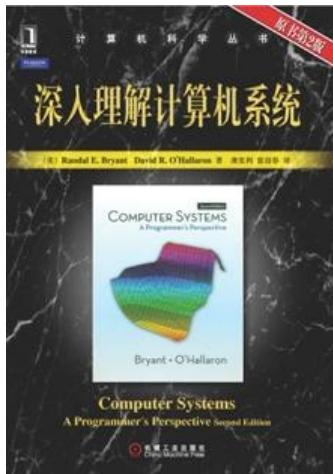


深入理解计算机系统（原书第 2 版）（CSAPP）

- 这本书被誉为“价值超过等重量黄金的无价资源宝库”；
- 这本书是 Amazon 五星图书，最伟大的计算机科学教材之一；
- 这本书由卡耐基梅隆大学计算机学院院长，IEEE 和 ACM 双院士倾力推荐；
- 这本书被超过 80 所美国和世界一流大学计算机专业选用本书为教材。



预计出版日期：2010.11

有学生如此评价本书第一版——

起点低，覆盖面广，适合大三，大四的本科生；一个优秀程序员必须理解底层操作，对体系结构、操作系统必须有认识，这本书的内容很适合作为起点。

如果你是个在校生，如果你学完了组成原理和体系结构，强烈建议你看看这本书。

这本书的确是好书，我没有理由不为此书而拍案叫绝，如果学计算机的能够依此书为教材那就再好不过了。看过此书你会对计算机原理、汇编语言和 C 语言有根本性的认识。

这本书写得太棒了！我在很多地方都弄不清楚的概念在此书中都有准确明了的阐述。每个概念都很简洁却都抓住重点。

经典中的经典。写代码的都应该买一本。

特别好的一本书，特别是对在校大学生。整个知识体系非常完善，相关阐述也是由浅入深，精品！



计算机系统漫游

计算机系统是由硬件和系统软件组成的，它们共同工作来运行应用程序。虽然系统的具体实现方式随着时间不断变化，但是系统内的概念却没有改变。所有计算机系统都有相似的硬件和软件组件，它们执行着相似的功能。一些程序员希望深入了解这些组件是如何工作的，以及这些组件是如何影响程序的正确性和性能的，以此来提高自身的技能。本书便是为这些读者而写的。

现在就要开始一次有趣的漫游历程了。如果你全力投身学习本书中的概念，完全理解底层计算机系统以及它对应用程序的影响，那么你将会逐渐成为凤毛麟角的“权威”程序员。

你将会学习一些实践技巧，比如如何避免由计算机表示数字的方式导致的奇怪的数字错误。你将学会怎样通过一些聪明的小窍门来优化你的 C 代码，以充分利用现代处理器和存储器系统的设计。你将了解到编译器是如何实现过程调用的，以及如何利用这些知识避免缓冲区溢出错误带来的安全漏洞，这些弱点会给网络和因特网软件带来了巨大的麻烦。你将学会如何识别和避免链接时那些令人讨厌的错误，它们困扰着普通的程序员。你将学会如何编写自己的 Unix 外壳、自己的动态存储分配包，甚至是自己的 Web 服务器。你会认识到并发带来的希望和陷阱，当单个芯片上集成了多个处理器核时，这个主题变得越来越重要。

在关于 C 编程语言的经典文献 [58] 中，Kernighan 和 Ritchie 通过图 1-1 所示的 hello 程序来向读者介绍 C 语言。尽管 hello 程序非常简单，但是为了让它完成运行，系统的每个主要组成部分都需要协调工作。从某种意义上来说，本书的目的就是要帮助你了解当你在系统上执行 hello 程序时，系统发生了什么以及为什么会这样。

```
----- code/intro/hello.c
1 #include <stdio.h>
2
3 int main()
4 {
5     printf("hello, world\n");
6 }
```

----- code/intro/hello.c

图 1-1 hello 程序

我们通过跟踪 hello 程序的生命周期来开始对系统的学习——从它被程序员创建，到在系统上运行，输出简单的消息，然后终止。我们将沿着这个程序的生命周期，简要地介绍一些逐步出现的关键概念、专业术语和组成部分。后面的章节将围绕这些内容展开。

1.1 信息就是位 + 上下文

hello 程序的生命周期是从一个源程序（或者说源文件）开始的，即程序员利用编辑器创建并保存的文本文件，文件名是 hello.c。源程序实际上就是一个由值 0 和 1 组成的位 (bit) 序列，8 个位被组织成一组，称为字节。每个字节表示程序中某个文本字符。

大部分的现代系统都使用 ASCII 标准来表示文本字符，这种方式实际上就是用一个唯一的单字节大小的整数值来表示每个字符。例如图 1-2 中给出了 hello.c 程序的 ASCII 码表示。

hello.c 程序以字节序列的方式存储在文件中。每个字节都有一个整数值，而该整数值对



应于某个字符。例如，第一个字节的整数值是 35，它对应的就是字符 ‘#’；第二个字节整数值为 105，它对应的字符是 ‘i’，依次类推。注意，每个文本行都是以一个不可见的换行符 ‘\n’ 来结束的，它所对应的整数值为 10。像 hello.c 这样只由 ASCII 字符构成的文件称为文本文件，所有其他文件都称为二进制文件。

#	i	n	c	l	u	d	e	<sp>	<	s	t	d	i	o	.
35	105	110	99	108	117	100	101	32	60	115	116	100	105	111	46
h	>	\n	\n	i	n	t	<sp>	m	a	i	n	()	\n	{
104	62	10	10	105	110	116	32	109	97	105	110	40	41	10	123
\n	<sp>	<sp>	<sp>	p	r	i	n	t	f	("	h	e	l	
10	32	32	32	112	114	105	110	116	102	40	34	104	101	108	
l	o	,	<sp>	w	o	r	l	d	\	n	")	;	\n	}
108	111	44	32	119	111	114	108	100	92	110	34	41	59	10	125

图 1-2 hello.c 的 ASCII 文本表示

hello.c 的表示方法说明了一个基本的思想：系统中所有的信息——包括磁盘文件、存储器中的程序、存储器中存放的用户数据以及网络上传送的数据，都是由一串位表示的。区分不同数据对象的唯一方法是我们读到这些数据对象时的上下文。比如，在不同的上下文中，一个同样的字节序列可能表示一个整数、浮点数、字符串或者机器指令。

作为程序员，我们需要了解数字的机器表示方式，因为它们与实际的整数和实数是不同的。它们是对真值的有限近似值，有时候会有意想不到的行为表现。这方面的基本原理将在第 2 章中详细描述。

C 编程语言的起源

C 语言是贝尔实验室的 Dennis Ritchie 于 1969 年～1973 年间创建的。美国国家标准学会（American National Standards Institute, ANSI）在 1989 年颁布了 ANSI C 的标准，后来使 C 语言标准化成为了国际标准化组织（International Standards Organization, ISO）的责任。这些标准定义了 C 语言和一系列函数库，即所谓的“C 标准库”。Kernighan 和 Ritchie 在他们的经典著作中描述了 ANSI C，这本著作被人们满怀感情地称为“K&R”[58]。用 Ritchie 的话来说 [88]，C 语言是“古怪的、有缺陷的，但也是一个巨大的成功”。为什么会成功呢？

- **C 语言与 Unix 操作系统关系密切。** C 语言从一开始就是作为一种用于 Unix 系统的程序设计语言而开发出来的。大部分 Unix 内核，以及所有支撑工具和函数库都是用 C 语言编写的。20 世纪 70 年代后期到 80 年代初期，Unix 在高等院校兴起，许多人开始接触 C 语言并喜欢上了它。因为 Unix 几乎全部是用 C 编写的，所以可以很方便地移植到新的机器上，这种特点为 C 和 Unix 赢得了更为广泛的支持。
- **C 语言小而简单。** 掌控 C 语言设计的是一个人而非一个协会，因此这是一个简洁明了、没有什么冗赘的一致的设计。K&R 这本书用了大量的例子和练习描述了完整的 C 语言及其标准库，而全书不过 261 页。C 语言的简单使它相对而言易于学习，也易于移植到不同的计算机上。
- **C 语言是为实践目的设计的。** C 语言是为实现 Unix 操作系统而设计的。后来，其他人发现能够用这门语言无障碍地编写他们想要的程序。

C 语言是系统级编程的首选，同时它也非常适用于应用级程序的编写。然而，它也并非适



用于所有的程序员和所有的情况。C 语言的指针是造成困惑和程序错误的一个常见原因。同时，C 语言还缺乏对非常有用的抽象（例如类、对象和异常）的显式支持。像 C++ 和 Java 这样针对应用级程序的新程序设计语言解决了这些问题。

