char name[10];
        int age;
};

int main(void)
{
        struct record array[2] = {{"Ken", 24}, {"Knuth", 28}};
        FILE *fp = fopen("recfile", "w");
        if (fp == NULL) {
                perror("Open file recfile");
                exit(1);
        }
        fwrite(array, sizeof(struct record), 2, fp);
        fclose(fp);
        return 0;
}
/* readrec.c */
#include <stdio.h>
#include <stdlib.h>

struct record {
        char name[10];
        int age;
};

int main(void)
```

```
{
        struct record array[2];
        FILE *fp = fopen("recfile", "r");
        if (fp == NULL) {
                perror("Open file recfile");
                exit(1);
        }
        fread(array, sizeof(struct record), 2, fp);
        printf("Name1: %s\tAge1: %d\n", array[0].name, array
        [0].age);
        printf("Name2: %s\tAge2: %d\n", array[1].name, array
        [1].age);
        fclose(fp);
        return 0;
}
```

```
$ gcc writerec.c -o writerec
$ gcc readrec.c -o readrec
$ ./writerec
$ od -tx1 -tc -Ax recfile
000000 4b 65 6e 00 00 00 00 00 00 00 00 18 00 00 00
       K  e  n \0 \0 \0 \0 \0 \0 \0 \0 030 \0 \0 \0
000010 4b 6e 75 74 68 00 00 00 00 00 00 1c 00 00 00
       K  n  u  t  h \0 \0 \0 \0 \0 \0 034 \0 \0 \0
000020
$ ./readrec
Name1: Ken  Age1: 24
Name2: Knuth    Age2: 28
```

我们把一个 struct record 结构体看作一条记录，每条记录占 16 字节（有填充字节），把两条记录写到文件中共占 32 字节。该程序生成的 recfile 文件是二进制文件而非文本文件，因为其中不仅保存着字符型数据，还保存着整型数据 24 和 28（在 od 命令的输出中以八进制显示为 030 和 034）。注意，直接在文件中读写结构体的程序是不可移植的，如果在一个平台上编译运行 writebin.c 程序，把生成的 recfile 文件拷贝到另一个平台，然后在另一个平台上编译运行 readbin.c 程序，则不一定能正确读出文件内容，因为不同平台的大小端可能不同，结构体的填充方式也可能不同，只有当两个平台遵循相同的 ABI 时才能保证正确读出文件内容。

24.2.9 格式化 I/O 函数

现在该正式讲一下 printf 和 scanf 函数了，这两个函数都有很多种形式。

```
#include <stdio.h>

int printf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
int sprintf(char *str, const char *format, ...);
int snprintf(char *str, size_t size, const char *format, ...);

#include <stdarg.h>
#include <stdio.h>

int vprintf(const char *format, va_list ap);
int vfprintf(FILE *stream, const char *format, va_list ap);
int vsprintf(char *str, const char *format, va_list ap);
```

```
int vsnprintf(char *str, size_t size, const char *format, va_list ap);
```

返回值：成功返回格式化输出的字节数（不包括字符串结尾的'\0'），出错返回一个负值

printf 格式化打印到标准输出，而 fprintf 打印到指定的文件 stream 中。sprintf 并不打印到文件，而是打印到用户提供的缓冲区 str 并在末尾加'\0'，由于格式化后的字符串长度不容易估计，有可能造成缓冲区溢出，所以用 snprintf 更好一些，参数 size 指定了缓冲区长度，如果格式化后的字符串长度超过缓冲区长度，snprintf 就把字符串截断到 size-1 字节，再加上一个'\0'写入缓冲区，也就是说 snprintf 保证字符串以'\0'结尾。snprintf 的返回值是格式化后的字符串长度（不包括结尾的'\0'），如果字符串被截断，返回的是截断之前的长度，把它和缓冲区中字符串的实际长度做比较就可以知道是否发生了截断。

上面列出的后四个函数在函数名开头多了个字母 v，表示可变参数不是以...的形式传进来，而是以 va_list 类型传进来的。下面我们用 vsnprintf 包装出一个类似 printf 的函数，带有格式化字符串参数和可变参数。

例 24.8 格式化打印错误信息的 err_sys 函数

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <stdarg.h>
#include <string.h>

#define MAXLINE 80

void err_sys(const char *fmt, ...)
{
        int err = errno, n;
        char buf[MAXLINE+1];
        va_list ap;

        va_start(ap, fmt);

        vsnprintf(buf, MAXLINE, fmt, ap);
        n = strlen(buf);
        snprintf(buf+n, MAXLINE-n, ": %s", strerror(err));
        strcat(buf, "\n");
        fputs(buf, stderr);

        va_end(ap);
        exit(1);
}

int main(int argc, char *argv[])
{
        FILE *fp;
        if (argc != 2) {
                fputs("Usage: ./a.out pathname\n", stderr);
                exit(1);
        }
        fp = fopen(argv[1], "r");

        if (fp == NULL)
```

```
                err_sys("%s, line %d - open file %s", __FILE__,
                  __LINE__, argv[1]);
        printf("Open %s OK\n", argv[1]);
        fclose(fp);
        return 0;
    }
```

前面在讲 strerror 函数时举过一个例子，用 fprintf 格式化打印错误信息，打印之后再调 exit 函数退出，现在 err_sys 函数进一步简化了流程，打印错误信息和退出一步完成。为了演示 snprintf 的用法，我们限制打印错误信息的最大长度不超过 80 个字符（标准字符终端一行可以显示的字符数）。

我们在第 23.6 节讲过可变参数的原理，printf 并不知道实际参数的类型，只能按转换说明指出的参数类型从栈帧上取参数，如果实参类型和转换说明要求的类型不符，结果可能会有些意外。现在总结一下转换说明有哪些写法，每种写法要求取什么类型的参数。在这里只列举几种常用的格式，其他格式请参考 Man Page 和参考文献[8]的 7.19.6.1 节。每个转换说明以%号开头，以转换字符结尾，我们以前用过的转换说明仅包含%号和转换字符，例如%d、%s，其实在这两个字符中间还有一些可选项，如表 24.1 所示。

表 24.1 printf 转换说明的可选项

选项	描述	举例
#	八进制前面加 0（转换字符为 o），十六进制前面加 0x（转换字符为 x）或 0X（转换字符为 X）	printf("%#x", 0xff)打印 0xff，printf("%x", 0xff)打印 ff
-	格式化后的内容居左，右边可以留空格	见下面的例子
最小宽度	指定一个整数值，表示格式化后的字符串至少要有这么长，如果没有这么长可以在左边留空格（如果前面指定了-号就在右边留空格）。宽度有一种特别表示形式，不指定整数值而是写成一个*号，表示取一个 int 型参数作为最小宽度	printf("-%10s-", "hello")打印 -␣␣␣␣␣hello- ，printf("-%-*s-", 10, "hello")打印-hello␣␣␣␣␣-
.	用于分隔最小宽度和精度	见下面的例子
精度	指定一个整数值，对于字符串来说指定了格式化后保留的最大长度（转换字符是 s），对于浮点数来说指定了格式化后小数点右边的位数（转换字符是 e E f），对于整数来说指定了格式化后的最小位数（转换字符是 d i o u x X）。精度也可以不指定整数值而是写成一个*号，表示取一个 int 型参数作为精度	printf("%.4s", "hello")打印 hell，printf("-%6.4d-", 100)打印 -␣␣0100- ，printf("-%*.*f-", 8, 4, 3.14)打印-␣␣3.1400-
字长	转换字符 d i o u x X 默认是取 int 型的参数，在转换字符前面加 hh、h、l、ll 可以改变字长，分别表示取 char、short、long、long long 型的字长，转换字符 d 和 i 把参数看作有符号数，转换字符 o u x X 把参数看作无符号数[a]。转换字符 e E f 默认是取 double 型的参数，在转换字符前面加 L 可以改变字长，表示取 long double 型的参数	printf("%hhu", -1)打印 255

[a] 我们知道传给 printf 的可变参数要做 Integer Promotion，所以指定字长为 hh 或 h 的整型参数其实是这样处理的：首先取 int 型参数，然后转换成有符号或无符号的 char 型或 short 型再打印输出。

常用的转换字符如表 24.2 所示。

表 24.2　printf 的转换字符

转换字符	描述	举例
d i	取 int 型参数格式化成有符号十进制表示，如果格式化后的位数小于指定的精度，就在左边补 0	printf("%.4d", 100) 打印 0100
o u x X	取 unsigned int 型参数格式化成无符号八进制（o）、十进制（u）、十六进制（x 或 X）表示，x 表示十六进制数字用小写 abcdef，X 表示十六进制数字用大写 ABCDEF，如果格式化后的位数小于指定的精度，就在左边补 0	printf("%#X", 0xdeadbeef) 打印 0XDEADBEEF
c	取 int 型参数转换成 unsigned char 型，格式化成对应的 ASCII 码字符	printf("%c", 256+'A')打印 A
s	取 const char *型参数所指的字符串格式化输出，遇到'\0'结束，或者达到指定的最大长度（精度）结束	printf("%.4s", "hello")打印 hell
p	取 void *型参数格式化输出，输出格式是 Implementation-defined。在 Linux 平台上以十六进制输出指针所指的地址，相当于%#x（ILP32）或%#lx（LP64）	printf("%p", main) 打印 main 函数的首地址 0x80483c4
f	取 double 型参数格式化成[-]ddd.ddd 这样的格式，小数点后的默认精度是 6 位	printf("%f", 3.14) 打印 3.140000，printf("%f", 0.00000314)打印 0.000003
e E	取 double 型参数格式化成[-]d.ddde±dd（转换字符是 e）或[-]d.dddE±dd（转换字符是 E）这样的格式，小数点后的默认精度是 6 位，指数至少是两位	printf("%e", 3.14) 打印 3.140000e+00
g G	取 double 型参数格式化，输出结果可能按%f 格式化，也可能按%e（转换字符是 g）或%E（转换字符是 G）格式化，取决于参数值用哪种格式输出比较合适（具体规则请查阅参考文献[8]的 7.19.6.1 节条款 8）。输出结果的小数部分末尾的 0 会去掉，如果小数部分全是 0，则连同小数点一起去掉	printf("%g", 3.00)打印 3，printf("%g", 0.00001234567)打印 1.23457e-05
%	打印一个%号	printf("%%")打印一个%号

下面看 scanf 函数的几种形式。

```
#include <stdio.h>

int scanf(const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
int sscanf(const char *str, const char *format, ...);

#include <stdarg.h>
#include <stdio.h>

int vscanf(const char *format, va_list ap);
```

```
int vsscanf(const char *str, const char *format, va_list ap);
int vfscanf(FILE *stream, const char *format, va_list ap);
```

返回值：返回成功匹配和赋值的参数个数，返回 0 表示一个参数都没有被匹配赋值，在完成第一个输入转换或遇到匹配失败之前已经到达输入末尾则返回 EOF，出错返回 EOF 并设置 errno

scanf 从标准输入读字符，按格式化字符串 format 中的转换说明解释这些字符，转换后赋给后面的赋值参数，赋值参数都是传出参数，因此必须传地址而不能传值。fscanf 从指定的文件 stream 中读字符，而 sscanf 从指定的字符串 str 中读字符。后面三个以 v 开头的函数的可变参数不是以...的形式传进来，而是以 va_list 类型传进来。

现在总结一下 scanf 的格式化字符串和转换说明，这里也只列举几种常用的格式，其他格式请参考 Man Page 和参考文献[8]的 7.19.6.2 节。scanf 用格式化字符串去匹配输入的字符，每次成功匹配一个转换说明就做一次输入转换，并给一个参数赋值，如果遇到匹配失败，或者格式化字符串都匹配完了，或者遇到输入末尾[5]就停止。如果遇到匹配失败而停止，文件的读写位置指向输入中第一个不匹配的字符，下次调用输入函数（如 fgetc、fgets、fscanf）读文件时可以从这个位置继续。

格式化字符串中包括：

- 空白字符，指 C 语言定义的空格、Tab、\r、\n、\v、\f 六个字符，格式化字符串中一个或多个连续的空白字符匹配输入中 0 个或多个连续的空白字符，一直匹配到输入中下一个非空白字符为止。
- 普通字符，和输入中相同的字符一一匹配。
- 转换说明，以%开头，以转换字符结尾，中间有一些可选项。转换说明从输入序列中的下一个非空白字符开始匹配[6]，匹配输入中格式相符的子串，直到遇到不匹配的字符时停止，或者达到指定的最大宽度时停止（稍后解释最大宽度）。

转换说明中的可选项有：

- *号，表示这个转换说明只用来匹配一段输入字符，也算一次成功的输入转换，但转换结果并不赋给参数，这次匹配也不计入返回值。
- 最大宽度，指定一个整数值 N，表示这个转换说明最多匹配 N 个输入字符。
- 对于整型的赋值参数可以指定字长，有 hh、h、l、ll 几种，含义和 printf 相同。对于浮点型的赋值参数也可以指定字长，有 l 和 L 两种，浮点型赋值参数的类型默认是 float *而非 double *，这一点跟 printf 的规定不同（请思考一下为什么会不同），前面加 l 表示 double *型，前面加 L 表示 long double *型。

常用的转换字符如表 24.3 所示。

⑤ 对于 scanf 和 fscanf 是指文件末尾，对于 sscanf 是指输入字符串的末尾。

⑥ %c 例外，并不跳过开头的空白字符，稍后详细说明。

表 24.3　scanf 的转换字符

转换字符	描述
d	匹配十进制整数（开头可以有正负号），赋值参数的类型是 int *
i	匹配整数（开头可以有正负号），赋值参数的类型是 int *，如果输入的数字部分以 0x 或 0X 开头则匹配十六进制整数，如果以 0 开头则匹配八进制整数，否则匹配十进制整数
o u x	分别匹配八进制、十进制、十六进制整数（开头可以有正负号），赋值参数的类型是 unsigned int *，注意输入是有符号数，要先转换成无符号数再给参数赋值
s	匹配一串非空白字符，从输入中的第一个非空白字符开始匹配到下一个空白字符之前，或者匹配到最大宽度，赋值参数的类型是 char *，在赋值的末尾自动添一个'\0'
c	匹配一串字符（可以包含空白字符），字符的个数由最大宽度指定，缺省宽度是 1，赋值参数的类型是 char *，不会在赋值的末尾添加'\0'。如果输入的开头有空白字符，这些空白字符并不会跳过，而是算在%c 匹配的字符之中，要想跳过开头的空白字符，可以在格式化字符串中%c 前面用一个空格去匹配
e f g	匹配浮点数（开头可以有正负号），赋值参数的类型是 float *
%	转换说明%%匹配输入中的一个%号，不做赋值

下面几个例子出自参考文献[3]。第一个例子，读取用户输入的浮点数累加起来。

例 24.9　用 scanf 实现简单的计算器

```
#include <stdio.h>

int main(void)  /* rudimentary calculator */
{
        double sum, v;

        sum = 0;
        while (scanf("%lf", &v) == 1)
                printf("\t%.2f\n", sum += v);
        return 0;
}
```