1.2 程序被其他程序翻译成不同的格式

hello 程序的生命周期是从一个高级 C 语言程序开始的，因为这种形式能够被人读懂。然而，为了在系统上运行 hello.c 程序，每条 C 语句都必须被其他程序转化为一系列的低级机器语言指令。然后这些指令按照一种称为可执行目标程序的格式打好包，并以二进制磁盘文件的形式存放起来。目标程序也称为可执行目标文件。

在 Unix 系统上，从源文件到目标文件的转化是由编译器驱动程序完成的：

```
unix> gcc -o hello hello.c
```

在这里，GCC 编译器驱动程序读取源程序文件 hello.c，并把它翻译成一个可执行目标文件 hello。这个翻译的过程可分为四个阶段完成，如图 1-3 所示。执行这四个阶段的程序（预处理器、编译器、汇编器和链接器）一起构成了编译系统（compilation system）。

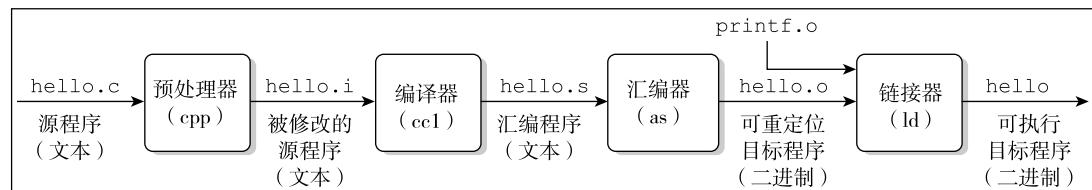


图 1-3 编译系统

- 预处理阶段。预处理器（cpp）根据以字符 # 开头的命令，修改原始的 C 程序。比如 hello.c 中第 1 行的 `#include <stdio.h>` 命令告诉预处理器读取系统头文件 stdio.h 的内容，并把它直接插入到程序文本中。结果就得到了另一个 C 程序，通常是以 .i 作为文件扩展名。
- 编译阶段。编译器（cc1）将文本文件 hello.i 翻译成文本文件 hello.s，它包含一个汇编语言程序。汇编语言程序中的每条语句都以一种标准的文本格式确切地描述了一条低级机器语言指令。汇编语言是非常有用的，因为它为不同高级语言的不同编译器提供了通用的输出语言。例如，C 编译器和 Fortran 编译器产生的输出文件用的都是一样的汇编语言。
- 汇编阶段。接下来，汇编器（as）将 hello.s 翻译成机器语言指令，把这些指令打包成一种叫做可重定位目标程序（relocatable object program）的格式，并将结果保存在目标文件 hello.o 中。hello.o 文件是一个二进制文件，它的字节编码是机器语言指令而不是字符。如果我们在文本编辑器中打开 hello.o 文件，看到的将是一堆乱码。
- 链接阶段。请注意，hello 程序调用了 printf 函数，它是每个 C 编译器都会提供的标准 C 库中的一个函数。printf 函数存在于一个名为 printf.o 的单独的预编译好了的目标文件中，而这个文件必须以某种方式合并到我们的 hello.o 程序中。链接器（ld）就负责处理这种合并。结果就得到 hello 文件，它是一个可执行目标文件（或者简称为可执行文件），可以被加载到内存中，由系统执行。

GNU 项目

GCC 是 GNU（GNU 是 GNU's Not Unix 的缩写）项目开发出来的众多有用工具之一。GNU 项目是 1984 年由 Richard Stallman 发起的一个免税的慈善项目。该项目的目标非常宏大，就是开



发出一个完整的类 Unix 的系统，其源代码能够不受限制地被修改和传播。GNU 项目已经开发出了一个包含 Unix 操作系统的所有主要部件的环境，但内核除外，内核是由 Linux 项目独立发展而来的。GNU 环境包括 EMACS 编辑器、GCC 编译器、GDB 调试器、汇编器、链接器、处理二进制文件的工具以及其他一些部件。GCC 编译器已经发展到支持许多不同的语言，能够针对许多不同的机器生成代码。支持的语言包括 C、C++、Fortran、Java、Pascal、面向对象 C 语言（Objective-C）和 Ada。

GNU 项目取得了非凡的成绩，但是却常常被忽略。现代开放源码运动（通常和 Linux 联系在一起）的思想起源是 GNU 项目中自由软件（free software）的概念。（此处的 free 为自由言论（free speech）中“自由”之意，而非免费啤酒（free beer）中“免费”之意。）而且，Linux 如此受欢迎在很大程度上还要归功于 GNU 工具，因为它们给 Linux 内核提供了环境。

1.3 了解编译系统如何工作是大有益处的

对于像 `hello.c` 这样简单的程序，我们可以依靠编译系统生成正确有效的机器代码。但是，有一些重要的原因促使程序员必须知道编译系统是如何工作的，其原因如下：

- 优化程序性能。现代编译器都是成熟的工具，通常可以生成很好的代码。作为程序员，我们无需为了写出高效代码而去了解编译器的内部工作。但是，为了在 C 程序中做出好的编码选择，我们确实需要了解一些机器代码以及编译器将不同的 C 语句转化为机器代码的方式。例如，一个 `switch` 语句是否总是比一系列的 `if-then-else` 语句高效得多？一个函数调用的开销有多大？`while` 循环比 `for` 循环更有效吗？指针引用比数组索引更有效吗？为什么将循环求和的结果放到一个本地变量中，与将其放到一个通过引用传递过来的参数中相比，运行速度要快很多呢？为什么我们只是简单地重新排列一下一个算术表达式中的括号就能让一个函数运行得更快？

在第 3 章中，我们将介绍两种相关的机器语言：IA32 和 x86-64。IA32 是 32 位的，目前普遍应用于运行 Linux、Windows 以及较新版本的 Macintosh 操作系统的机器上；x86-64 是 64 位的，可以用在比较新的微处理器上。我们会介绍编译器是如何把不同的 C 语言结构转换成它们的机器语言的。第 5 章，你将学习如何通过简单转换 C 语言代码，以帮助编译器更好地完成工作，从而调整 C 程序的性能。在第 6 章，你将学习到存储器系统的层次结构特性，C 语言编译器将数组存放在存储器中的方式，以及 C 程序又是如何能够利用这些知识从而更高效地运行。

- 理解链接时出现的错误。根据我们的经验，一些最令人困扰的程序错误往往都与链接器操作有关，尤其是当你试图构建大型的软件系统时。例如，链接器报告它无法解析一个引用，这是什么意思？静态变量和全局变量的区别是什么？如果你在不同的 C 文件中定义了名字相同的两个全局变量会发生什么？静态库和动态库的区别是什么？我们在命令行上排列库的顺序有什么影响？最严重的是，为什么有些链接错误直到运行时才会出现？在第 7 章，你将得到这些问题的答案。
- 避免安全漏洞。多年来，缓冲区溢出错误是造成大多数网络和 Internet 服务器上安全漏洞的主要原因。存在这些错误是因为很少有人能理解限制他们从不受信任的站点接收数据的数量和格式的重要性。学习安全编程的第一步就是理解数据和控制信息存储在程序栈上的方式会引起的后果。作为学习汇编语言的一部分，我们将在第 3 章中描述堆栈原理和缓冲区溢出错误。我们还将学习程序员、编译器和操作系统可以用来降低攻击威胁的方法。



1.4 处理器读并解释存储在存储器中的指令

此刻，hello.c 源程序已经被编译系统翻译成了可执行目标文件 hello，并存放在磁盘上。要想在 Unix 系统上运行该可执行文件，我们将它的文件名输入到称为外壳（shell）的应用程序中：

```
unix> ./hello
hello, world
unix>
```

外壳是一个命令行解释器，它输出一个提示符，等待你输入一个命令行，然后执行这个命令。如果该命令行的第一个单词不是一个内置的外壳命令，那么外壳就会假设这是一个可执行文件的名字，它将加载并运行这个文件。所以在此例中，外壳将加载并运行 hello 程序，然后等待程序终止。hello 程序在屏幕上输出它的信息，然后终止。外壳随后输出一个提示符，等待下一个输入的命令行。

1.4.1 系统的硬件组成

为了理解运行 hello 程序时发生了什么，我们需要了解一个典型系统的硬件组织，如图 1-4 所示。这张图是 Intel Pentium 系统产品系列的模型，但是所有其他系统也有相同的外观和特性。现在不要担心这张图的复杂性——我们将在本书分阶段介绍大量的细节。