运行结果如下：

```
$ ./a.out
12.2（回车）
        12.20
1（回车）
        13.20
3e-2（回车）
        13.23
（输入 Ctrl-D）
```

从终端设备读的函数都要阻塞等待用户输入，直到敲回车才返回，换行符也是空白字符，每次循环的 scanf("%lf", &v)匹配到换行符之前，下次循环的 scanf("%lf", &v)从换行符之后的第一个非空白字符开始匹配。

初学者常犯的一个错误是从这个例子类推得到这样的代码：

```
char c;
while (scanf("%c", &c) == 1)
        printf("\t%c\n", c);
```

注意%c是可以匹配空白字符的，想想结果会是什么样。

第二个例子是从日期字符串中分离出年月日，如果我们要读取25 Dec 1988这样的日期格式，可以这样写：

```
char *str = "25 Dec 1988";
int day, year;
char monthname[20];

sscanf(str, "%d %s %d", &day, monthname, &year);
```

读者可以做一个小练习，在str和格式化字符串中添加或删除一些空白字符，看还能不能正确匹配。如果要读取12/25/1998这样的日期格式，就需要在格式化字符串中用/斜线匹配输入字符中的/斜线：

```
int day, month, year;

scanf("%d/%d/%d", &month, &day, &year);
```

如果我们的程序需要读取标准输入识别以上两种日期格式，比较好的办法是先用fgets读到一个缓冲区中，然后交给sscanf处理，用一种格式匹配不上可以再试另一种格式：

```
while (fgets(line, sizeof(line), stdin) > 0) {
        if (sscanf(line, "%d %s %d", &day, monthname, &year) == 3)
                printf("valid: %s\n", line); /* 25 Dec 1988 form */
        else if (sscanf(line, "%d/%d/%d", &month, &day, &year) == 3)
                printf("valid: %s\n", line); /* mm/dd/yy form */
            else
                printf("invalid: %s\n", line); /* invalid form */
}
```

习题

1. 在第10章举过一个例子，用scanf读入一个字符串，然后自己写代码把它转换成整数，但那个程序存在很多问题，几经修改，直到那一章结束也没有给出圆满的解决办法，现在你有办法了吗？你能给出几种解决办法？

2. 下面这段代码有什么问题？这也是初学者常犯的错误。

```
int i;
scanf("%d\n", &amp;i);
```

24.2.10　C 标准库的 I/O 缓冲区

用户程序调用 C 标准 I/O 库函数读写文件或设备，而这些库函数要通过系统调用把读写请求传给内核，最终由内核驱动磁盘或设备完成 I/O 操作。fopen 要通过 open(2)系统调用打开文件，fgetc/fgets/fread/fscanf 等函数要通过 read(2)系统调用请求内核读设备，fputc/fputs/fwrite/fprintf 等函数要通过 write(2)系统调用请求内核写设备，fclose 要通过 close(2)系统调用关闭文件。

C 标准库为每个打开的文件分配一个用户空间 I/O 缓冲区以加速读写操作，库函数内部通过 FILE 结构体的成员可以访问到这个缓冲区，用户程序调用读写函数大多数时候都在 I/O 缓冲区中读写，只有少数时候需要把读写请求传给内核。

以 fgetc/fputc 为例，当用户程序第一次调用 fgetc 读一个字节时，fgetc 函数可能通过 read(2)系统调用进入内核读 1K 字节到 I/O 缓冲区，然后返回 I/O 缓冲区中的第一个字节给用户，把读写位置指向 I/O 缓冲区中的第二个字节，以后用户程序再调 fgetc 就直接从 I/O 缓冲区中读取，而不需要进内核了，当用户程序把这 1K 字节都读完之后，再次调用 fgetc 时，fgetc 函数会再次进入内核读 1K 字节到 I/O 缓冲区。在这个场景中，用户程序、C 标准库和内核之间的关系就像第 16.5 节讲过的 CPU、Cache 和内存之间的关系一样，C 标准库之所以会从内核预读一些数据到 I/O 缓冲区，是希望用户程序稍后要用到这些数据，如果能直接从用户空间 I/O 缓冲区读数据，就省去了系统调用和模式切换的开销，效率要高得多。

另一方面，用户程序调用 fputc 通常只是写到 I/O 缓冲区中，这样 fputc 函数可以很快地返回，如果 I/O 缓冲区写满了，fputc 就通过 write(2)系统调用把 I/O 缓冲区中的数据传给内核，内核最终把数据写回磁盘。如果某个时刻用户程序希望把 I/O 缓冲区中的数据立刻写回内核（这称为 Flush 操作），而不是等 I/O 缓冲区写满了再写回内核，可以调用库函数 fflush。fclose 函数在关闭文件之前也会自动做 Flush 操作。

```
#include <stdio.h>

int fflush(FILE *stream);
返回值：成功返回 0，出错返回 EOF 并设置 errno
```

作为一个特例，调用 fflush(NULL)可以对当前进程所有打开的文件做 Flush 操作。注意，fflush 只保证通过 write(2)系统调用将数据写回内核，并不保证数据一定写到了设备上，通常内核里也会有一个 I/O 缓存，如果一定要求内核把数据写到设备上可以调用 fsync(2)，请查阅参考文献[31]的 3.13 节。

图 24.2 以 fgets/fputs 为例说明 I/O 缓冲区的作用，使用 fgets/fputs 函数时在用户程序中也需要提供缓冲区（图中的 buf1 和 buf2），请注意区分用户程序的缓冲区和 C 标准库的 I/O 缓冲区。

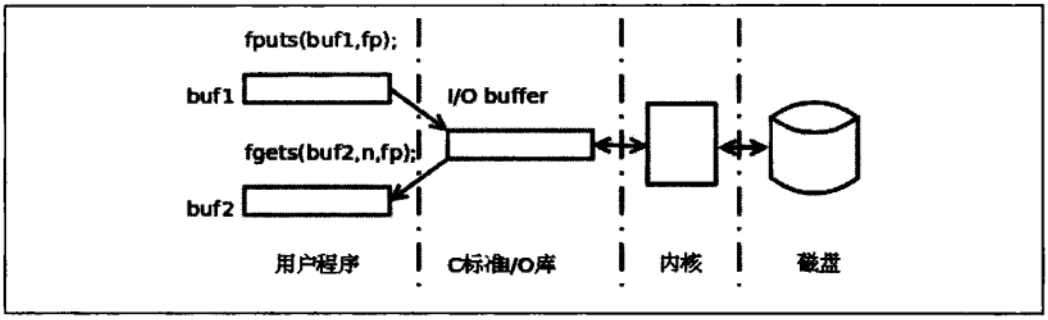


图 24.2 C 标准库的 I/O 缓冲区

C 标准库的 I/O 缓冲区有三种类型：全缓冲、行缓冲和无缓冲。当用户程序调用库函数做写操作时，不同类型的缓冲区具有不同的特性。

全缓冲

如果缓冲区写满了就写回内核，常规文件通常是全缓冲的。

行缓冲

如果用户程序写的数据中有换行符就把这一行写回内核，或者如果缓冲区写满了就写回内核。标准输入和标准输出对应的终端设备通常是行缓冲的。

无缓冲

用户程序每次调库函数做写操作都要立刻写回内核。标准错误输出通常是无缓冲的，这样用户程序产生的错误信息可以尽快输出到设备。

下面通过一个简单的例子证明标准输出对应的终端设备是行缓冲的。

```
#include <stdio.h>

int main(void)
{
        printf("hello world");
        while(1);
        return 0;
}
```

运行这个程序，会发现 hello world 并没有打印出来。用 Ctrl-C 终止它，去掉程序中的 while(1);语句再试一次：

```
$ ./a.out
hello world$
```

hello world 被打印到屏幕上，后面直接跟 Shell 提示符而没有换行。

我们知道 main 函数被启动代码这样调用：exit(main(argc, argv));。main 函数 return 时启动代码会调用 exit，exit 函数首先关闭所有尚未关闭的 FILE *指针（关闭之

前要自动做 Flush 操作），然后通过_exit 系统调用进入内核退出当前进程[7]。

在上面的例子中，由于标准输出是行缓冲的，printf("hello world");打印的字符串没有换行符，所以只把字符串写到标准输出的 I/O 缓冲区而没有写回内核，如果敲 Ctrl-C，进程是异常终止的，并没有调用 exit，也就没有机会做 Flush 操作，因此字符串最终没有打印到屏幕上。如果把打印语句改成 printf("hello world\n");，有换行符，就会立刻写回内核。或者如果把 while(1);去掉也可以写回内核，因为从 main 函数 return 相当于调 exit。在本书的其他例子中，printf 打印的字符串末尾都有换行符，以保证字符串在 printf 调用结束时就写回内核，如果你用 printf 打印调试信息，保证这一点尤其重要。我们修改程序验证一下：

```
#include <stdio.h>
#include <unistd.h>

int main(void)
{
        printf("hello world");
        _exit(0);
}
```

直接调用_exit 退出，字符串不会打印出来，如果把_exit 调用改成 exit 就可以打印了。

除了写满缓冲区和写入换行符之外，还有两种特殊情况会导致行缓冲的文件被 Flush。如果：

- 用户程序调用库函数从某个无缓冲的文件中读取
- 或者从某个行缓冲的文件中读取，并且这次读操作会引发 read(2)系统调用从内核读取数据

那么在读取之前会自动 Flush 所有打开的行缓冲文件。例如：

```
#include <stdio.h>
#include <unistd.h>

int main(void)
{
        char buf[20];
        printf("Please input a line: ");
        fgets(buf, 20, stdin);
        return 0;
}
```

虽然调用 printf 并不会写回内核，但紧接着调用 fgets 读一个行缓冲的文件（标准输入），在读取之前会自动 Flush 所有行缓冲，包括标准输出，所以字符串会打印出来。我们再试试 fflush 函数：

⑦ 其实在调_exit 进内核之前还有其他工作要处理，用户程序中通过 atexit(3)注册的退出处理函数正是在这个时候被调用，请查阅参考文献[31]的 7.3 节。

```
#include <stdio.h>

int main(void)
{
        printf("hello world");
        fflush(stdout);
        while(1);
}
```

虽然字符串中没有换行，但用户程序调用 fflush 强制写回内核，所以字符串也能打印出来。下面用常规文件做一个实验。常规文件是全缓冲的，即使写了\n 也不会触发 Flush 操作，要写满缓冲区才会 Flush。

```
#include <stdio.h>

int main(void)
{
        FILE *fp = fopen("newfile", "w");
        fputs("hello world\n", fp);
        while (1);
}
```

在一个终端窗口中运行这个程序，同时打开另一个终端窗口查看这个文件：

```
$ ls -l newfile
-rw-r--r-- 1 akaedu akaedu 0 2010-07-14 22:28 newfile
```

可以看到文件被创建了，但是长度为 0 字节，这时如果在另一个进程中读这个文件是读不到任何数据的。

关于 I/O 缓冲区还有一点要注意，在图 24.2 中我们看到，如果一个文件以可读可写的方式打开（fopen 的 mode 参数中包含+号），读和写用的是同一个 I/O 缓冲区，I/O 缓冲区在某个时刻可以用作读缓冲，在另一个时刻可以用作写缓冲，但不能同时支持读写操作，因此 C 标准对于这种文件的读写操作有一些特别规定。

1．写操作后面不能紧跟着读操作，在写操作和读操作之间应该做一次 Flush，使 I/O 缓冲的数据写回内核，这样 I/O 缓冲区可以重新利用做读缓冲。要做 Flush 操作，除了直接调用 fflush 之外还可以调用 fseek、fsetpos 或 rewind，Seek 操作会使 I/O 缓冲区中的数据无效，因此在 Seek 操作之前会自动做 Flush。

2．读操作后面不能紧跟着写操作，在读操作和写操作之间应该调用 fseek、fsetpos 或 rewind 做一次 Seek 操作，声明 I/O 缓冲区中的数据无效，这样 I/O 缓冲区才可以重新利用做写缓冲，注意在读操作之后不能调用 fflush（C 标准规定这种情况是 Undefined）。

3．上一条规则有一个例外，如果读操作遇到了文件末尾，后面允许紧跟着写操作。

24.2.11 本节综合练习

1. 编程读写一个文件 test.txt，每隔 1 秒向文件中写入一行记录，类似于这样：

```
1 2010-7-30 15:16:42
2 2010-7-30 15:16:43
```

该程序应该无限循环，直到按 Ctrl-C 终止。下次再启动程序应该在 test.txt 文件末尾追加记录，并且序号能够接续上次的序号，比如：

```
1 2010-7-30 15:16:42
2 2010-7-30 15:16:43
3 2010-7-30 15:19:02
4 2010-7-30 15:19:03
5 2010-7-30 15:19:04
```

这类似于很多系统服务维护的日志文件，例如在我的机器上系统服务进程 acpid 维护一个日志文件/var/log/acpid，就像这样：

```
$ cat /var/log/acpid
[Sun Oct 26 08:44:46 2008] logfile reopened
[Sun Oct 26 10:11:53 2008] exiting
[Sun Oct 26 18:54:39 2008] starting up
...
```

acpid 进程以追加方式打开这个文件，每当有事件发生就追加一条记录，包括事件发生的时刻以及事件描述信息。

获取当前的系统时间需要调用 time(2)函数，在第 8.3 节讲过这个函数，返回值是 time_t 类型（其实是一种整型），然后调用 localtime(3)将 time_t 表示的时间转换成 struct tm 类型，该类型的各数据成员分别表示年月日时分秒，具体用法请查阅 Man Page。调用 sleep(3)函数可以指定程序睡眠多少秒，这也是一种阻塞调用。

2. ini 文件是一种很常见的配置文件，很多 Windows 程序都采用这种格式的配置文件，在 Linux 系统中 Qt 程序通常也采用这种格式的配置文件。比如：

```
;Configuration of http
[http]
domain=www.mysite.com
port=8080
cgihome=/cgi-bin

;Configuration of db
[database]
server = mysql
user = myname
password = toopendatabase
```

一个配置文件由若干个 Section 组成，由[]括号括起来的是 Section 名。每个 Section 下面有若干个“key = value”形式的键值对，等号两边可以有零个或多个空白字符（空格或 Tab），每个键值对占一行。以;号开头的行是注释。相邻的 Section

之间有一个或多个空行分隔，空行是仅包含零个或多个空白字符（空格或 Tab）的行。

现在 xml 兴起了，ini 文件显得有点土。现在要求编程把 ini 文件转换成 xml 文件。上面的例子经转换后应该变成这样：

```
<!-- Configuration of http -->
<http>
<domain>www.mysite.com</domain>
<port>8080</port>
<cgihome>/cgi-bin</cgihome>
</http>

<!-- Configuration of db -->
<database>
<server>mysql</server>
<user>myname</user>
<password>toopendatabase</password>
</database>
```

3. 实现类似 gcc 的-M 选项的功能，给定一个.c 文件，列出它直接或间接包含的所有头文件，例如有一个 main.c 文件：

```
#include <errno.h>
#include "stack.h"

int main(void)
{
        return 0;
}
```