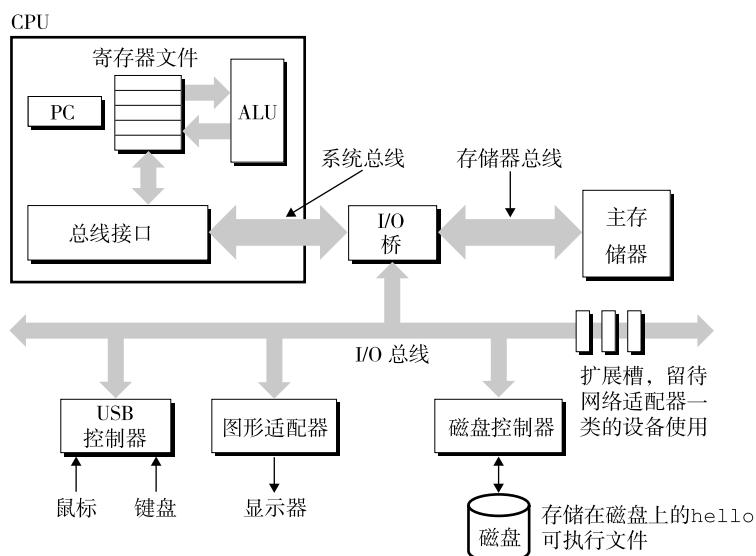


图 1-4 一个典型的硬件组成

CPU：中央处理单元；ALU：算术 / 逻辑单元；PC：程序计数器；USB：通用串行总线

1. 总线

贯穿整个系统的是一组电子管道，称做总线，它携带信息字节并负责在各个部件间传递。通常总线被设计成传送定长的字节块，也就是字（word）。字中的字节数（即字长）是一个基本的系统参数，在各个系统中的情况都不尽相同。现在的大多数机器字长有的是 4 个字节（32 位），有的是 8 个字节（64 位）。为了讨论的方便，假设字长为 4 个字节，并且总线每次只传送 1 个字。

2. I/O 设备

输入 / 输出（I/O）设备是系统与外部世界的联系通道。我们的示例系统包括 4 个 I/O 设备：作为用户输入的键盘和鼠标，作为用户输出的显示器，以及用于长期存储数据和程序的磁盘驱动



器（简单地说就是磁盘）。最初，可执行程序 hello 就存放在磁盘上。

每个 I/O 设备都通过一个控制器或适配器与 I/O 总线相连。控制器和适配器之间的区别主要在于它们的封装方式。控制器是置于 I/O 设备本身的或者系统的主印制电路板（通常称为主板）上的芯片组，而适配器则是一块插在主板插槽上的卡。无论如何，它们的功能都是在 I/O 总线和 I/O 设备之间传递信息。

第 6 章会更多地说明磁盘之类的 I/O 设备是如何工作的。在第 10 章，你将学习如何在应用程序中利用 Unix I/O 接口访问设备。我们将特别关注网络类设备，不过这些技术对于其他设备来说也是通用的。

3. 主存

主存是一个临时存储设备，在处理器执行程序时，用来存放程序和程序处理的数据。从物理上来说，主存是由一组动态随机存取存储器（DRAM）芯片组成的。从逻辑上来说，存储器是一个线性的字节数组，每个字节都有其唯一的地址（即数组索引），这些地址是从零开始的。一般来说，组成程序的每条机器指令都由不同数量的字节构成。与 C 程序变量相对应的数据项的大小是根据类型变化的。例如，在运行 Linux 的 IA32 机器上，short 类型的数据需要 2 个字节，int、float 和 long 类型需要 4 个字节，而 double 类型需要 8 个字节。

第 6 章将具体介绍存储技术，如 DRAM 芯片是如何工作的，以及它们又是如何组合起来构成主存的。

4. 处理器

中央处理单元（CPU），简称处理器，是解释（或执行）存储在主存中指令的引擎。处理器的核心是一个字长的存储设备（或寄存器），称为程序计数器（PC）。在任何时刻，PC 都指向主存中的某条机器语言指令（即含有该条指令的地址）。[⊖]

从系统通电开始，直到系统断电，处理器一直在不断地执行程序计数器指向的指令，再更新程序计数器，使其指向下一条指令。处理器看上去是按照一个非常简单的指令执行模型来操作的，这个模型是由指令集结构决定的。在这个模型中，指令按照严格的顺序执行，而执行一条指令包含执行一系列的步骤。处理器从程序计数器（PC）指向的存储器处读取指令，解释指令中的位，执行该指令指示的简单操作，然后更新 PC，使其指向下一条指令，而这条指令并不一定与存储器中刚刚执行的指令相邻。

这样的简单操作并不多，而且操作是围绕着主存、寄存器文件（register file）和算术/逻辑单元（ALU）进行的。寄存器文件是一个小的存储设备，由一些 1 字长的寄存器组成，每个寄存器都有唯一的名字。ALU 计算新的数据和地址值。下面列举一些简单操作的例子，CPU 在指令的要求下可能会执行以下操作：

- 加载：把一个字节或者一个字从主存复制到寄存器，以覆盖寄存器原来的内容。
- 存储：把一个字节或者一个字从寄存器复制到主存的某个位置，以覆盖这个位置上原来的内容。
- 操作：把两个寄存器的内容复制到 ALU，ALU 对这两个字做算术操作，并将结果存放到一个寄存器中，以覆盖该寄存器中原来的内容。
- 跳转：从指令本身中抽取一个字，并将这个字复制到程序计数器（PC）中，以覆盖 PC 中原来的值。

处理器看上去只是它的指令集结构的简单实现，但是实际上现代处理器使用了非常复杂的机制来加速程序的执行。因此，我们可以这样区分处理器的指令集结构和微体系结构：指令集结构

[⊖] PC 也普遍地被用来作为“个人计算机”的缩写。然而，两者之间的区别应该可以很清楚地从上下文中看出来。



描述的是每条机器代码指令的效果；而微体系统结构描述的是处理器实际上是如何实现的。第3章我们研究机器代码时考虑的是机器的指令集结构所提供的抽象性。第4章将更详细地介绍处理器实际上是如何实现的。

1.4.2 运行 hello 程序

前面简单描述了系统的硬件组成和操作，现在开始介绍当我们运行示例程序时到底发生了些什么。在这里我们必须省略很多细节稍后再做补充，但是从现在起我们将很满意这种整体上的描述。

初始时，外壳程序执行它的指令，等待我们输入一个命令。当我们在键盘上输入字符串“./hello”后，外壳程序将字符逐一读入寄存器，再把它存放到存储器中，如图 1-5 所示。

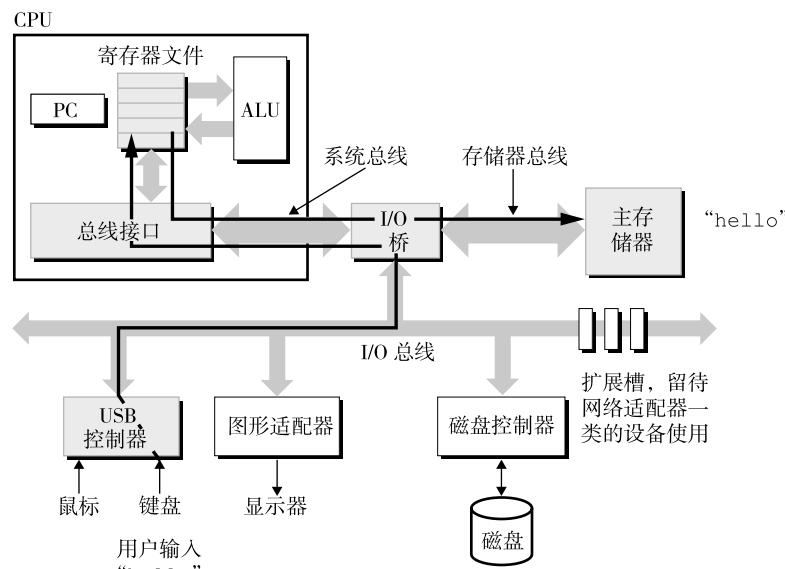


图 1-5 从键盘上读取 hello 命令

当我们在键盘上敲回车键时，外壳程序就知道我们已经结束了命令的输入。然后外壳执行一系列指令来加载可执行的 hello 文件，将 hello 目标文件中的代码和数据从磁盘复制到主存。数据包括最终会被输出的字符串“hello, world\n”。

利用直接存储器存取（DMA，将在第 6 章讨论）的技术，数据可以不通过处理器而直接从磁盘到达主存。这个步骤如图 1-6 所示。

一旦目标文件 hello 中的代码和数据被加载到主存，处理器就开始执行 hello 程序的 main 程序中的机器语言指令。这些指令将“hello, world\n”字符串中的字节从主存复制到寄存器文件，再从寄存器文件中复制到显示设备，最终显示在屏幕上。这个步骤如图 1-7 所示。

1.5 高速缓存至关重要

这个简单的示例揭示了一个重要的问题，即系统花费了大量的时间把信息从一个地方挪到另一个地方。hello 程序的机器指令最初是存放在磁盘上的，当程序加载时，它们被复制到主存；当处理器运行程序时，指令又从主存复制到处理器。相似地，数据串“hello, world\n”初始时在磁盘上，然后复制到主存，最后从主存上复制到显示设备。从程序员的角度来看，这些复制就是开销，减缓了程序“真正”的工作。因此，系统设计者的一个主要目标就是使这些复制操作尽可能快地完成。

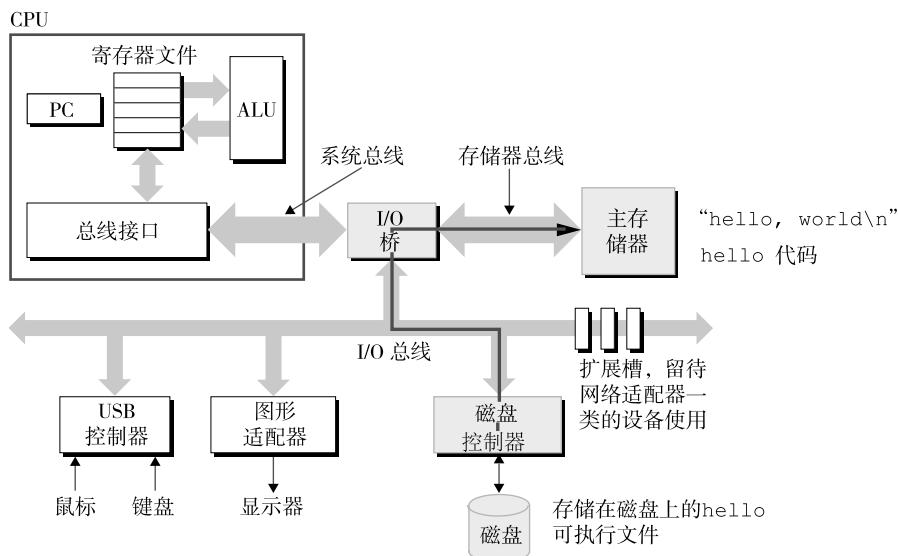


图 1-6 从磁盘加载可执行文件到主存

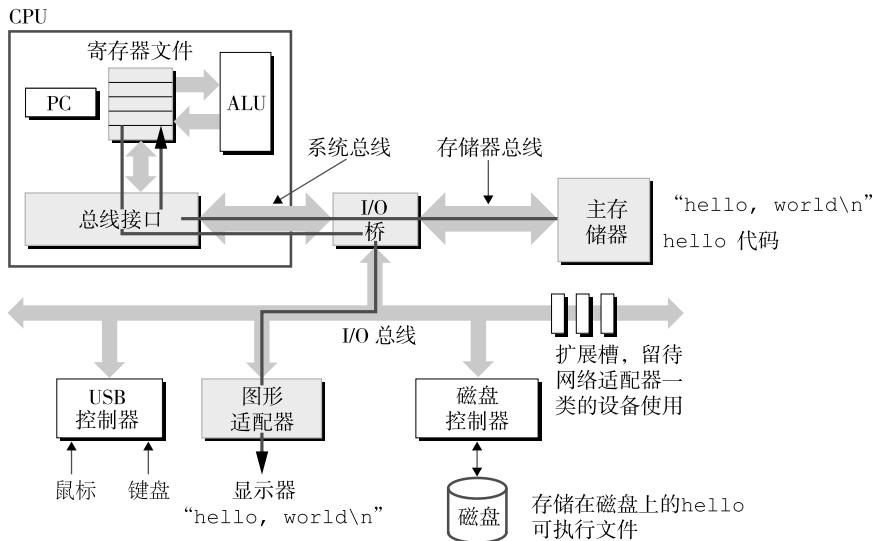


图 1-7 将输出字符串从内存写到显示器

根据机械原理，较大的存储设备要比较小的存储设备运行得慢，而快速设备的造价远高于同类的低速设备。例如，一个典型系统上的磁盘驱动器可能比主存大 1000 倍，但是对处理器而言，从磁盘驱动器上读取一个字的时间开销要比从主存中读取的开销大 1000 万倍。

类似地，一个典型的寄存器文件只存储几百字节的信息，而主存里可存放几十亿字节。然而，处理器从寄存器文件中读数据的速度比从主存中读取几乎要快 100 倍。更麻烦的是，随着这些年半导体技术的进步，这种处理器与主存之间的差距还在持续增大。加快处理器的运行速度比加快主存的运行速度要容易和便宜得多。

针对这种处理器与主存之间的差异，系统设计者采用了更小、更快的存储设备，即高速缓存存储器（简称高速缓存），作为暂时的集结区域，用来存放处理器近期可能会需要的信息。图 1-8 展示了一个典型系统中的高速缓存存储器。位于处理器芯片上的 L1 高速缓存的容量可以达

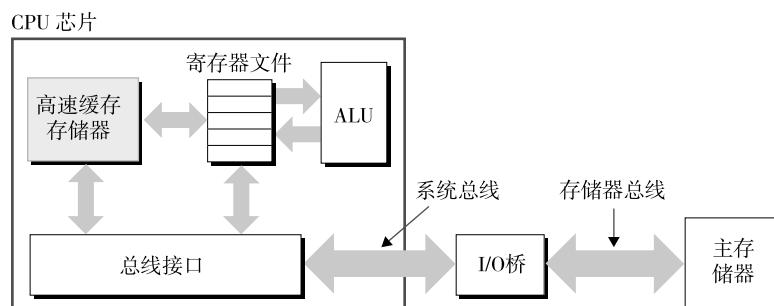


图 1-8 高速缓存存储器

到数万字节，访问速度几乎和访问寄存器文件一样快。一个容量为数十万到数百万字节的更大的 L2 高速缓存通过一条特殊的总线连接到处理器。进程访问 L2 高速缓存的时间要比访问 L1 高速缓存的时间长 5 倍，但是这仍然比访问主存的时间快 5~10 倍。L1 和 L2 高速缓存是用一种叫做静态随机访问存储器（SRAM）的硬件技术实现的。比较新的、处理能力更强大的系统甚至有三级高速缓存：L1、L2 和 L3。系统可以获得一个很大的存储器，同时访问速度也很快，原因是利用了高速缓存的局部性原理，即程序具有访问局部区域里的数据和代码的趋势。通过让高速缓存里存放可能经常访问的数据的方法，大部分的存储器操作都能在快速的高速缓存中完成。

本书得出的重要结论之一，就是意识到高速缓存存在的程序员可以利用高速缓存将他们程序的性能提高一个数量级。你将在第 6 章学习这些重要的设备以及如何利用它们。

1.6 存储设备形成层次结构

在处理器和一个又大又慢的设备（例如主存）之间插入一个更小更快的存储设备（例如高速缓存）的想法已经成为了一个普遍的观念。实际上，每个计算机系统中的存储设备都被组织成了一个存储器层次结构，如图 1-9 所示。在这个层次结构中，从上至下，设备变得访问速度越来越慢、容量越来越大，并且每字节的造价也越来越便宜。寄存器文件在层次结构中位于最顶部，也就是第 0 级或记为 L0。这里我们展示的是三层高速缓存 L1 到 L3，占据存储器层次结构的第 1

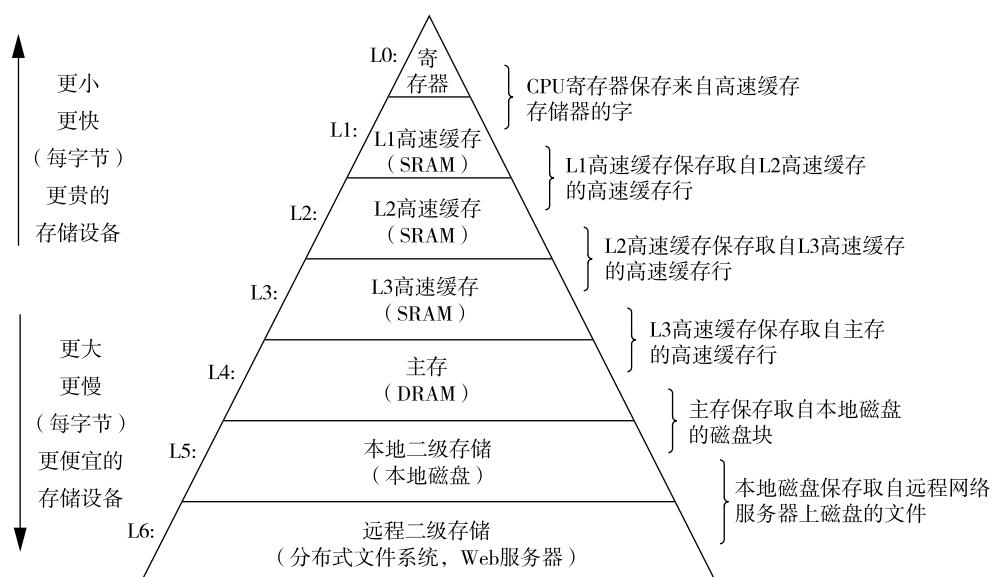


图 1-9 一个存储器层次结构的示例



层到第3层。主存在第4层，以此类推。

存储器层次结构的主要思想是一层上的存储器作为低一层存储器的高速缓存。因此，寄存器文件就是L1的高速缓存，L1是L2的高速缓存，L2是L3的高速缓存，L3是主存的高速缓存，而主存又是磁盘的高速缓存。在某些具有分布式文件系统的网络系统中，本地磁盘就是存储在其他系统中磁盘上的数据的高速缓存。