你的程序读取这个文件，打印出其中包含的所有头文件的绝对路径：

```
$ ./a.out main.c
/usr/include/errno.h
/usr/include/features.h
...
/home/akaedu/stack.h: cannot find
```

如果有的头文件找不到，就像上面那样打印/home/akaedu/stack.h: cannot find。首先复习一下第 19.2.2 节讲过的头文件查找顺序，本题目不必考虑-I 选项指定的目录，只在.c 文件所在的目录和系统目录/usr/include 下查找。提示：可以采用深度优先搜索的策略。

24.3 数值字符串转换函数

```
#include <stdlib.h>

int atoi(const char *nptr);
double atof(const char *nptr);
返回值：转换结果
```

atoi 把一个字符串开头可以识别成十进制整数的部分转换成 int 型，相当于

strtol(nptr, (char **) NULL, 10);。例如 atoi("123abc")的返回值是 123，字符串开头可以有若干空格，例如 atoi("␣-90.6-")的返回值是-90。如果字符串开头没有可识别的整数，例如 atoi("asdf")，则返回 0，而 atoi("0***")也返回 0，根据返回值并不能区分这两种情况，所以使用 atoi 函数不能检查出错的情况。下面要讲的 strtol 函数可以检查出错的情况，在严格的场合下应该用 strtol，而 atoi 用起来更简便，所以也很常用。

atof 把一个字符串开头可以识别成浮点数的部分转换成 double 型，相当于 strtod(nptr, (char **) NULL);。字符串开头可以识别的浮点数格式和 C 语言的浮点数常量格式相同，例如 atof("31.4 ")的返回值是 31.4，atof("3.14e+1AB")的返回值也是 31.4。atof 也不能检查出错的情况，而 strtod 函数可以检查。

```
#include <stdlib.h>

long int strtol(const char *nptr, char **endptr, int base);
double strtod(const char *nptr, char **endptr);
返回值：转换结果，出错时设置 errno
```

strtol 是 atoi 的增强版，主要体现在以下几方面：

- 不仅可以识别十进制整数，还可以识别其他进制的整数，取决于 base 参数，比如 strtol("0XDEADbeE～～", NULL, 16)返回 0xdeadbee 的值，strtol ("0777～～", NULL, 8)返回 0777 的值。
- endptr 是一个传出参数，函数返回时指向 nptr 中未被识别的第一个字符。例如 char *pos; strtol("123abc", &pos, 10);，strtol 返回 123，pos 指向字符串"123abc"中的字母 a。如果字符串开头没有可识别的整数，例如 char *pos; strtol("ABCabc", &pos, 10);，则 strtol 返回 0，pos 指向字符串开头，可以根据 pos 的传出值判断这种出错情况，而这是 atoi 处理不了的。
- 如果字符串中的整数值超出 long int 的表示范围（上溢或下溢），则 strtol 返回它所能表示的最大（或最小）整数，并设置 errno 为 ERANGE，例如 strtol ("0XDEADbeef～～", NULL, 16)返回 0x7fffffff 并设置 errno 为 ERANGE。

回想一下使用 fopen 的套路：

```
if ( (fp = fopen(...)) == NULL) {
        读取 errno
}
```

fopen 在出错时会返回 NULL，因此我们知道需要读 errno，但 strtol 在出错时可能返回 0x7fffffff，在成功调用时也可能返回 0x7fffffff，我们如何知道需要读 errno 呢？最严谨的做法是首先把 errno 置 0，再调用 strtol，再查看 errno 是否变成了错误码。Man Page 上有一个很好的例子：

例 24.10 strtol 的出错处理

```
#include <stdlib.h>
#include <limits.h>
```

```
#include <stdio.h>
#include <errno.h>

int main(int argc, char *argv[])
{
        int base;
        char *endptr, *str;
        long val;

        if (argc <.2) {
                fprintf(stderr, "Usage: %s str [base]\n", argv[0]);
                exit(EXIT_FAILURE);
    }

        str = argv[1];
        base = (argc > 2) ? atoi(argv[2]) : 10;

        errno = 0;    /* To distinguish success/failure after call */
        val = strtol(str, &endptr, base);

        /* Check for various possible errors */

        if ((errno == ERANGE && (val == LONG_MAX || val == LONG_MIN))
            || (errno != 0 && val == 0)) {
            perror("strtol");
            exit(EXIT_FAILURE);
        }

        if (endptr == str) {
                fprintf(stderr, "No digits were found\n");
                exit(EXIT_FAILURE);
        }

        /* If we got here, strtol() successfully parsed a number */

        printf("strtol() returned %ld\n", val);

        if (*endptr != '\0')        /* Not necessarily an error... */
                printf("Further characters after number: %s\n", endptr);

        exit(EXIT_SUCCESS);
}
```

strtod 是 atof 的增强版，增强的功能和 strtol 类似，本书不做详细介绍。

24.4 分配内存的函数

除了 malloc 之外，C 标准库还提供了另外两个在堆空间分配内存的函数，它们分配的内存同样由 free 释放。

```
#include <stdlib.h>

void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);
返回值：成功返回所分配内存空间的首地址，出错返回 NULL
```

calloc 的参数很像 fread/fwrite 的参数，分配 nmemb 个元素的内存空间，每个元素占 size 字节，并且 calloc 负责把这块内存空间用字节 0 填充，而 malloc 并不负责把分配的内存空间清零。

有时候用 malloc 或 calloc 分配的内存空间使用了一段时间之后需要改变它的大小，一种办法是调用 malloc 分配一块新的内存空间，把原内存空间中的数据拷到新的内存空间，然后调用 free 释放原内存空间。使用 realloc 函数简化了这些步骤，把原内存空间的指针 ptr 传给 realloc，通过参数 size 指定新的大小（字节数），realloc 返回新内存空间的首地址，并释放原内存空间。新内存空间中的数据尽量和原来保持一致，如果 size 比原来小，则前 size 个字节不变，后面的数据被截断，如果 size 比原来大，则原来的数据全部保留，后面长出来的一块内存空间未初始化（realloc 不负责清零）。注意，参数 ptr 要么是 NULL，要么必须是先前调用 malloc、calloc 或 realloc 返回的指针，不能把任意指针传给 realloc 要求重新分配内存空间。作为两个特例，如果调用 realloc(NULL, size)，则相当于调用 malloc(size)，如果调用 realloc(ptr, 0)，ptr 不是 NULL，则相当于调用 free(ptr)。

```
#include <alloca.h>

void *alloca(size_t size);
返回值：返回所分配内存空间的首地址，如果 size 太大导致栈空间耗尽，结果是未定义的
```

参数 size 是请求分配的字节数，alloca 函数不是在堆上分配空间，而是在调用者函数的栈帧上分配空间，类似于 C99 的变长数组，当调用者函数返回时自动释放栈帧，所以不需要 free。很多 UNIX 系统都提供了这个函数，但它既不属于 C 标准也不属于 POSIX 标准。

第 25 章

链表、二叉树和哈希表

在第 12 章讲过数据结构是数据的组织方式，包含存储方式和访问方式两层意思。用指针组织的数据存储方式比数组更加灵活，相应的访问方式也会有新的特性。本章介绍链表、二叉树和哈希表三种数据结构，它们和数组[①]一样可以表示一组相同类型元素的集合，一样可以支持添加、删除、查找等访问操作，那么在实际应用中要表示一个集合到底该选用哪种数据结构呢？应该看哪种数据结构的存储方式和访问方式最符合你的应用需要。

25.1 链表

25.1.1 单链表

图 22.6 所示的链表即单链表（Single Linked List），本节我们学习如何创建和操作这种链表。每个链表有一个头指针，通过头指针可以找到第一个节点，每个节点都可以通过指针域找到它的后继，最后一个节点的指针域为 NULL，表示没有后继。数组在内存中是连续存放的，而链表在内存中的布局是不规则的，我们知道要访问某个数组元素可以通过“基地址+n×每个元素的字节数”得到它地址，或者说数组支持随机访问，而链表是不支持随机访问的，只能通过前一个节点的指针域得知后一个节点的地址，因此只能从头指针开始顺序访问各节点。以下代码实现了单链表的基本操作。

例 25.1 单链表

```
/* linkedlist.h */
#ifndef LINKEDLIST_H
#define LINKEDLIST_H

typedef struct node *link;
struct node {
        unsigned char elem;
        link next;
};
```

① 本章所说的数组是指一块连续的内存空间，可以是数组类型的全局变量或局部变量，也可以是用 malloc 分配一块内存然后把它看成数组。

```
link make_node(unsigned char elem);
void free_node(link p);
link search(unsigned char elem);
void insert(link p);
void delete(link p);
void traverse(void (*visit)(link));
void destroy(void);
void push(link p);
link pop(void);

#endif
/* linkedlist.c */
#include <stdlib.h>
#include "linkedlist.h"

static link head = NULL;

link make_node(unsigned char elem)
{
        link p = malloc(sizeof *p);
        p->elem = elem;
        p->next = NULL;
        return p;
}

void free_node(link p)
{
        free(p);
}

link search(unsigned char elem)
{
        link p;
        for (p = head; p; p = p->next)
                if (p->elem == elem)
                        return p;
        return NULL;
}

void insert(link p)
{
        p->next = head;
        head = p;
}

void delete(link p)
{
        link pre;
        if (p == head) {
                head = p->next;
                return;
        }
        for (pre = head; pre; pre = pre->next)
                if (pre->next == p) {
                        pre->next = p->next;
                        return;
                }
}

void traverse(void (*visit)(link))
```

```
{
        link p;
        for (p = head; p; p = p->next)
                visit(p);
}

void destroy(void)
{
        link q, p = head;
        head = NULL;
        while (p) {
                q = p;
                p = p->next;
                free_node(q);
        }
}

void push(link p)
{
        insert(p);
}

link pop(void)
{
        if (head == NULL)
                return NULL;
        else {
                link p = head;
                head = head->next;
                return p;
        }
}
/* main.c */
#include <stdio.h>
#include "linkedlist.h"

void print_elem(link p)
{
        printf("%d\n", p->elem);
}

int main(void)
{
        link p;

        insert(make_node(10));
        insert(make_node(5));
        insert(make_node(90));
        p = search(5);
        delete(p);
        free_node(p);
        traverse(print_elem);
        destroy();
        push(make_node(100));
        push(make_node(200));
        push(make_node(250));
        while (p = pop()) {
                print_elem(p);
                free_node(p);
        }
```

```
        return 0;
}
```

在初始化时把头指针 head 初始化为 NULL，表示空链表。然后 main 函数调用 make_node 创建几个节点，分别调用 insert 插入到链表中。

```
void insert(link p)
{
        p->next = head;
        head = p;
}
```

如图 25.1 所示，insert 函数虽然简单，其中也隐含了一种特殊情况（Special Case）的处理，当 head 为 NULL 时，执行 insert 操作插入第一个节点之后，head 指向第一个节点，而第一个节点的 next 指针域成为 NULL，这很合理，因为它也是最后一个节点。所以空链表虽然是一种特殊情况，却不需要特殊的代码来处理，和一般情况用同样的代码处理即可，这样写出来的代码更简洁，但是在读代码时要想到可能存在的特殊情况。

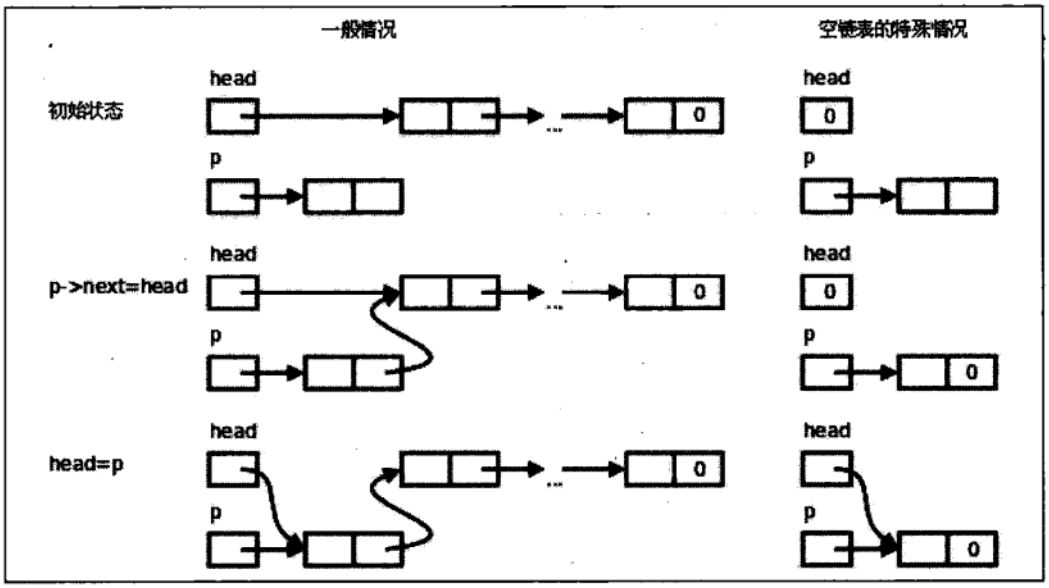


图 25.1　链表的插入操作

当然，insert 函数传进来的参数 p 也可能有特殊情况，传进来的 p 可能是 NULL，甚至是野指针，本章的函数代码都假定调用者传进来的参数是合法的，不对参数做特别检查。事实上，对指针参数做检查是不现实的，如果传进来的是 NULL 还可以检查一下，如果传进来的是野指针，根本无法检查它指向的内存单元是不是合法的，C 标准库函数通常也不对指针参数做检查，例如 strcpy（p, NULL）就会引起段错误。

接下来 main 函数调用 search 在链表中查找某个节点，如果找到就返回指向该节点的指针，找不到就返回 NULL。

```
link search(unsigned char elem)
```

```
{
        link p;
        for (p = head; p; p = p->next)
                if (p->elem == elem)
                        return p;
        return NULL;
}
```

search 函数其实也隐含了对于空链表这种特殊情况的处理，如果是空链表则循环体一次都不执行，直接返回 NULL。

然后 main 函数调用 delete 从链表中摘除用 search 找到的节点，再调用 free_node 释放该节点的存储空间。

```
void delete(link p)
{
        link pre;
        if (p == head) {
                head = p->next;
                return;
        }
        for (pre = head; pre; pre = pre->next)
                if (pre->next == p) {
                        pre->next = p->next;
                        return;
                }
}
```

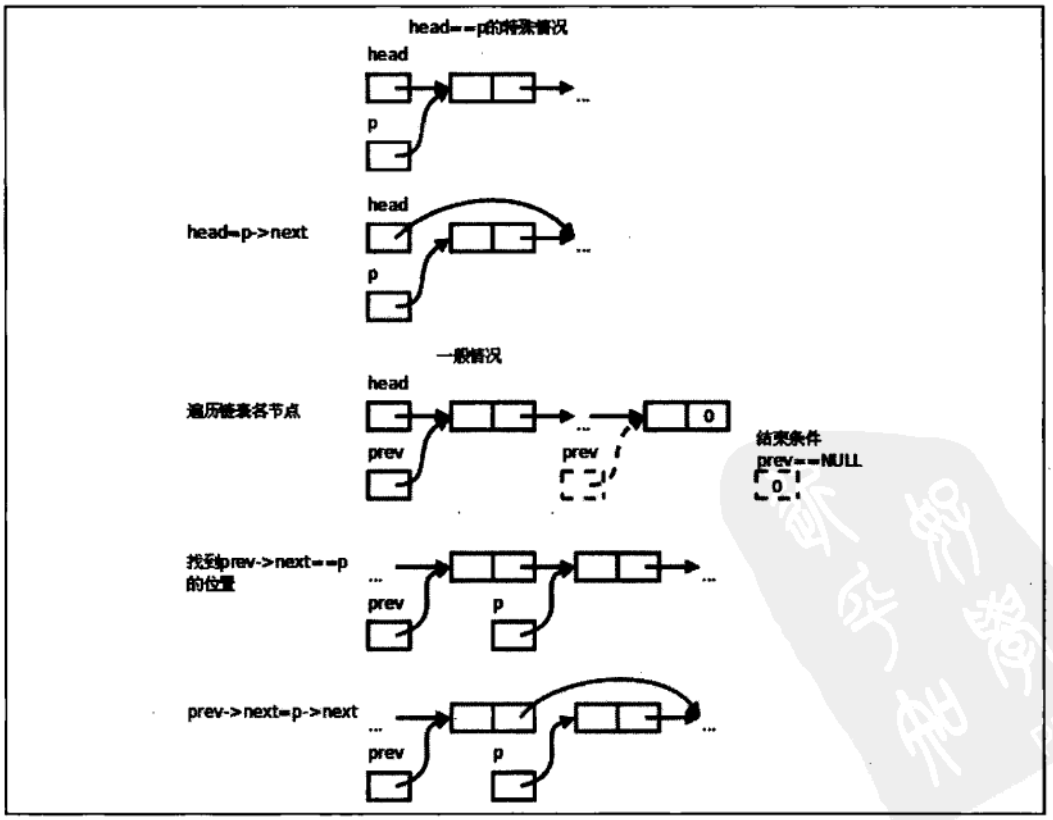


图 25.2 链表的删除操作

从图 25.2 可以看出，要摘除一个节点需要首先找到它的前趋然后才能做摘除操作，而在单链表中通过某个节点只能找到它的后继而不能找到它的前趋，所以删除操作要麻烦一些，需要从第一个节点开始依次查找要摘除的节点的前趋。delete 操作也要处理一种特殊情况，如果要摘除的节点是链表的第一个节点，它是没有前趋的，这种情况要用特殊的代码处理，而不能和一般情况用同样的代码处理，因为 p == head 的情况处理代码是 head = p->next;，而 pre->next == p 的情况处理代码是 pre->next = p->next;，处理代码不一样，不能合并。如果想办法把处理代码变成一样的就可以合并了，结合图 25.3 看下面的解法：

```
void delete(link p)
{
        link *pnext;
        for (pnext = &head; *pnext; pnext = &(*pnext)->next)
                if (*pnext == p) {
                        *pnext = p->next;
                        return;
                }
}
```

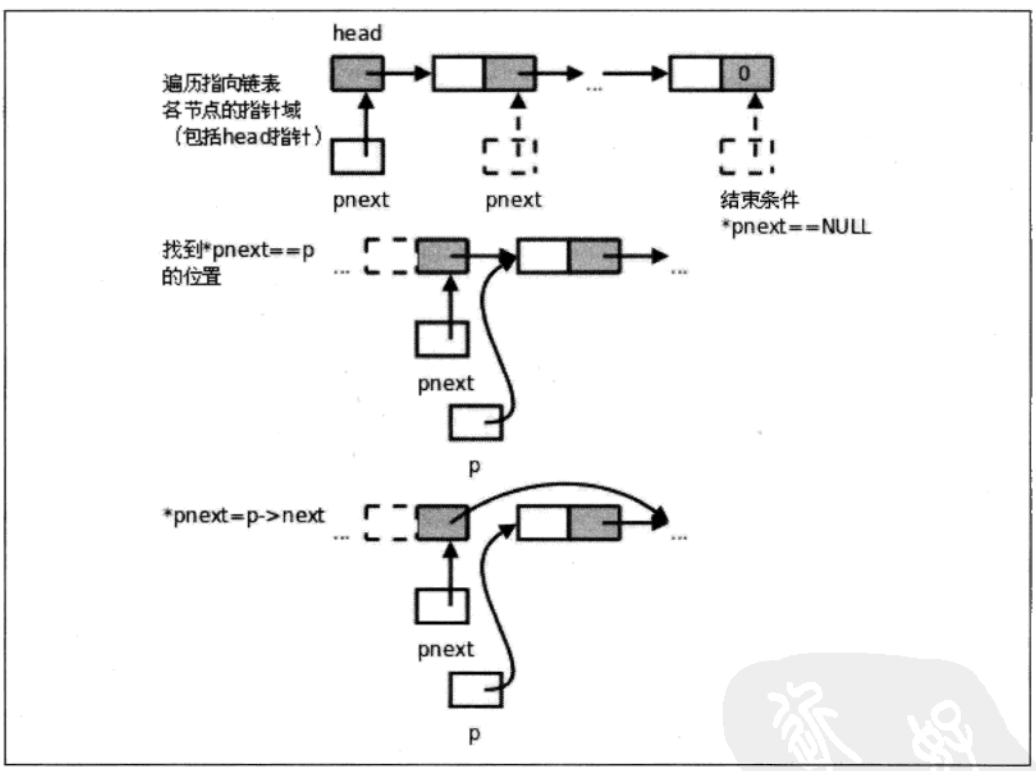


图 25.3　消除特殊情况的链表删除操作

定义一个指向指针的指针 pnext，在 for 循环中 pnext 遍历 head 指针和每个节点的 next 指针域，这样无论是 head 指针还是每个节点的 next 指针域都可以用*pnext 访问，就把处理代码统一起来了。

然后 main 函数调用 traverse 函数遍历整个链表，调用 destroy 函数销毁整个链表，请读者自己阅读这两个函数的代码。destroy 函数依次释放链表中的每个节点，想

一想能不能用 traverse（free）代替 destory 函数？

如果限定每次只允许在链表头部插入和删除元素，就形成一个 LIFO 的访问序列，所以在链表头部插入和删除元素的操作实现了堆栈的 Push 和 Pop 操作，main 函数的最后几步把链表当成堆栈来操作，从打印结果可以看到出栈顺序和入栈是相反的。

习题

1．比较数组和链表各自的优缺点，想想应该从哪些方面做比较？

2．修改 insert 函数实现插入排序的功能，链表中的元素按从小到大排列，每次插入新的元素都要在链表中找到合适的位置再插入。在第 11.6 节讲过，如果数组元素是有序排列的，可以用折半查找算法更快地找到其中某个元素，想一想如果链表中的元素是有序排列的，是否适用折半查找算法？为什么？

3．基于单链表实现队列的 enqueue 和 dequeue 操作。在链表的末尾再维护一个指针 tail，在 tail 处 enqueue，在 head 处 dequeue。想一想能不能反过来，在 head 处 enqueue 而在 tail 处 dequeue？

4．实现函数 void reverse(void);将单链表反转，如图 25.4 所示。要求尽可能把特殊情况转化成一般情况处理。

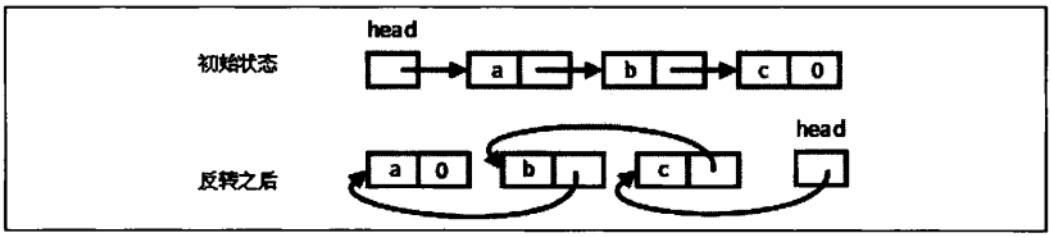


图 25.4 单链表的反转

25.1.2 双向链表

链表的 delete 操作需要首先找到要摘除的节点的前趋，而在单链表中找某个节点的前趋需要从表头开始依次查找，对于 n 个节点的链表，删除操作的时间复杂度是 O(n)。可以想象得到，如果每个节点再维护一个指向前趋的指针，删除操作就像插入操作一样容易了，时间复杂度是 O(1)，这样的链表称为双向链表（Doubly Linked List）。要实现双向链表只需在上一节代码的基础上改动两个地方。

在 linkedlist.h 中修改链表节点的结构体定义：

```
struct node {
        unsigned char elem;
        link prev, next;
};
```

在 linkedlist.c 中修改 insert 和 delete 函数：

```
void insert(link p)
{
        p->next = head;
        if (head)
                head->prev = p;
        head = p;
        p->prev = NULL;
}

void delete(link p)
{
        if (p->prev)
                p->prev->next = p->next;
        else
                head = p->next;
        if (p->next)
                p->next->prev = p->prev;
}
```

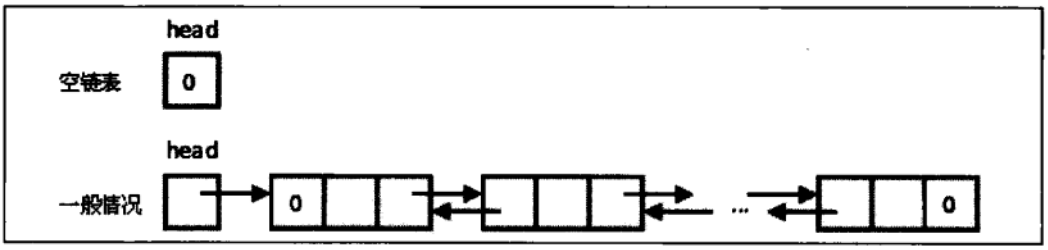


图 25.5　双向链表

注意图 25.5 中每个节点的 prev 指针应该指向前一个节点开头，但为了美观我都画成指向前一个节点末尾，现在对本章所画的图做以下约定：一个指针箭头可以指向一个结构体方框的任何位置，都表示指向这个结构体的首地址。

由于引入了 prev 指针，insert 和 delete 函数中都有一些特殊情况需要用特殊代码处理，不能和一般情况用同样的代码处理，这非常不爽，如果在表头和表尾各添加一个 Sentinel 节点（这两个节点只用于界定表头和表尾，不保存元素），就可以把这些特殊情况都转化为一般情况了。

例 25.2　带 Sentinel 的双向链表

```
/* doublylinkedlist.h */
#ifndef DOUBLYLINKEDLIST_H
#define DOUBLYLINKEDLIST_H

typedef struct node *link;
struct node {
        unsigned char elem;
        link prev, next;
};

link make_node(unsigned char elem);
void free_node(link p);
link search(unsigned char elem);
```

```
void insert(link p);
void delete(link p);
void traverse(void (*visit)(link));
void destroy(void);
void enqueue(link p);
link dequeue(void);

#endif
/* doublylinkedlist.c */
#include <stdlib.h>
#include "doublylinkedlist.h"

struct node tailsentinel;
struct node headsentinel = {0, NULL, &tailsentinel};
struct node tailsentinel = {0, &headsentinel, NULL};

static link head = &headsentinel;
static link tail = &tailsentinel;

link make_node(unsigned char elem)
{
        link p = malloc(sizeof *p);
        p->elem = elem;
        p->prev = p->next = NULL;
        return p;
}

void free_node(link p)
{
        free(p);
}

link search(unsigned char elem)
{
        link p;
        for (p = head->next; p != tail; p = p->next)
                if (p->elem == elem)
                        return p;
        return NULL;
}

void insert(link p)
{
        p->next = head->next;
        head->next->prev = p;
        head->next = p;
        p->prev = head;
}

void delete(link p)
{
        p->prev->next = p->next;
        p->next->prev = p->prev;
}

void traverse(void (*visit)(link))
{
        link p;
        for (p = head->next; p != tail; p = p->next)
                visit(p);
```

```
}

void destroy(void)
{
        link q, p = head->next;
        head->next = tail;
        tail->prev = head;
        while (p != tail) {
                q = p;
                p = p->next;
                free_node(q);
        }
}

void enqueue(link p)
{
        insert(p);
}

link dequeue(void)
{
        if (tail->prev == head)
                return NULL;
        else {
                link p = tail->prev;
                delete(p);
                return p;
        }
}
/* main.c */
#include <stdio.h>
#include "doublylinkedlist.h"

void print_elem(link p)
{
    printf("%d\n", p->elem);
}

int main(void)
{
        link p;
        insert(make_node(10));
        insert(make_node(5));
        insert(make_node(90));
        p = search(5);
        delete(p);
        free_node(p);
        traverse(print_elem);
        destroy();
        enqueue(make_node(100));
        enqueue(make_node(200));
        enqueue(make_node(250));
        while (p = dequeue()) {
                print_elem(p);
                free_node(p);
        }
        return 0;
}
```

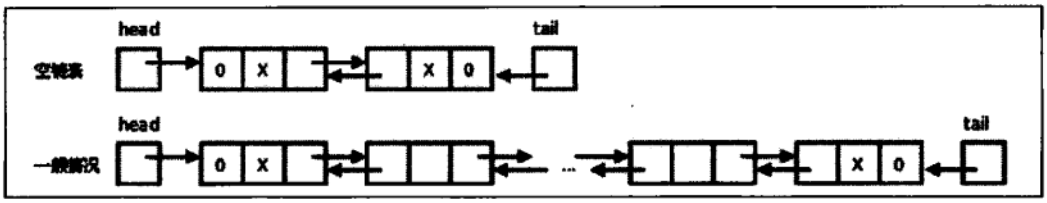


图 25.6 带 Sentinel 的双向链表

初始化空链表时两个 Sentinel 节点互相指向对方：

```
struct node tailsentinel;
struct node headsentinel = {0, NULL, &tailsentinel};
struct node tailsentinel = {0, &headsentinel, NULL};
```

第一行是个 Tentative Definition，第一行声明的变量名 tailsentinel 在第二行的 Initializer 中被引用，第二行声明的变量名 headsentinel 又在第三行的 Initializer 中被引用。

上面的例子还实现了队列的 enqueue 和 dequeue 操作，在 head 处 enqueue 而在 tail 处 dequeue。现在结合第 12.5 节想一想，其实用链表实现环形队列是最自然的，以前基于数组实现环形队列，我们还要“假想”它是首尾相接的，而如果基于链表实现环形队列，根本不需要假想，它就是用指针串起来首尾相接的。把上面的程序改成环形链表（Circular Linked List）也非常简单，只需要把 doublylinkedlist.c 中的：

```
struct node tailsentinel;
struct node headsentinel = {0, NULL, &tailsentinel};
struct node tailsentinel = {0, &headsentinel, NULL};

static link head = &headsentinel;
static link tail = &tailsentinel;
```