正如可以运用不同的高速缓存的知识来提高程序性能一样，程序员同样可以利用对整个存储器层次结构的理解来提高程序性能。第6章将更详细地讨论这个问题。

1.7 操作系统管理硬件

我们继续讨论hello程序的例子。当外壳加载和运行hello程序，以及hello程序输出自己的消息时，外壳和hello程序都没有直接访问键盘、显示器、磁盘或者主存。取而代之的是，它们依靠操作系统提供的服务。我们可以把操作系统看成是应用程序和硬件之间插入的一层软件，如图1-10所示。所有应用程序对硬件的操作尝试都必须通过操作系统。

操作系统有两个基本功能：1) 防止硬件被失控的应用程序滥用。2) 向应用程序提供简单一致的机制来控制复杂而又通常大相径庭的低级硬件设备。操作系统通过几个基本的抽象概念（进程、虚拟存储器和文件）来实现这两个功能。如图1-11所示，文件是对I/O设备的抽象表示，虚拟存储器是对主存和磁盘I/O设备的抽象表示，进程则是对处理器、主存和I/O设备的抽象表示。我们将依次讨论每种抽象表示。

Unix 和 Posix

20世纪60年代是大型、复杂操作系统盛行的年代，如IBM的OS/360和Honeywell的Multics系统。OS/360是历史上最成功的软件项目之一，而Multics虽然持续存在了多年，却从来没有被广泛应用。贝尔实验室曾经是Multics项目的最初参与者，但是考虑到该项目的复杂性和缺乏进展于1969年退出。鉴于Multics项目不愉快的经历，一组贝尔实验室的研究人员——Ken Thompson、Dennis Ritchie、Doug McIlroy和Joe Ossanna，从1969年开始在DEC PDP-7计算机上完全用机器语言编写了一个简单得多的操作系统。这个新系统中的很多思想，如层次文件系统、作为用户级进程的外壳概念，都是来自于Multics，只不过在一个更小、更简单的程序包里实现。1970年，Brian Kernighan给新系统命名为“Unix”，这也是一个双关语，暗指“Multics”的复杂性。1973年用C语言重新编写其内核，1974年，Unix开始正式对外发布[89]。

贝尔实验室以慷慨的条件向学校提供源代码，所以Unix在大专院校里获得了很多支持并继续发展。最有影响的工作是20世纪70年代晚期到80年代早期，在美国加州大学伯克利分校，伯克利的研究人员在一系列发布版本中增加了虚拟存储器和Internet协议，称为Unix 4.xBSD(Berkeley Software Distribution)。与此同时，贝尔实验室也在发布自己的版本，即System V Unix。其他厂商的版本，如Sun Microsystems的Solaris系统，则是从这些最初的BSD和System

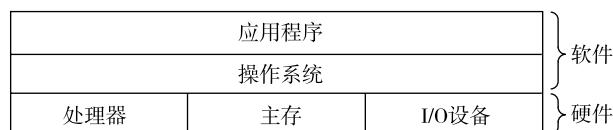


图1-10 计算机系统的分层视图

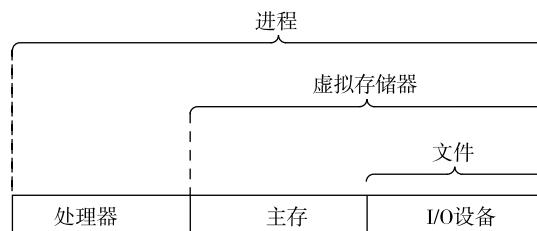


图1-11 操作系统提供的抽象表示



V 版本中衍生而来。

20世纪80年代中期，Unix厂商试图通过加入新的、往往不兼容的特性来使它们的程序与众不同，麻烦也就随之而来了。为了阻止这种趋势，IEEE（电气和电子工程师协会）开始努力标准化Unix的开发，后来由Richard Stallman命名为“Posix”。结果就得到了一系列的标准，称做Posix标准。这套标准涵盖了很多方面，比如Unix系统调用的C语言接口、外壳程序和工具、线程及网络编程。随着越来越多的系统日益完全地遵从Posix标准，Unix版本之间的差异正在逐渐消失。

1.7.1 进程

像hello这样的程序在现代系统上运行时，操作系统会提供一种假象，就好像系统上只有这个程序在运行，看上去只有这个程序在使用处理器、主存和I/O设备。处理器看上去就像在不间断地一条接一条地执行程序中的指令，即该程序的代码和数据是系统存储器中唯一的对象。这些假象是通过进程的概念来实现的，进程是计算机科学中最重要和最成功的概念之一。

进程是操作系统对一个正在运行的程序的一种抽象。在一个系统上可以同时运行多个进程，而每个进程都好像在独占地使用硬件。而并发运行，则是说一个进程的指令和另一个进程的指令是交错执行的。在大多数系统中，需要运行的进程数是多于可以运行它们的CPU个数的。传统系统在一个时刻只能执行一个程序，而先进的多核处理器同时能够执行多个程序。无论是在单核还是多核系统中，一个CPU看上去都像是在并发地执行多个进程，这是通过处理器在进程间切换来实现的。操作系统实现这种交错执行的机制称为上下文切换。为了简化讨论，我们只考虑包含一个CPU的单处理器系统的情况。我们会在1.9.1节讨论多处理器系统。

操作系统保持跟踪进程运行所需的所有状态信息。这种状态，也就是上下文，它包括许多信息，例如PC和寄存器文件的当前值，以及主存的内容。在任何一个时刻，单处理器系统都只能执行一个进程的代码。当操作系统决定要把控制权从当前进程转移到某个新进程时，就会进行上下文切换，即保存当前进程的上下文、恢复新进程的上下文，然后将控制权传递到新进程。新进程就会从上次停止的地方开始。图1-12展示了示例hello程序运行场景的基本理念。

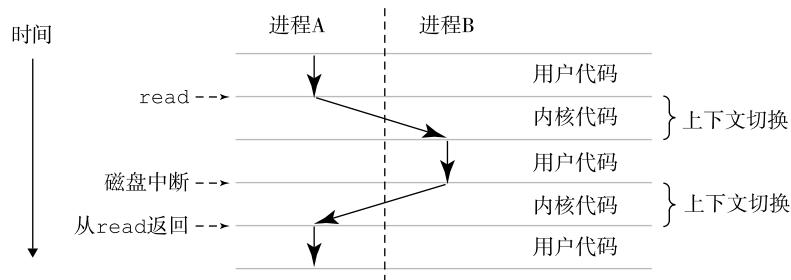


图1-12 进程的上下文切换

示例场景中有两个并发的进程：外壳进程和hello进程。起初，只有外壳进程在运行，即等待命令行上的输入。当我们让它运行hello程序时，外壳通过调用一个专门的函数，即系统调用，来执行我们的请求，系统调用会将控制权传递给操作系统。操作系统保存外壳进程的上下文，创建一个新的hello进程及其上下文，然后将控制权传递给新的hello进程。hello进程终止后，操作系统恢复外壳进程的上下文，并将控制权传回给它，外壳进程将继续等待下一个命令行输入。

实现进程这个抽象概念需要低级硬件和操作系统软件之间的紧密合作。我们将在第8章揭示这项工作的原理，以及应用程序是如何创建和控制它们的进程的。



1.7.2 线程

尽管通常我们认为一个进程只有单一的控制流，但是在现代系统中，一个进程实际上可以由多个称为线程的执行单元组成，每个线程都运行在进程的上下文中，并共享同样的代码和全局数据。由于网络服务器对并行处理的需求，线程成为越来越重要的编程模型，因为多线程之间比多进程之间更容易共享数据，也因为线程一般来说都比进程更高效。当有多处理器可用的时候，多线程也是一种使程序可以更快运行的方法，我们将在1.9.1节讨论这个问题。你将在第12章学习到并发的基本概念，以及如何写线程化的程序。