改成：

```
struct node sentinel = {0, &sentinel, &sentinel};
static link head = &sentinel;
```

再把 doublylinkedlist.c 中所有的 tail 替换成 head 即可，相当于把 head 和 tail 合二为一了，如图 25.7 所示。

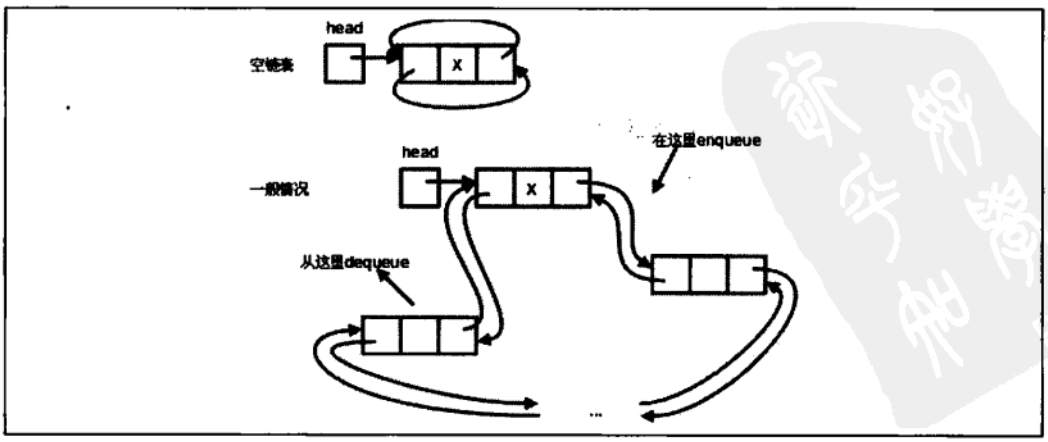


图 25.7 环形链表

25.1.3　静态链表

现在回想一下在例 12.4 中使用的数据结构，我把图重新画在下面，如图 25.8 所示。

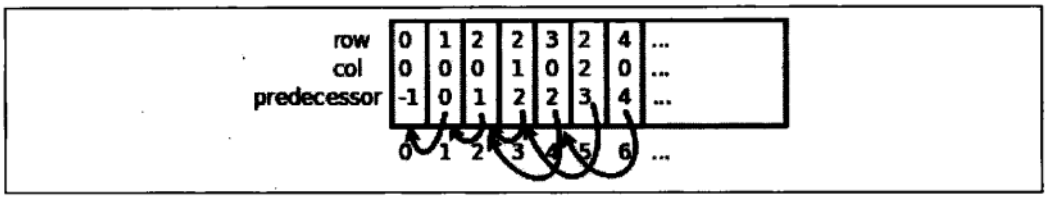


图 25.8　广度优先搜索的队列数据结构

这是一个静态分配的数组，每个数组元素都有 row、col 和 predecessor 三个成员，predecessor 成员保存一个数组下标，指向数组中的另一个元素，这其实也是链表的一种形式，称为静态链表，比如图 25.8 中的第 6、4、2、1、0 个元素串成一条链表。

25.1.4　本节综合练习

1．Josephus 是公元 1 世纪的著名历史学家，相传在一次战役中他和另外几个人被围困在山洞里，他们宁死不屈，决定站成一圈，每次数到三个人就杀一个，直到全部死光为止。Josephus 和他的一个朋友不想死，于是串通好了站在适当的位置上，最后只剩下他们俩的时候这个游戏就停止了。如果一开始的人数为 N，每次数到 M 个人就杀一个，那么要想不死应该站在什么位置呢？这个问题比较复杂，参考文献[30]的 1.3 节专门研究了 Josephus 问题的解，有兴趣的读者可以参考。现在我们做个比较简单的练习，用链表模拟 Josephus 他们玩的这个游戏，N 和 M 作为命令行参数传入，每个人的编号依次是 1～N，打印每次被杀者的编号，打印最后一个幸存者的编号。

2．在第 24.2.11 节的习题 1 中规定了一种日志文件的格式，每行是一条记录，由行号、日期、时间三个字段组成，由于记录是按时间先后顺序写入的，可以看作所有记录按日期排序，对于日期相同的记录再按时间排序。现在要求从这样一个日志文件中读出所有记录组成一个链表，链表中的记录首先按时间排序，对于时间相同的记录再按日期排序，最后写回文件中。比如原文件的内容是：

```
1 2010-7-30 15:16:42
2 2010-7-30 15:16:43
3 2010-7-31 15:16:41
4 2010-7-31 15:16:42
5 2010-7-31 15:16:43
6 2010-7-31 15:16:44
```

重新排序输出的文件内容是：

```
1 2010-7-31 15:16:41
2 2010-7-30 15:16:42
3 2010-7-31 15:16:42
4 2010-7-30 15:16:43
```

```
5 2010-7-31 15:16:43
6 2010-7-31 15:16:44
```

25.2 二叉树

25.2.1 二叉树的基本概念

链表的每个节点可以有一个后继，而二叉树（Binary Tree）的每个节点可以有两个后继。比如这样定义二叉树的节点：

```
typedef struct node *link;
struct node {
    unsigned char elem;
    link l, r;
};
```

这样的节点可以组织成如图 25.9 所示的各种形态。

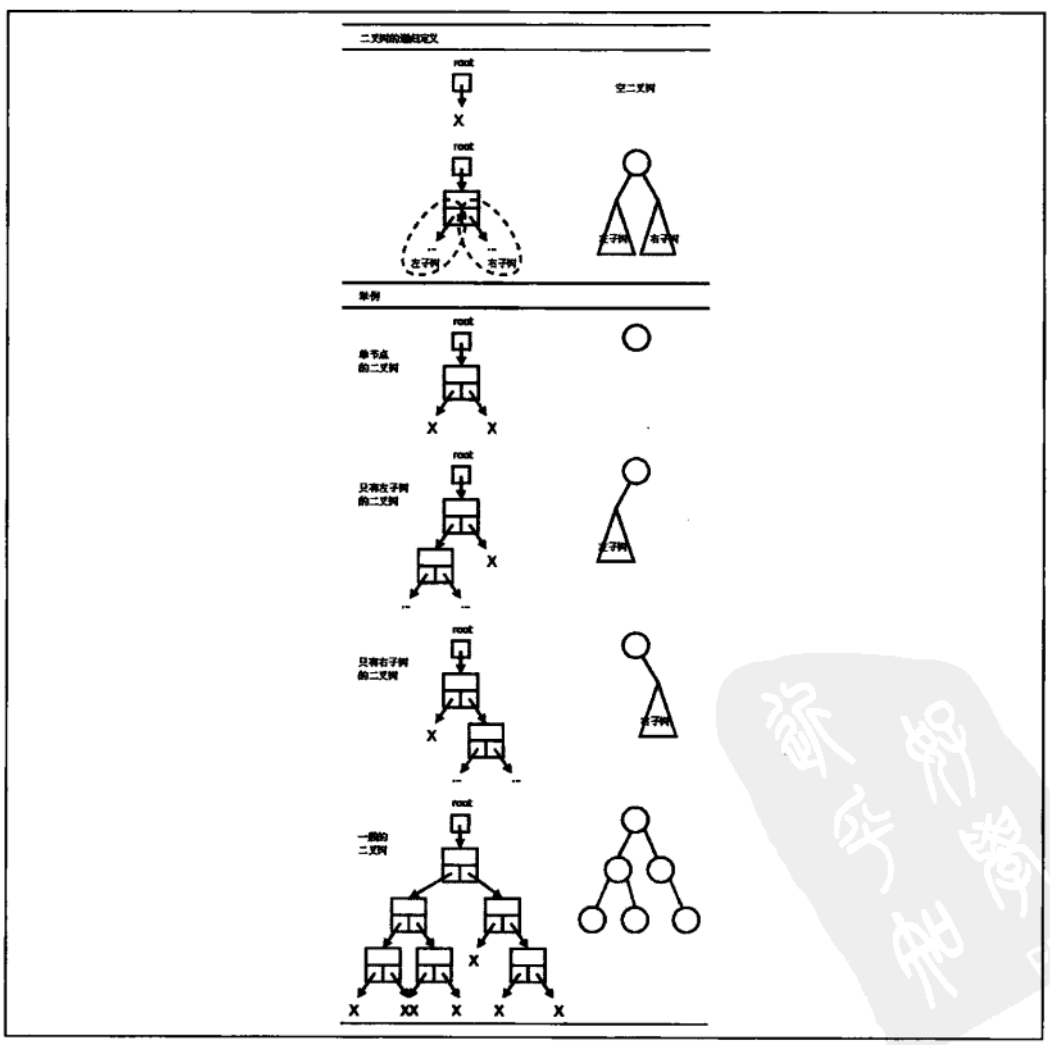


图 25.9 二叉树的定义和举例

二叉树可以这样递归地定义：

1．就像链表有头指针一样，每个二叉树有一个根指针（图 25.9 中的 root 指针）。根指针可以是 NULL，表示空二叉树。

2．根指针可以指向一个节点，称为根节点，根节点除了保存元素之外还有两个指针域，这两个指针域又分别是另外两个二叉树（左子树和右子树）的根指针。

图 25.9 举例示意了几种情况。

- 单节点的二叉树：左子树和右子树都是空二叉树。
- 只有左子树的二叉树：右子树是空二叉树。
- 只有右子树的二叉树：左子树是空二叉树。
- 一般的二叉树：左右子树都不为空。注意图 25.9 中由圈和线段组成的简化图示，以后我们都采用这种简化图示法，在圈中标上该节点所保存的元素的值。

链表的遍历方法是显而易见的：从前到后遍历即可。而二叉树是一种树状结构，如何做到把所有节点都走一遍不重不漏呢？可以采用深度优先策略的遍历算法，比如前序遍历（Pre-order Traversal）、中序遍历（In-order Traversal）和后序遍历（Post-order Traversal）。也可以采用广度优先策略的遍历算法，比如层序遍历（Level-order Traversal）。我们结合图 25.10 的例子来理解。

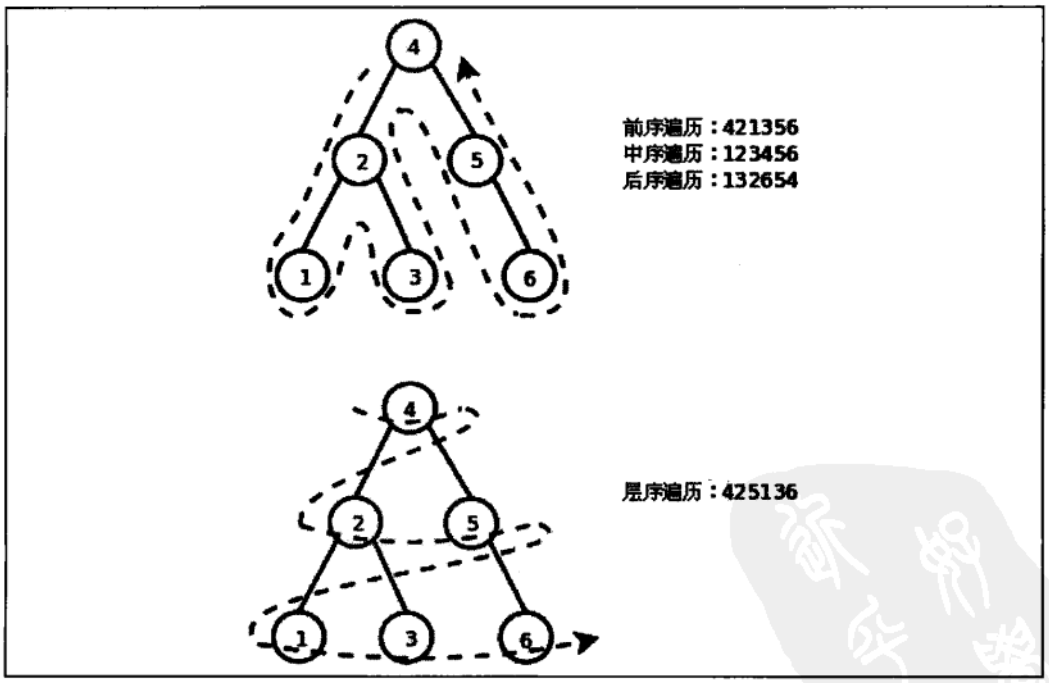


图 25.10　二叉树的遍历

前序遍历

1）访问根节点（图 25.10 标出了访问二叉树各节点的顺序，称为遍历序列）2）递归遍历左子树 3）递归遍历右子树

中序遍历

1）递归遍历左子树 2）访问根节点 3）递归遍历右子树

后序遍历

1）递归遍历左子树 2）递归遍历右子树 3）访问根节点

层序遍历

根节点是第一层，根节点所指向的节点是第二层，第二层节点所指向的节点是第三层，一层一层遍历下去

在例 12.3 中每次走过一个点都要标记这个点已走过，而树状结构比迷宫的网状结构简单，在遍历过程中不需要标记已经走过的点，想一想为什么。前序和中序遍历序列合在一起可以唯一确定二叉树的形态，或者说根据遍历序列可以构造出二叉树，构造过程如图 25.11 所示。

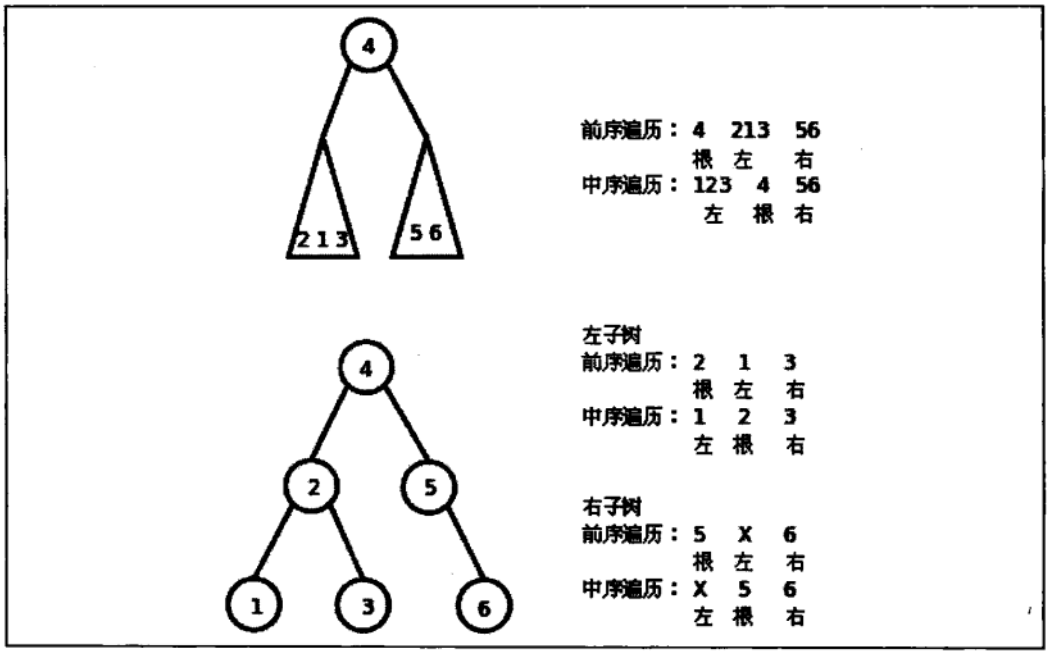


图 25.11 根据前序和中序遍历序列构造二叉树

1．前序遍历的第一个元素是 4，它应该是根节点。

2．在中序遍历序列中，4 左边的 1、2、3 应该是左子树的节点，4 右边的 5、6 应该是右子树的节点。

3．左子树的前序遍历序列是 213，中序遍历序列是 123，右子树的前序遍历序列是 56，中序遍历序列是 56，可以根据本算法递归地构造左子树和右子树。

想一想，根据中序和后序遍历序列能否构造二叉树？根据前序和后序遍历序列能否构造二叉树？下面的例子实现了二叉树的基本操作。

例 25.3 二叉树的基本操作

```
/* binarytree.h */
#ifndef BINARYTREE_H
#define BINARYTREE_H

typedef struct node *link;
struct node {
        unsigned char elem;
        link l, r;
};

link init(unsigned char VLR[], unsigned char LVR[], int n);
void pre_order(link t, void (*visit)(link));
void in_order(link t, void (*visit)(link));
void post_order(link t, void (*visit)(link));
int count(link t);
int depth(link t);
void destroy(link t);

#endif
/* binarytree.c */
#include <stdlib.h>
#include "binarytree.h"

static link make_node(unsigned char elem)
{
        link p = malloc(sizeof *p);
        p->elem = elem;
        p->l = p->r = NULL;
        return p;
}

static void free_node(link p)
{
        free(p);
}

link init(unsigned char VLR[], unsigned char LVR[], int n)
{
        link t;
        int k;
        if (n <= 0)
                return NULL;
        for (k = 0; VLR[0] != LVR[k]; k++);
        t = make_node(VLR[0]);
        t->l = init(VLR+1, LVR, k);
        t->r = init(VLR+1+k, LVR+1+k, n-k-1);
        return t;
}

void pre_order(link t, void (*visit)(link))
{
        if (!t)
                return;
        visit(t);
        pre_order(t->l, visit);
        pre_order(t->r, visit);
}
```

```
void in_order(link t, void (*visit)(link))
{
        if (!t)
                return;
        in_order(t->l, visit);
        visit(t);
        in_order(t->r, visit);
}

void post_order(link t, void (*visit)(link))
{
        if (!t)
                return;
        post_order(t->l, visit);
        post_order(t->r, visit);
        visit(t);
}

int count(link t)
{
        if (!t)
                return 0;
        return 1 + count(t->l) + count(t->r);
}

int depth(link t)
{
        int dl, dr;
        if (!t)
                return 0;
        dl = depth(t->l);
        dr = depth(t->r);
        return 1 + (dl > dr ? dl : dr);
}

void destroy(link t)
{
        post_order(t, free_node);
}
/* main.c */
#include <stdio.h>
#include "binarytree.h"

void print_elem(link p)
{
        printf("%d", p->elem);
}

int main(void)
{
        unsigned char pre_seq[] = { 4, 2, 1, 3, 6, 5, 7 };
        unsigned char in_seq[] = { 1, 2, 3, 4, 5, 6, 7 };
        link root = init(pre_seq, in_seq, 7);
        pre_order(root, print_elem);
        putchar('\n');
        in_order(root, print_elem);
        putchar('\n');
        post_order(root, print_elem);
        putchar('\n');
```

```
        printf("count=%d depth=%d\n", count(root), depth(root));
        destroy(root);
        return 0;
}
```

习题

1. 本节描述了二叉树的递归定义，想一想单链表的递归定义应该怎么表述？请仿照本节的例子用递归实现单链表的基本操作：

```
link init(unsigned char elements[], int n);
void pre_order(link t, void (*visit)(link));
void post_order(link t, void (*visit)(link));
int count(link t);
void destroy(link t);
```

2. 设二叉树的每个节点保存一个字符，中序遍历的结果是 BGCAFHED，后序遍历的结果是 GBFAEDHC，请画图表示这个二叉树。

25.2.2　排序二叉树

排序二叉树（Binary Search Tree，BST）具有这样的性质：根节点的元素大于左子树所有节点的元素，且小于右子树所有节点的元素，并且左子树和右子树也是排序二叉树。其实如图 25.10 所示的二叉树就是排序二叉树，可以看出排序二叉树的中序遍历序列是从小到大排列的。下面的例子实现了排序二叉树的基本操作。

例 25.4　排序二叉树的基本操作

```
/* bst.h */
#ifndef BST_H
#define BST_H

typedef struct node *link;
struct node {
        unsigned char elem;
        link l, r;
};

link search(link t, unsigned char elem);
link insert(link t, unsigned char elem);
link delete(link t, unsigned char elem);
void print_tree(link t);

#endif
/* bst.c */
#include <stdlib.h>
#include <stdio.h>
#include "bst.h"

static link make_node(unsigned char elem)
{
```

```
        link p = malloc(sizeof *p);
        p->elem = elem;
        p->l = p->r = NULL;
        return p;
}

static void free_node(link p)
{
        free(p);
}

link search(link t, unsigned char elem)
{
        if (!t)
                return NULL;
        if (t->elem > elem)
                return search(t->l, elem);
        if (t->elem < elem)
                return search(t->r, elem);
        /* if (t->elem == elem) */
        return t;
}

link insert(link t, unsigned char elem)
{
        if (!t)
                return make_node(elem);
        if (t->elem > elem) /* insert to left subtree */
                t->l = insert(t->l, elem);
        else /* if (t->elem <= elem), insert to right subtree */
                t->r = insert(t->r, elem);
        return t;
}

link delete(link t, unsigned char elem)
{
        link p;
        if (!t)
                return NULL;
        if (t->elem > elem) /* delete from left subtree */
                t->l = delete(t->l, elem);
        else if (t->elem < elem) /* delete from right subtree */
                t->r = delete(t->r, elem);
        else { /* if (t->elem == elem) */
                if (t->l == NULL && t->r == NULL) { /* if t is leaf
                node */
                        free_node(t);
                        t = NULL;
                } else if (t->l) { /* if t has left subtree */
                        /* replace t with the rightmost node in left
                        subtree */
                        for (p = t->l; p->r; p = p->r);
                        t->elem = p->elem;
                        t->l = delete(t->l, t->elem);
        } else { /* if t has right subtree */
                    /* replace t with the leftmost node in right
                    subtree */
                    for (p = t->r; p->l; p = p->l);
                    t->elem = p->elem;
                    t->r = delete(t->r, t->elem);
```

```
                }
        }
        return t;
}

void print_tree(link t)
{
        if (t) {
                printf("(");
                printf("%d", t->elem);
                print_tree(t->l);
                print_tree(t->r);
                printf(")");
        } else
                printf("()");
}
/* main.c */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "bst.h"

#define RANGE 100
#define N 6

void print_elem(link p)
{
        printf("%d", p->elem);
}

int main(void)
{
        int i, elem;
        link root = NULL;
        srand(time(NULL));
        for (i = 0; i < N; i++)
                root = insert(root, rand() % RANGE);
        printf("\t\\tree");
        print_tree(root);
        printf("\n\n");
        while (root) {
                elem = rand() % RANGE;
                if (search(root, elem)) {
                        printf("delete %d in tree\n", elem);
                        root = delete(root, elem);
                        printf("\t\\tree");
                        print_tree(root);
                        printf("\n\n");
            }
        }
}
```

```
$ ./a.out
        \tree(83(77(15()(35()()))())(86()(93()())))

delete 86 in tree
        \tree(83(77(15()(35()()))())(93()()))

delete 35 in tree
        \tree(83(77(15()()))(93()()))

delete 93 in tree
        \tree(83(77(15()()))())
```

```
delete 15 in tree
        \tree(83(77()())())

delete 83 in tree
        \tree(77()())

delete 77 in tree
        \tree()
```

注意这个例子中 insert 和 delete 函数的参数是元素的值，而不是指向节点的指针，这和前面讲链表时实现的 insert 和 delete 函数不同，其实 insert 是一边查找一边做插入操作，delete 也是一边查找一边做删除操作。可以看出，search、insert 和 delete 操作的迭代次数都是和树的层数成正比，如果树中有 n 个节点，树的层数至少是 O(lg(n))，所以这三个操作的时间复杂度在最好情况下是 O(lg(n))，平均情况的时间复杂度分析起来比较复杂，我们略去分析直接给出结果，也是 O(lg(n))。这其实和第 11.4 节的习题 1 介绍的快速排序算法的性质类似。

print_tree 函数通过前序遍历打印树中的所有元素，为什么要打印成这种古怪的格式呢？因为这种格式可以用 Greg Lee 编写的 The Tree Preprocessor（http://www.essex.ac.uk/linguistics/external/clmt/latex4ling/trees/tree/）转换成树形，非常形象：

```
$ ./a.out | ./tree/tree
          83
      ____|___
     |        |
     77      86
   _|__    _|__
  |    |  |    |
  15     |     93
 _|__   |   _|__
|    |  |  |    |
     35  |
   _|__  |
  |    | |
```

```
delete 86 in tree
          83
      ____|___
     |        |
     77      93
   _|__    _|__
  |    |  |    |
  15     |
 _|__   |
|    |  |
     35  |
   _|__  |
  |    | |
```

```
delete 35 in tree
          83
      ____|___
     |        |
     77      93
   _|__    _|__
  |    |  |    |
```

```
        15
       _|__
      |   |

delete 93 in tree
        83
       _|__
      |   |
      77
     _|__
    |   |
     15
    _|__
   |   |

delete 15 in tree
        83
       _|__
      |   |
      77
     _|__
    |   |

delete 83 in tree
        77
       _|__
      |   |

delete 77 in tree
```

习题

1. 请仿照本节的例子用递归实现有序单链表的基本操作：

```
link search(link t, unsigned char elem);
link insert(link t, unsigned char elem);
link delete(link t, unsigned char elem);
```

2. 本节例子中 delete 函数的实现虽然易懂，但效率不够高，请分析一下它做的哪些工作是重复的，然后想办法改进这个函数。

25.3　哈希表

图 25.12 示意了一个简单的哈希表（Hash Table）。

要构造这样的哈希表，首先分配一个指针数组，数组中的每个元素是一个链表的头指针，哈希表由若干个链表组成，每个链表称为一个槽（Slot）。然后向哈希表中插入节点，哪个节点应该插入哪个槽中由哈希函数决定，在这个例子中我们简单地选取哈希函数 h(x) = x % 11，这样任意节点的元素 x 都可以映射成 0～10 之间的一个数，这个数就是槽的编号，把节点插入该槽的链表头部即可。要在哈希表中查找某个元素，首先根据元素的值计算槽编号，然后在槽中查找该元素并返回指向相应节点的指针。要在哈希表中删除某个元素，也是首先根据元素的值

计算槽编号，然后在槽中找到该元素并删除它。

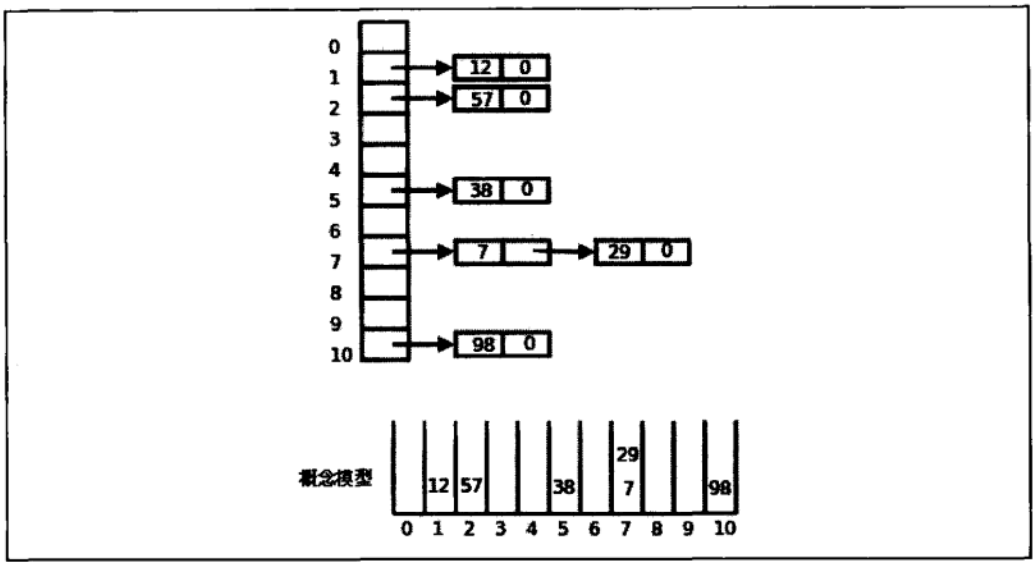


图 25.12 哈希表

请读者自己编写程序构造这样一个哈希表，并实现 search、insert 和 delete 操作：

```
link search(unsigned char elem);
void insert(unsigned char elem);
void delete(unsigned char elem);
```

如果每个槽中至多只有一个元素，可以想象这种情况下 search、insert 和 delete 操作的时间复杂度都是 O(1)，但有时会有多个元素被哈希函数映射到同一个槽中，这称为碰撞（Collision），设计一个好的哈希函数可以把元素比较均匀地分布到各个槽中，尽量避免碰撞。如果能把 n 个元素比较均匀地分布到 m 个槽中，每个槽中约有 n/m 个元素，则 search、insert 和 delete 操作的时间复杂度都是 O(n/m)，如果 n 和 m 的比是常数，则时间复杂度仍然是 O(1)。一般来说，要处理的元素越多，构造哈希表时分配的槽也应该越多，所以 n 和 m 成正比这个假设是成立的。

本节的哈希表，上一节的排序二叉树，以及上一节习题 1 的有向单链表，这三种数据结构都可以表示 n 个元素的集合，都支持 search、insert 和 delete 操作，现在我们比较一下这三种数据结构的 search、insert 和 delete 操作在平均情况下的时间复杂度，顺便拉上有序数组一起做比较，如表 25.1 所示。

表 25.1 几种数据结构的 search、insert 和 delete 操作在平均情况下的时间复杂度比较

数据结构	search	insert	delete
有序数组	O(lgn)	O(n)	O(n)
有序单链表	O(n)	O(n)	O(n)
排序二叉树	O(lgn)	O(lgn)	O(lgn)
哈希表	O(1)	O(1)	O(1)

看起来哈希表在各方面都胜出一筹，其实这样的比较未必公平，哈希表也有很多限制：

- 元素数 n 应该与槽数 m 成正比
- 要找到合适的哈希函数，使元素可以均匀地分布到 m 个槽中
- 哈希表不能把所有元素按顺序输出，而另外三种数据结构都可以做到

习题

1．统计一个文本文件中每个单词出现的次数，然后按出现次数排序并打印输出。单词由连续的英文字母组成，不区分大小写。

2．实现一个函数求两个数组的交集：

```
size_t intersect(const int a[], size_t nmema, const int b[], size_t
nmemb, int c[], size_t nmemc);
```

数组元素是 32 位 int 型的。数组 a 有 nmema 个元素且各不相同，数组 b 有 nmemb 个元素且各不相同。要求找出数组 a 和数组 b 的交集保存到数组 c 中，nmemc 是数组 c 的最大长度，返回值表示交集中实际有多少个元素，如果交集中实际的元素数量超过了 nmemc 则返回 nmemc 个元素。数组 a 和数组 b 的元素数量可能会很大（比如上百万个），需要设计尽可能快的算法。

本阶段总结

在这一阶段我们又学习了很多新的语法规则，首先读者应该回到上篇的阶段总结把那些知识点重新总结一遍。然后我们总结一下各种开发调试工具的用法。

1．gcc 的常用选项

-c

编译生成目标文件（Relocatable），详见第 18.2 节。

-Dmacro[=defn]

定义一个宏，详见第 20.3 节。

-dynamic-linker /path/to/dynamic_linker

指定动态链接器的路径，详见第 18.2 节。

-E

只做预处理而不编译，cpp 命令也可以达到同样的效果，详见第 20.2.1 节。

-foption

指定一些选项控制生成的代码，比如-fno-common表示生成的代码中没有Common符号，详见第19.2.3节，-fPIC表示生成位置无关代码，详见第19.4节。

-g

在生成的目标文件中添加调试信息，所谓调试信息就是源代码和指令之间的对应关系，在gdb调试和objdump反汇编时要用到这些信息，详见第10.1节。如果要编译链接多个目标文件，必须在编译每个目标文件时加-g选项，而不能只在最后链接时加-g选项，详见第19.4节。

-Idir

dir是头文件所在的目录，详见第19.2.2节。

-lname

链接libname库文件，可能是动态库（例如libname.so）也可能是静态库（例如libname.a），如果动态库和静态库都能找到则优先链接动态库，如果指定了-static选项则只链接静态库，详见第19.3节。

-Ldir

dir是库文件所在的目录，详见第19.3节。

-M和-MM

输出“.o文件: .c文件 .h文件”这种形式的Makefile规则，-MM的输出不包括系统头文件，详见第21.4节。

-o outfile

outfile输出文件的文件名，详见第18.2节。

-O?