1.7.3 虚拟存储器

虚拟存储器是一个抽象概念，它为每个进程提供了一个假象，即每个进程都在独占地使用主存。每个进程看到的是一致的存储器，称为虚拟地址空间。图1-13所示的是Linux进程的虚拟地址空间（其他Unix系统的设计也与此类似）。在Linux中，地址空间最上面的区域是为操作系统中的代码和数据保留的，这对所有进程来说都是一样的。地址空间的底部区域存放用户进程定义的代码和数据。请注意，图中的地址是从下往上增大的。

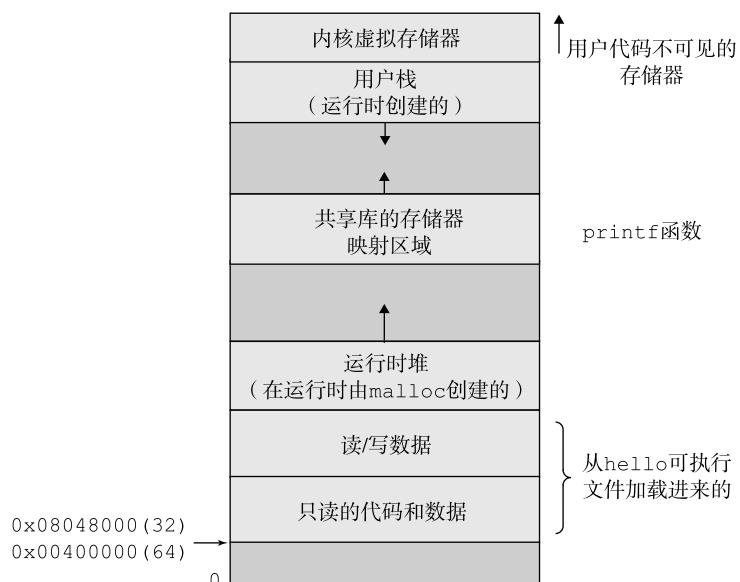


图1-13 进程的虚拟地址空间

每个进程看到的虚拟地址空间由大量准确定义的区构成，每个区都有专门的功能。本书的后续章节将介绍更多的有关这些区的知识，但是先简单了解每一个区将是非常有益的。我们是从最低的地址开始，逐步向上介绍。

- 程序代码和数据。对于所有的进程来说，代码是从同一固定地址开始，紧接着的是和C全局变量相对应的数据位置。代码和数据区是直接按照可执行目标文件的内容初始化的，在示例中就是可执行文件hello。第7章我们研究链接和加载，你将会学习到更多有关地址空间的内容。
- 堆。代码和数据区后紧随着的是运行时堆。代码和数据区是在进程一开始运行时就被规定了大小，与此不同，当调用如malloc和free这样的C标准库函数时，堆可以在运行时动态地扩展和收缩。第9章学习管理虚拟存储器时，我们将更详细地研究堆。
- 共享库。大约在地址空间的中间部分是一块用来存放像C标准库和数学库这样共享库的代码和数据的区域。共享库的概念非常强大，也相当难懂。第7章介绍动态链接时，我们将



学习共享库是如何工作的。

- **栈**。位于用户虚拟地址空间顶部的是用户栈，编译器用它来实现函数调用。和堆一样，用户栈在程序执行期间可以动态地扩展和收缩。特别是每次我们调用一个函数时，栈就会增长；从一个函数返回时，栈就会收缩。在第3章，你将学习编译器是如何使用栈的。
- **内核虚拟存储器**。内核总是驻留在内存中，是操作系统的一部分。地址空间顶部的区域是为内核保留的，不允许应用程序读写这个区域的内容或者直接调用内核代码定义的函数。

虚拟存储器的运作需要硬件和操作系统软件之间精密复杂的交互，包括对处理器生成的每个地址的硬件翻译。其基本思想是把一个进程虚拟存储器的内容存储在磁盘上，然后用主存作为磁盘的高速缓存。第9章将解释虚拟存储器如何工作，以及它为什么对现代系统的运行如此重要。

1.7.4 文件

文件就是字节序列，仅此而已。每个I/O设备，包括磁盘、键盘、显示器，甚至网络，都可以视为文件。系统中的所有输入输出都是通过使用一小组称为Unix I/O的系统函数调用读写文件来实现的。

文件这个简单而精致的概念其内涵是非常丰富的，因为它向应用程序提供了一个统一的视角，来看待系统中可能含有的所有各式各样的I/O设备。例如，处理磁盘文件内容的应用程序员非常幸福，因为他们无需了解具体的磁盘技术。进一步说，同一个程序可以在使用不同磁盘技术的不同系统上运行。第10章将介绍Unix I/O。

Linux项目

1991年8月，芬兰研究生Linus Torvalds谨慎地发布了一个新的类Unix的操作系统内核，内容如下：

来自：torvalds@klaava.Helsinki.FI (Linus Benedict Torvalds)

新闻组：comp.os.minix

主题：在minix中你最想看到什么？

摘要：关于我的新操作系统的小调查

时间：1991年8月25日20:57:08格林尼治时间

每个使用minix的朋友，你们好。

我正在做一个（免费的）用在386（486）AT上的操作系统（只是业余爱好，它不会像GNU那样庞大和专业）。这个想法从4月份起就开始酝酿，现在快要完成了。我希望得到各位对minix的任何反馈意见，因为我的操作系统在某些方面是与它相类似的（其中包括相同的文件系统的物理设计（因为某些实际的原因））。

我已经移植了bash（1.08）和gcc（1.40），并且看上去能运行。这意味着我需要用几个月的时间使它变得更实用一些，并且我想知道大多数人想要的特性。欢迎提出任何建议，但是我无法保证都能实现。:-)

Linus (torvalds@kruuna.helsinki.fi)

接下来，如他们所说，这就成为了历史。Linux逐渐发展成为一个技术和文化现象。通过结合GNU项目的力量，Linux项目发展成为一个完整的、符合Posix标准的Unix操作系统的版本，包括内核和所有支撑的基础设施。从手持设备到大型计算机，Linux在范围如此广泛的计算机上得到了应用。IBM的一个工作组甚至把Linux移植到了一块腕表中！

1.8 系统之间利用网络通信

系统漫游至此，我们一直是把系统视为一个孤立的硬件和软件的集合体。实际上，现代系



统经常通过网络和其他系统连接到一起。从一个单独的系统来看，网络可视为一个 I/O 设备，如图 1-14 所示。当系统从主存将一串字节复制到网络适配器时，数据流经过网络到达另一台机器，而不是其他地方，例如本地磁盘驱动器。相似地，系统可以读取从其他机器发送来的数据，并把数据复制到自己的主存。

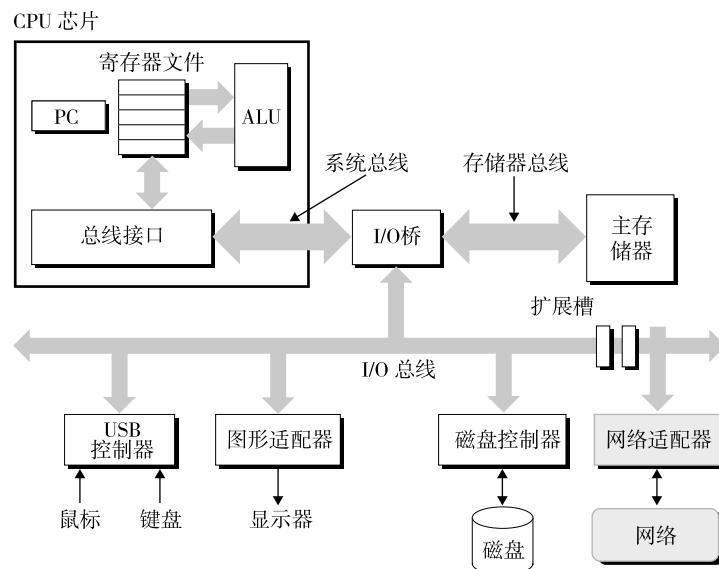


图 1-14 网络也是一种 I/O 设备

随着 Internet 这样的全球网络的出现，将一台主机的信息复制到另外一台主机已经成为计算机系统最重要的用途之一。例如，电子邮件、即时通信、万维网、FTP 和 telnet 这样的应用都是基于网络复制信息的功能。