各种编译优化选项，详见第18.6节。

-print-search-dirs

打印库文件的默认搜索路径，详见第19.3节。

-static

参考上面-lname选项的说明。

-std=c99

如果代码中使用了某些C99的新特性，需要用这个选项编译，详见第6.3节。

-S

编译生成汇编代码，详见第 18.2 节。

-v

打印详细的编译链接过程，详见第 18.2 节。

-Wall

打印所有的警告信息，详见第 1.4 节。

-Wl,options

options 是传递给链接器的选项，详见第 19.4 节。

2．gdb 的基本用法

- 在第 10 章集中介绍了 gdb 的基本命令和调试方法。
- 在第 18.1 节提到了 gdb 的指令级调试和反汇编命令。
- 在第 18.6 节讲编译优化选项时提到，为调试而编译时不要指定优化选项，否则无法做源码级调试，因为源代码和指令可能对应不上。
- 如果一个程序由多个.c 文件编译链接而成，用 gdb 调试时如何定位某个源文件中的某一行代码呢？在第 19.1 节有介绍。
- 在第 22.6 节讲到用 gdb 调试时如何给程序提供命令行参数。

3．其他开发调试工具

ar

把目标文件打包成静态库，详见第 19.3 节。

as

汇编器，详见例 17.1。

hexdump

二进制文件查看工具，以十六进制或 ASCII 字符显示一个文件，详见第 17.5.1 节 。

ld

链接器，详见例 17.1，用--verbose 选项可以显示默认链接脚本，详见第 19.1 节。

ldd

用动态链接器测试一个程序依赖于哪些共享库，并查找这些共享库都在什么路径下，详见第 19.4 节。

make

管理源代码的编译过程，详见第 21 章。

nm

查看符号表，详见第 18.2 节。

objdump

显示目标文件中的信息，在本书中主要用它做反汇编，详见第 17.5.1 节。

od

另一种二进制文件查看工具，以八进制、十六进制或 ASCII 字符显示一个文件，详见第 24.2.1 节。

ranlib

给 ar 打包的静态库建索引，详见第 19.3 节。

readelf

读 ELF 文件信息，详见第 17.5.1 节。

strip

去除可执行文件中的符号信息，详见第 17.5.2 节。

附录 A

字符编码

A.1 ASCII 码

ASCII 码的取值范围是 0～127，可以用 7 个 bit 表示。C 语言规定 char 型占一个字节，如果存放 ASCII 码则只用到低 7 位，高位为 0。ASCII 码表如表 A.1 所示。

表 A.1 ASCII 码表

Dec Hx Oct Char	Dec Hx Oct Char	Dec Hx Oct Char	Dec Hx Oct Char
0 0 000 NUL (null)	32 20 040	64 40 100 @	96 60 140 `
1 1 001 SOH (start of heading)	33 21 041 !	65 41 101 A	97 61 141 a
2 2 002 STX (start of text)	34 22 042 "	66 42 102 B	98 62 142 b
3 3 003 ETX (end of text)	35 23 043 #	67 43 103 C	99 63 143 c
4 4 004 EOT (end of transmission)	36 24 044 $	68 44 104 D	100 64 144 d
5 5 005 ENQ (enquiry)	37 25 045 %	69 45 105 E	101 65 145 e
6 6 006 ACK (acknowledge)	38 26 046 &	70 46 106 F	102 66 146 f
7 7 007 BEL (bell)	39 27 047 '	71 47 107 G	103 67 147 g
8 8 010 BS (backspace)	40 28 050 (	72 48 110 H	104 68 150 h
9 9 011 TAB (horizontal tab)	41 29 051)	73 49 111 I	105 69 151 i
10 a 012 LF (NL line feed, new line)	42 2a 052 *	74 4a 112 J	106 6a 152 j
11 b 013 VT (vertical tab)	43 2b 053 +	75 4b 113 K	107 6b 153 k
12 c 014 FF (NP form feed, new page)	44 2c 054 ,	76 4c 114 L	108 6c 154 l
13 d 015 CR (carriage return)	45 2d 055 -	77 4d 115 M	109 6d 155 m
14 e 016 SO (shift out)	46 2e 056 .	78 4e 116 N	110 6e 156 n
15 f 017 SI (shift in)	47 2f 057 /	79 4f 117 O	111 6f 157 o
16 10 020 DLE (data link escape)	48 30 060 0	80 50 120 P	112 70 160 p
17 11 021 DC1 (device control 1)	49 31 061 1	81 51 121 Q	113 71 161 q
18 12 022 DC2 (device control 2)	50 32 062 2	82 52 122 R	114 72 162 r
19 13 023 DC3 (device control 3)	51 33 063 3	83 53 123 S	115 73 163 s
20 14 024 DC4 (device control 4)	52 34 064 4	84 54 124 T	116 74 164 t
21 15 025 NAK (negative acknowledge)	53 35 065 5	85 55 125 U	117 75 165 u
22 16 026 SYN (synchronous idle)	54 36 066 6	86 56 126 V	118 76 166 v

续表

Dec Hx Oct Char	Dec Hx Oct Char	Dec Hx Oct Char	Dec Hx Oct Char
23 17 027 ETB (end of trans. block)	55 37 067 7	87 57 127 W	119 77 167 w
24 18 030 CAN (cancel)	56 38 070 8	88 58 130 X	120 78 170 x
25 19 031 EM (end of medium)	57 39 071 9	89 59 131 Y	121 79 171 y
26 1a 032 SUB (substitute)	58 3a 072 :	90 5a 132 Z	122 7a 172 z
27 1b 033 ESC (escape)	59 3b 073 ;	91 5b 133 [	123 7b 173 {
28 1c 034 FS (file separator)	60 3c 074 <	92 5c 134 \	124 7c 174 |
29 1d 035 GS (group separator)	61 3d 075 =	93 5d 135]	125 7d 175 }
30 1e 036 RS (recored separator)	62 3e 076 >	94 5e 136 ^	126 7e 176 ~
31 1f 037 US (unit separator)	63 3f 077 ?	95 5f 137 _	127 7f 177 DEL

绝大多数计算机的一个字节是 8 位，取值范围是 0～255，而 ASCII 码并没有规定编号为 128～255 的字符，为了能用一个字节表示更多的字符，各厂商制定了很多种 ASCII 码的扩展规范。注意，虽然通常把这些规范称为扩展 ASCII 码（Extended ASCII），但其实它们并不属于 ASCII 码标准。例如下面这种扩展 ASCII 码由 IBM 制定，在字符终端下被广泛采用，其中包含了很多表格边线字符用来画界面，如图 A.1 所示。

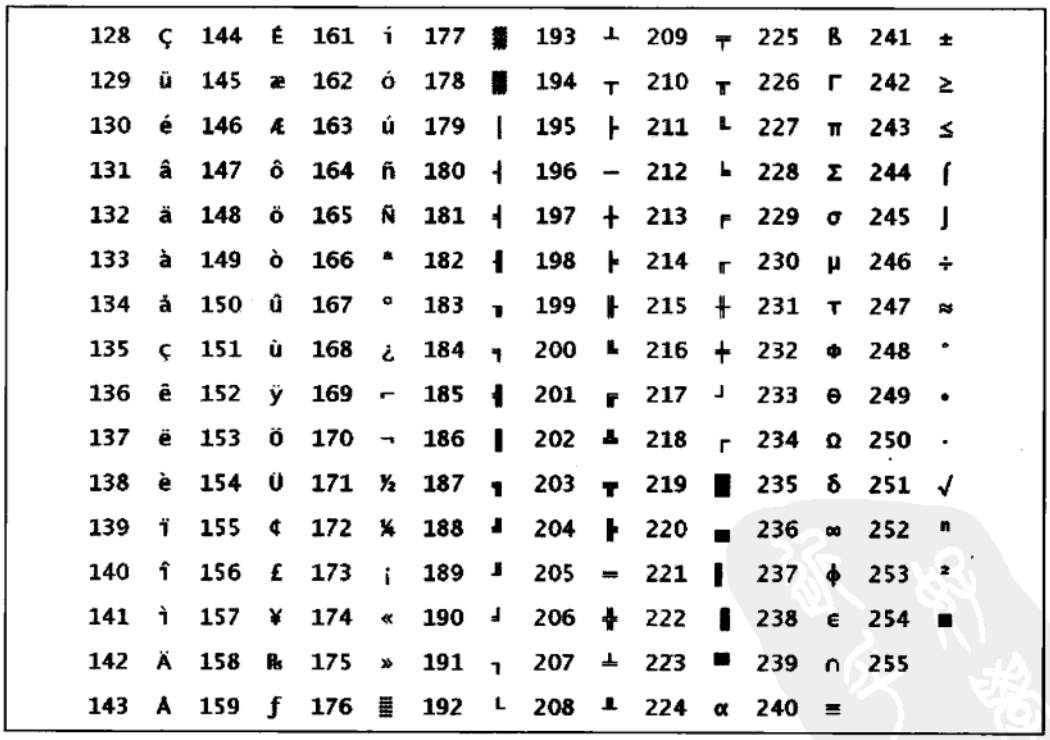


图 A.1 IBM 的扩展 ASCII 码表

在图形界面下最广泛使用的扩展 ASCII 码是 ISO-8859-1，也称为 Latin-1，其中包含欧洲各国语言中最常用的非英文字母，但毕竟只扩展了 128 个字符，一些不常用的字母就没有包含进来。如表 A.2 所示。

表A.2 ISO-8859-1

160	176 °	192 À	208 Ð	224 à	240 ð
161 ¡	177 ±	193 Á	209 Ñ	225 á	241 ñ
162 ¢	178 ²	194 Â	210 Ò	226 â	242 ò
163 £	179 ³	195 Ã	211 Ó	227 ã	243 ó
164 ¤	180 ´	196 Ä	212 Ô	228 ä	244 ô
165 ¥	181 µ	197 Å	213 Õ	229 å	245 õ
166 ¦	182 ¶	198 Æ	214 Ö	230 æ	246 ö
167 §	183 ·	199 Ç	215 ×	231 ç	247 ÷
168 ¨	184 ¸	200 È	216 Ø	232 è	248 ø
169 ©	185 ¹	201 É	217 Ù	233 é	249 ù
170 ª	186 º	202 Ê	218 Ú	234 ê	250 ú
171 «	187 »	203 Ë	219 Û	235 ë	251 û
172 ¬	188 ¼	204 Ì	220 Ü	236 ì	252 ü
173	189 ½	205 Í	221 Ý	237 í	253 ý
174 ®	190 ¾	206 Î	222 Þ	238 î	254 þ
175 ¯	191 ¿	207 Ï	223 ß	239 ï	255 ÿ

编号128～159的是一些控制字符，在表A.2中没有列出。

A.2 Unicode和UTF-8

为了统一全世界各国语言文字和专业领域符号（例如数学符号、乐谱符号）的编码，ISO制定了ISO 10646标准，也称为UCS（Universal Character Set）。UCS编码的长度是31位，可以表示2^{31}个字符。如果两个字符编码的高位相同，只有低16位不同，则它们属于同一个平面（Plane），所以一个平面由2^{16}个字符组成。目前绝大多数常用字符都位于第一个平面（编码范围是0x0000～0xFFFF），称为BMP（Basic Multilingual Plane）或Plane 0，为了向后兼容，其中编号为0～256的字符和ASCII码以及Latin-1相同。UCS编码通常用U-xxxxxxxx这种形式表示，而BMP的编码通常用U+xxxx这种形式表示，其中x是十六进制数字。在ISO制定UCS的同时，另一个厂商联合组织也在着手制定这样的编码，称为Unicode，后来两家联手制定统一的编码，但各自发布各自的标准文档，所以UCS编码和Unicode码是相同的。

有了字符编码，另一个问题就是这样的编码在计算机中怎么表示。现在已经不可能用一个字节表示一个字符了，最直接的想法就是用四个字节表示一个字符，这种表示方法称为UCS-4或UTF-32，UTF是Unicode Transformation Format的缩写。这样表示显然比较浪费存储空间，如果表示BMP字符，4个字节中的高两个字节都是0，如果表示ASCII或Latin-1字符，4个字节中的高3个字节都是0，而我们常用的绝大多数字符都在BMP、ASCII或Latin-1字符集中。

另一种比较节省存储空间的办法是用两个字节表示一个字符，称为 UCS-2 或 UTF-16，这样只能表示 BMP 中的字符，但 BMP 中有一些控制字符用于扩展，可以用两个这样的控制字符表示其他平面的字符，称为 Surrogate Pair。无论是 UTF-32 还是 UTF-16 都有一个更严重的问题是和 C 语言不兼容，在 C 语言中字节 0 表示字符串结尾，库函数 strlen、strcpy 等都依赖于这一点，如果字符串用 UTF-32 存储，其中有很多字节 0 并不表示字符串结尾，那就乱套了。