继续讨论我们的 hello 示例，我们可以使用熟悉的 telnet 应用在一个远程主机上运行 hello 程序。假设用本地主机上的 telnet 客户端连接远程主机上的 telnet 服务器。在我们登录到远程主机并运行外壳后，远端的外壳就在等待接收输入命令。从这点开始，远程运行 hello 程序包括（如图 1-15 所示）的五个基本步骤。

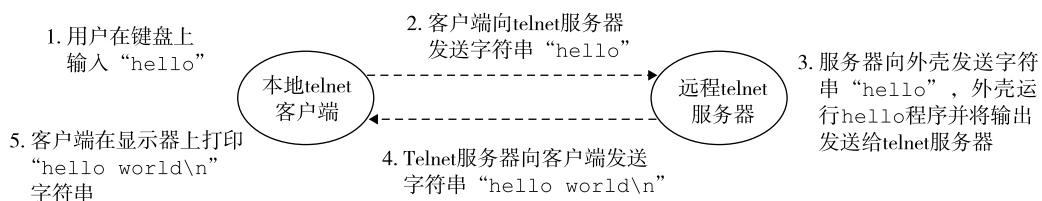


图 1-15 利用 telnet 通过网络远程运行 hello 程序

当我们在 telnet 客户端键入“hello”字符串并敲下回车键后，客户端软件就会将这个字符串发送到 telnet 的服务器。telnet 服务器从网络上接收到这个字符串后，会把它传递给远端外壳程序。接下来，远端外壳运行 hello 程序，并将输出行返回给 telnet 服务器。最后，telnet 服务器通过网络把输出串转发给 telnet 客户端，客户端就将输出串输出到我们的本地终端上。

这种客户端和服务器之间交互的类型在所有的网络应用中是非常典型的。在第 11 章，你将学到如何构造网络应用程序，并利用这些知识创建一个简单的 Web 服务器。



1.9 重要主题

在此，总结一下我们旋风式的系统漫游。这次讨论得出一个很重要的观点，那就是系统不仅仅只是硬件。系统是硬件和系统软件互相交织的集合体，它们必须共同协作以达到运行应用程序的最终目的。本书的余下部分会讲述硬件和软件的详细内容，通过了解这些详细内容，你可以写出更快速、更可靠和更安全的程序。

我们在此强调几个贯穿计算机系统所有方面的重要概念作为本章的结束。我们还会在本书中的多处讨论这些概念的重要性。

1.9.1 并发和并行

数字计算机的整个历史中，有两个需求是驱动进步的持续动力：一个是我们想要计算机做得更多，另一个是我们想要计算机运行得更快。当处理器同时能够做更多事情时，这两个因素都会改进。我们用的术语并发（concurrency）是一个通用的概念，指一个同时具有多个活动的系统；而术语并行（parallelism）指的是用并发使一个系统运行得更快。并行可以在计算机系统的多个抽象层次上运用。在此，我们按照系统层次结构中由高到低的顺序重点强调三个层次。

1. 线程级并发

构建进程这个抽象，我们能够设计出同时执行多个程序的系统，这就导致了并发。使用线程，我们甚至能够在一个进程中执行多个控制流。从20世纪60年代初期出现时间共享以来，计算机系统中就开始有了对并发执行的支持。传统意义上，这种并发执行只是模拟出来的，是通过使一台计算机在它正在执行的进程间快速切换的方式实现的，就好像一个杂技演员保持多个球在空中飞舞。这种并发形式允许多个用户同时与系统交互，例如，当许多人想要从一个Web服务器获取页面时。它还允许一个用户同时从事多个任务，例如，在一个窗口中开启Web浏览器，在另一窗口中运行字处理器，同时又播放音乐。在以前，即使处理器必须在多个任务间切换，大多数实际的计算也都是由一个处理器来完成的。这种配置称为单处理器系统。

当构建一个由单操作系统内核控制的多处理器组成的系统时，我们就得到了一个多处理器系统。其实从20世纪80年代开始，在大规模的计算中就采用了这种系统，但是直到最近，随着多核处理器和超线程（hyperthreading）的出现，这种系统才变得常见。图1-16列出了这些不同处理器类型的分类。

多核处理器是将多个CPU（称为“核”）集成到一个集成电路芯片上。图1-17描述的是Intel Core i7处理器的组织结构，其中微处理器芯片有4个CPU核，每个核都有自己的L1和L2高速缓存，但是它们共享更高层次的高速缓存，以及到主存的接口。工业界的专家预言他们能够将几十个、最终会是上百个核做到一个芯片上。

超线程，有时称为同时多线程（simultaneous multi-threading），是一项允许一个CPU执行多个控制流的技术。它涉及CPU某些硬件有多个备份，比如程序计数器和寄存器文件；而其他的硬件部分只有一份，比如执行浮点算术运算的单元。常规的处理器需要大约20 000个时钟周期做不同线程间的转换，而超线程的处理器可以在单个周期的基础上决定要执行哪一个线程。这使得CPU能够更好地利用它的处理资源。例如，假设一个线程必须等到某些数据被装载到高速缓存中，那CPU就可以继续去执行另一个线程。举例来说，Intel Core i7处理器可以让一个核执行两个线程，所以一个4核的系统实际上可以并行地执行8个线程。

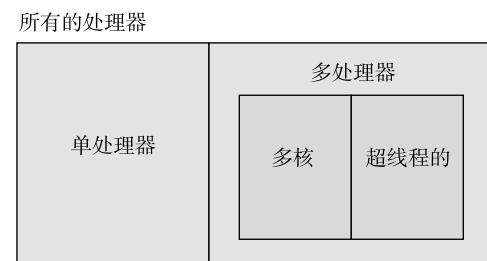


图1-16 不同的处理器配置分类。随着多核处理器和超线程的出现，多处理器变得普遍了

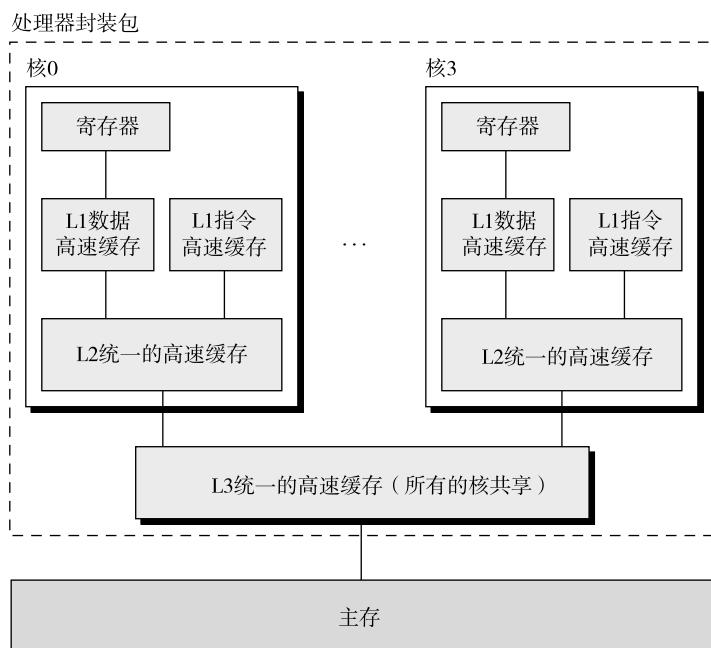


图 1-17 Intel Core i7 的组织结构。4 个处理器核集成到一个芯片上

多处理器的使用可以从两个方面提高系统性能。首先，它减少了在执行多个任务时模拟并发的需要。正如前面提到的，即使是只有一个用户使用的个人计算机也需要并发地执行多个活动。其次，它可以使应用程序运行得更快。当然，这必须要求程序是以多线程方式来书写的，这些线程可以并行地高效执行。因此，虽然并发原理的形成和研究已经超过 50 年的时间了，但是直到多核和超线程系统的出现才极大地激发了人们的一种愿望，即找到书写应用程序的方法利用硬件开发线程级并行性。第 12 章将更深入地探讨并发，以及使用并发来提供处理器资源的共享，使得程序的执行允许有更多的并行。

2. 指令级并行

在较低的抽象层次上，现代处理器可以同时执行多条指令的属性称为指令级并行。早期的微处理器，如 1978 年的 Intel 8086，需要多个（通常是 3 ~ 10 个）时钟周期来执行一条指令。比较先进的处理器可以保持每个时钟周期 2 ~ 4 条指令的执行速率。其实每条指令从开始到结束需要长得多的时间，大约 20 个或者更多的周期，但是处理器使用了非常多的聪明技巧来同时处理多达 100 条的指令。在第 4 章，我们将研究流水线（pipelining）的使用。在流水线中，将执行一条指令所需要的活动划分成不同的步骤，将处理器的硬件组织成一系列的阶段，每个阶段执行一个步骤。这些阶段可以并行地操作，用来处理不同指令的不同部分。我们会看到一个相当简单的硬件设计，它能够达到接近于一个时钟周期一条指令的执行速率。