UNIX 之父 Ken Thompson 提出的 UTF-8 编码很好地解决了这个问题，因此得到广泛应用。和 UTF-16、UTF-32 不同的是，UTF-8 编码的长度不固定，每个字符用 1～6 个字节表示。UTF-8 编码具有以下性质：

- 编码为 U+0000～U+007F 的字符只占一个字节，就是 0x00～0x7F，和 ASCII 码兼容。
- 编码大于 U+007F 的字符用 2～6 个字节表示，每个字节的最高位都是 1，而所有 ASCII 码的最高位都是 0，因此非 ASCII 码字符的 UTF-8 编码中不会出现 ASCII 码的字节（也不会出现字节 0）。
- 在非 ASCII 码字符的多字节编码中，第一个字节的取值范围是 0xC0～0xFD，根据第一个字节可以判断后面还有几个字节也属于当前字符的编码。后面每个字节的取值范围都是 0x80～0xBF，见下面的详细说明。
- 所有 Unicode 字符（共 2^{31} 个）都可以用 UTF-8 编码表示出来。
- UTF-8 编码最长 6 个字节，BMP 字符的 UTF-8 编码最长三个字节。
- 0xFE 和 0xFF 这两个字节在 UTF-8 编码中不会出现。

具体来说，UTF-8 编码有以下几种格式：

```
U-00000000 – U-0000007F: 0xxxxxxx
U-00000080 – U-000007FF: 110xxxxx 10xxxxxx
U-00000800 – U-0000FFFF: 1110xxxx 10xxxxxx 10xxxxxx
U-00010000 – U-001FFFFF: 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
U-00200000 – U-03FFFFFF: 111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx
U-04000000 – U-7FFFFFFF: 1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx
10xxxxxx
```

ASCII 码字符的 UTF-8 编码只有一个字节（就是 ASCII 码本身），最高位是 0。非 ASCII 码字符的第一个字节最高位是 1，并且后面至少还要跟一个 1，最高位后面跟几个 1 就表示后面还有几个字节也属于当前字符的编码，例如 111110xx，最高位后面跟 4 个 1，表示后面还有 4 个字节也属于当前字符的编码。后面每个字节的最高两位都是 10，而第一个字节的最高两位要么是 0x，要么是 11，因此可以和后面的字节区分开。这样的设计有利于误码同步，例如在网络传输过程中丢失了几个字节，很容易判断当前字符是不完整的，也很容易找到下一个字符应该从哪儿开始，顶多丢掉一两个字符就可以同步了，而不会导致后面的解码过程全部错乱。上面的格式中标为 x 的位就是字符的 Unicode 码，最后一种 6 字节的格式中 x 位有 31 个，可以表示 31 位的 Unicode 码。UTF-8 编码就像一列火车，第一个字节是车头，后面每个字节是车厢，其中承载的货物是 Unicode 码。UTF-8 规定承载的 Unicode 码以大端表示，就是说第一个字节中的 x 位是 Unicode 码的

高位，后面字节中的 x 位是 Unicode 码的低位。

例如 U+00A9(©字符)的二进制是 10101001，编码成 UTF-8 是 11000010 10101001（0xC2 0xA9），但不能编码成 11100000 10000010 10101001，UTF-8 规定每个字符只能用尽可能少的字节来编码。

A.3 在 Linux C 编程中使用 Unicode 和 UTF-8

目前各种 Linux 发行版都支持 UTF-8 编码，在磁盘上保存一个含有非 ASCII 字符的文本文件，默认是以 UTF-8 编码的。当前系统的字符编码设置可以用 locale 命令查看：

```
$ locale
LANG=zh_CN.UTF-8
LC_CTYPE="zh_CN.UTF-8"
LC_NUMERIC="zh_CN.UTF-8"
LC_TIME="zh_CN.UTF-8"
LC_COLLATE="zh_CN.UTF-8"
LC_MONETARY="zh_CN.UTF-8"
LC_MESSAGES="zh_CN.UTF-8"
LC_PAPER="zh_CN.UTF-8"
LC_NAME="zh_CN.UTF-8"
LC_ADDRESS="zh_CN.UTF-8"
LC_TELEPHONE="zh_CN.UTF-8"
LC_MEASUREMENT="zh_CN.UTF-8"
LC_IDENTIFICATION="zh_CN.UTF-8"
LC_ALL=
```

Locale 定义了语言、字符编码、国家地区、数字格式、货币格式等参数，详见 locale(1)、locale(5)、locale(7)。locale 命令列出的这些参数每一个都可以用环境变量单独设置，但通常这些参数的取值是一致的。在我的系统中只设置了环境变量 LANG，其他参数没有设置，用 locale 命令可以看到其他 Locale 参数也继承了环境变量 LANG 的值。

常用汉字都位于 BMP 中，所以一个汉字的 UTF-8 编码通常是 3 个字节。例如编辑一个 C 程序：

```
#include <stdio.h>

int main(void)
{
        printf("你好\n");
        return 0;
}
```

源文件是以 UTF-8 编码存储的：

```
$ hexdump -C nihao.c
00000000  23 69 6e 63 6c 75 64 65  20 3c 73 74 64 69 6f 2e  |#include
<stdio.|
00000010  68 3e 0a 0a 69 6e 74 20  6d 61 69 6e 28 76 6f 69  |h>..int
main(voi|
```

```
    00000020  64 29 0a 7b 0a 09 70 72  69 6e 74 66 28 22 e4 bd
|d).{..printf("..|
    00000030  a0 e5 a5 bd 5c 6e 22 29  3b 0a 09 72 65 74 75 72
|....\n");..retur|
    00000040  6e 20 30 3b 0a 7d 0a                        |n 0;.}.|
    00000047
```

其中 e4 bd a0 这三个字节就是“你”的 UTF-8 编码，e5 a5 bd 这三个字节就是“好”的 UTF-8 编码。把它编译成目标文件，"你好\n"这个字符串就成了这样一串字节：e4 bd a0 e5 a5 bd 0a 00，转义序列由两个字节变成一个字节，字符串末尾添了一个字节 0，而汉字仍然占 3 个字节，在 C 标准中多字节编码的字符称为 Multibyte Character。运行这个程序会把这一串字节输出到当前终端设备，如果当前终端能够识别 UTF-8 编码（比如图形界面的终端窗口）就能打印出汉字，如果不能识别 UTF-8 编码（比如一般的字符终端）就打印不出汉字。也就是说，在这个程序中识别汉字的工作既不是由 C 编译器做的也不是由 printf 函数做的，C 编译器原封不动地把源文件中的 UTF-8 编码复制到目标文件中，printf 函数再把这一串字节当做以 Null 结尾的字符串原封不动地输出到终端设备，识别汉字的工作是由终端设备做的。

仅有这种程度的汉字支持是不够的，有时候我们需要在 C 程序中操作字符串里的字符，比如求字符串"你好\n"中有几个汉字或字符，用 strlen 就不灵了，因为 strlen 求的是**字节数**而不是**字符数**。为了在程序中操作 Unicode 字符，C 标准定义了宽字符（Wide Character）类型 wchar_t。在字符常量或字符串字面值前面加一个 L 就表示宽字符常量或宽字符串，例如定义 wchar_t c = L'你';，变量 c 的值就是汉字“你”的 31 位 Unicode 码，而 L"你好\n"就相当于数组 wchar_t str[] = { L'你', L'好', L'\n', 0 };。C 标准还定义了一些操作宽字符串的库函数，例如 wcslen 函数可以取宽字符串中的字符个数。

注意 Wide Character 和 Multibyte Character 这两个概念的区别：

1．C 标准没有规定 Wide Character 和 Multibyte Character 应该采用什么编码，但目前各种 Linux 发行版的 Wide Character 都采用 Unicode 码，Multibyte Character 都采用 UTF-8 编码。

2．每个 Wide Character 有固定的长度，用 wchar_t 类型来表示，而 Multibyte Character 是 Wide Character 的转换编码，每个 Multibyte Character 的长度不固定，没有规定一种类型来表示 Multibyte Character。

3．Wide Character 中可能包含字节 0，所以不能保存在普通的以 Null 结尾的字符串中，而必须保存在宽字符串中。而 Multibyte Character 中不允许出现字节 0（只有 Null 字符除外），因此可以保存在以 Null 结尾的字符串中。

4．Wide Character 适合做字符运算，比如统计字符数，而 Multibyte Character 适合做存储和传输，存储时比较节省空间，传输时有较好的容错性。

看下面的例子：

```
#include <stdio.h>
#include <locale.h>

int main(void)
{
        if (!setlocale(LC_CTYPE, "")) {
                fprintf(stderr, "Can't set the specified locale! "
                        "Check LC_ALL, LC_CTYPE, LANG.\n");
                return 1;
        }
        printf("%ls", L"你好\n");
        return 0;
}
```

宽字符串 L"你好\n"在源代码中当然还是 UTF-8 编码，但编译器会把它转换成 4 个 Unicode 码 0x00004f60 0x0000597d 0x0000000a 0x00000000 保存到目标文件中，按小端存储就是 60 4f 00 00 7d 59 00 00 0a 00 00 00 00 00 00 00，用 hexdump 命令查看目标文件应该能找到这些字节。

printf 的转换说明%ls 表示把后面的参数按宽字符串解释，不是见到字节 0 就结束，而是见到 Null 字符的 Unicode 码（4 个字节 0）才结束，但输出到终端仍然要以 Multibyte Character 编码输出，这样终端设备才能识别，所以 printf 函数在内部先把宽字符串转换成 UTF-8 编码的 Multibyte Character 字符串再输出到终端。我们把输出重定向到文件会看得更清楚：

```
$ gcc main.c
$ ./a.out > foo
$ hexdump -C foo
00000000  e4 bd a0 e5 a5 bd 0a                              |.......|
00000007
```

最后解释一下 setlocale(3)函数。Locale 中的 LC_CTYPE 参数影响 C 语言对宽字符和宽字符串的处理，上面提到 printf 函数在内部先把宽字符串转换成 UTF-8 编码的 Multibyte Character 字符串再输出到终端，这样做的前提是 LC_CTYPE 被设置为 UTF-8 编码，上面程序中的 setlocale 调用就是为了做这个设置。

虽然当前系统的 Locale 设置是"zh_CN.UTF-8"，但 C 程序在启动时各种 Locale 参数都设置成默认值"C"，并不继承当前系统的 Locale 设置。这样规定是为了代码的可移植性，如果一个程序不调用 setlocale 函数，那么它不管在什么系统上运行，其各种 Locale 参数都是"C"，其中 LC_CTYPE 参数是"C"表示采用 ASCII 字符集。上面的程序如果去掉 setlocale 调用，则无法打印出“你好”，因为这两个字符不属于 ASCII 字符集。

我们调用 setlocale 传的第一个参数是 LC_CTYPE，它在 C 语言中被定义成一个整数常量，第二个参数是设置给它的值。如果第二个参数是空字符串""表示采用当前系统的 Locale 设置，setlocale 函数依次查找环境变量 LC_ALL、LC_CTYPE 和 LANG，找到第一个有定义的环境变量就用它的值来设置 LC_CTYPE。

关于 Unicode 和 UTF-8 本节只介绍了最基本的概念，部分内容出自参考文献[35]，读者可进一步参考这篇文章。

参考文献

[1] How To Think Like A Computer Scientist: Learning with C++. Allen B. Downey.

[2] Programming from the Ground Up: An Introduction to Programming using Linux Assembly Language. Jonathan Bartlett.

[3] The C Programming Language. 2. Brian W. Kernighan 和 Dennis M. Ritchie.

[4] Standard C: A Reference. P. J. Plauger 和 Jim Brodie.

[5] The Standard C Library. P. J. Plauger.

[6] Rationale for International Standard — Programming Languages — C. 5.10.

[7] The Art of UNIX Programming. Eric Raymond.

[8] ISO/IEC 9899: Programming Languages — C. 2.

[9] Fundamentals of Digital Logic with VHDL Design. 2. Stephen Brown 和 Zvonko Vranesic.

[10] Introduction to Automata Theory, Languages, and Computation. 2. John E. Hopcroft、Rajeev Motwani 和 Jeffrey D. Ullman.

[11] Compilers: Principles, Techniques, & Tools. 2. Alfred V. Aho、Monica S. Lam、Ravi Sethi 和 Jeffrey D. Ullman.

[12] Structure and Interpretation of Computer Programs. 2. Harold Abelson、Gerald Jay Sussman 和 Julie Sussman.

[13] The Mythical Man-Month: Essays on Software Engineering. Anniversary Edition. Frederick P. Brooks, Jr..

[14] Linux 内核源代码目录下的 Documentation/CodingStyle 文件.

[15] Debugging with GDB: The GNU Source-Level Debugger. 9. Richard Stallman、Roland Pesch 和 Stan Shebs.

[16] Introduction to Algorithms. 2. Thomas H. Cormen、Charles E. Leiserson、Ronald L. Rivest 和 Clifford Stein.

[17] The Art of Computer Programming. Donald E. Knuth.

[18] Programming Pearls. 2. Jon Bentley.

[19] Object-Oriented Software Construction. Bertrand Meyer.

[20] Algorithms + Data Structures = Programs. Niklaus Wirth.

[21] Expert C Programming. Peter van der Linden.

[22] Linux Assembly HOWTO（http://tldp.org/HOWTO/Assembly-HOWTO/）. Konstantin Boldyshev 和 Francois-Rene Rideau.

[23] Introduction to 80x86 Assembly Language and Computer Architecture. Richard C. Detmer.

[24] GCC online documentation（http://gcc.gnu.org/onlinedocs/）.

[25] GCC: The Complete Reference. Arthur Griffith.

[26] GNU Binutils Documentation（http://www.gnu.org/software/binutils/）.

[27] GNU Make Manual（http://www.gnu.org/software/make/manual/）.

[28] Smashing The Stack For Fun And Profit，网上到处都可以搜到这篇文章. Aleph One.

[29] The New C: It All Began with FORTRAN（http://www.ddj.com/cpp/184401313）. Randy Meyers.

[30] Concrete Mathematics. 2. Ronald L. Graham、Donald E. Knuth 和 Oren Patashnik.

[31] Advanced Programming in the UNIX Environment. 2. W. Richard Stevens 和 Stephen A. Rago.

[32] Understanding the Linux Kernel. 3. Daniel P. Bovet 和 Marco Cesati.

[33] TCP/IP Illustrated, Volume 1: The Protocols. W. Richard Stevens.

[34] UNIX Network Programming, Volume 1: The Sockets Networking API. 3. W. Richard Stevens、Bill Fenner 和 Andrew M. Rudoff.

[35] UTF-8 and Unicode FAQ, http://www.cl.cam.ac.uk/~mgk25/unicode.html. Markus Kuhn.

索引

符号

A

B

C

D

E

F

G

H

I

N

O

P

T

U

V

[General Information]
书名=一站式学习C编程
作者=宋劲杉编著
页数=471
出版社=北京市：电子工业出版社
出版日期=2011.04
SS号=12870066
DX号=000008070703
URL=http://book2.duxiu.com/bookDetail.jsp?dxNumber=000008070703&d=D4D769847FC8FAD6950A4AB697399B8C