如果处理器可以达到比一个周期一条指令更快的执行速率，就称之为超标量（superscalar）处理器。大多数现代处理器都支持超标量操作。第 5 章，将介绍超标量处理器的高级模型。应用程序员可以用这个模型来理解他们程序的性能。然后，他们就能写出拥有更高程度的指令级并行性的程序代码，因而也运行得更快。

3. 单指令、多数据并行

在最低层次上，许多现代处理器拥有特殊的硬件，允许一条指令产生多个可以并行执行的操作，这种方式称为单指令、多数据，即 SIMD 并行。例如，较新的 Intel 和 AMD 处理器都具有



并行地对 4 对单精度浮点数（C 数据类型 float）做加法的指令。

提供这些 SIMD 指令多是为了提高处理影像、声音和视频数据应用的执行速度。虽然有些编译器试图从 C 程序中自动抽取 SIMD 并行性，但是更可靠的方法是使用编译器支持的特殊向量数据类型来写程序，例如 GCC 就支持向量数据类型。作为对比较通用的程序优化讲述的补充，在网络旁注 OPT:SIMD 中描述了这种编程方式。

1.9.2 计算机系统中抽象的重要性

抽象的使用是计算机科学中最为重要的概念之一。例如，为一组函数规定一个简单的应用程序接口（API）就是一个很好的编程习惯，程序员无需了解它内部的工作便可以使用这些代码。不同的编程语言提供不同形式和等级的抽象支持，例如 Java 类的声明和 C 语言的函数原型。

我们已经介绍了计算机系统中使用的几个抽象，如图 1-18 所示。在处理器里，指令集结构提供了对实际处理器硬件的抽象。使用这个抽象，机器代码程序表现得就好像它是运行在一个一次只执行一条指令的处理器上。底层的硬件比抽象描述的要复杂精细得多，它并行地执行多条指令，但又总是与那个简单有序的模型保持一致。只要执行模型一样，不同的处理器实现也能执行同样的机器代码，而又提供不同的开销和性能。

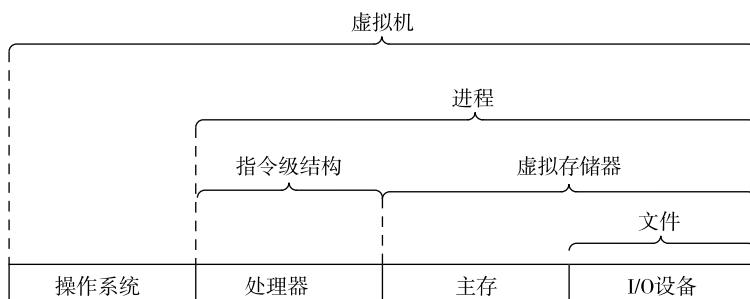


图 1-18 计算机系统提供的一些抽象

注：计算机系统中的一个重大主题就是提供不同层次的抽象表示，来隐藏实际实现的复杂性。

在学习操作系统时，我们介绍了三个抽象：文件是对 I/O 的抽象，虚拟存储器是对程序存储器的抽象，而进程是对一个正在运行的程序的抽象。我们再增加一个新的抽象：虚拟机，它提供对整个计算机（包括操作系统、处理器和程序）的抽象。虚拟机的思想是 IBM 在 20 世纪 60 年代提出来的，但是最近才显示出其管理计算机方式上的优势，因为一些计算机必须能够运行为不同操作系统（例如，Microsoft Windows、MacOS 和 Linux）或同一操作系统的不同版本而设计的程序。

在本书后续的章节中，我们会具体介绍这些抽象。

1.10 小结

计算机系统是由硬件和系统软件组成的，它们共同协作以运行应用程序。计算机内部的信息被表示为一组组的位，它们依据上下文有不同的解释方式。程序被其他程序翻译成不同的形式，开始时是 ASCII 文本，然后被编译器和链接器翻译成二进制可执行文件。

处理器读取并解释存放在主存里的二进制指令。因为计算机把大量的时间用于存储器、I/O 设备和 CPU 寄存器之间复制数据，所以将系统中的存储设备划分成层次结构——CPU 寄存器在顶部，接着是多层的硬件高速缓存存储器、DRAM 主存和磁盘存储器。在层次模型中，位于更



高层的存储设备比低层的存储设备要更快，单位比特开销也更高。层次结构中较高层次存储设备可以作为较低层次设备的高速缓存。通过理解和运用这种存储层次结构的知识，程序员可以优化 C 程序的性能。

操作系统内核是应用程序和硬件之间的媒介。它提供三个基本的抽象：1) 文件是对 I/O 设备的抽象；2) 虚拟存储器是对主存和磁盘的抽象；3) 进程是对处理器、主存和 I/O 设备的抽象。

最后，网络提供了计算机系统之间通信的手段。从特殊系统的角度来看，网络就是一种 I/O 设备。

参考文献说明

Ritchie 写了关于早期 C 和 Unix 的有趣的第一手资料 [87, 88]。Ritchie 和 Thompson 提供了最早出版的 Unix 资料 [89]。Silberschatz、Gavin 和 Gagne[98] 提供了关于 Unix 不同版本的详尽历史。GNU (www.gnu.org) 和 Linux (www.linux.org) 的网站上有大量的当前信息和历史资料。Posix 标准可以在线获得 (www.unix.org)。





| 第一部分 |
Computer Systems : A Programmer's Perspective, 2E

程序结构和执行

我们对计算机系统的探索是从学习计算机本身开始的，它由处理器和存储器子系统组成。在核心部分，我们需要方法来表示基本数据类型，比如整数和实数运算的近似值。然后，我们考虑机器级指令如何操作这样的数据，以及编译器如何将 C 程序翻译成这样的指令。接下来，研究几种实现处理器的方法，帮助我们更好地了解硬件资源是如何被用来执行指令。一旦理解了编译器和机器级代码，我们就能通过编写最高性能的 C 程序，来分析如何最大化程序的性能。本部分以存储器子系统的设计作为结束，这是现代计算机系统最复杂的部分之一。

本书的这一部分将引领着你深入了解如何表示和执行应用程序。你将学会一些技巧，它们将帮助你写出安全、可靠且充分利用计算资源的程序。



第2章 |

Computer Systems : A Programmer's Perspective, 2E

信息的表示和处理

现代计算机存储和处理的信息以二值信号表示。这些微不足道的二进制数字，或者称为位（bit），奠定了数字革命的基础。大家熟悉并且使用了1000多年的十进制（以十为基数）起源于印度，12世纪被阿拉伯数学家改进，并在13世纪被意大利数学家Leonardo Pisano（公元1170-1250，更为大家所熟知的名字是Fibonacci）带到西方。对于有10个手指的人类来说，使用十进制表示法是很自然的事情，但是当构造存储和处理信息的机器时，二进制的值工作得更好。二值信号能够很容易地被表示、存储和传输，例如，可以表示为穿孔卡片上有洞或无洞、导线上的高电压或低电压，或者顺时针或逆时针的磁场。对二值信号进行存储和执行计算的电子电路非常简单和可靠，制造商能够在一个单独的硅片上集成数百万甚至数十亿个这样的电路。

孤立地讲，单个的位不是非常有用。然而，当把位组合在一起，再加上某种解释（interpretation），即给不同的可能位模式赋予含义，我们就能够表示任何有限集合的元素。比如，使用一个二进制数字系统，我们能够用位组来编码非负数。通过使用标准的字符码，我们能够对文档中的字母和符号进行编码。在本章中，我们将讨论这两种编码，以及表示负数和对实数近似值的编码。

我们研究三种最重要的数字表示。无符号（unsigned）编码基于传统的二进制表示法，表示大于或者等于零的数字。补码（two's-complement）编码是表示有符号整数的最常见的方式，有符号整数就是可以为正或者为负的数字。浮点数（floating-point）编码是表示实数的科学记数法的以二为基数的版本。计算机用这些不同的表示方法实现算术运算，例如加法和乘法，类似于对应的整数和实数运算。

计算机的表示法是用有限数量的位来对一个数字编码，因此，当结果太大以至不能表示时，某些运算就会溢出（overflow）。溢出会导致某些令人吃惊的后果。例如，现在的大多数计算机（使用32位来表示数据类型int），计算表达式 $200*300*400*500$ 会得出结果-884 901 888。这违背了整数运算的特性，计算一组正数的乘积不应产生一个为负的结果。

另一方面，整数的计算机运算满足人们所熟知的真正整数运算的定律。例如，利用乘法的结合律和交换律，计算下面任何一个C表达式，都会得出结果-884 901 888：

```
(500*400)*(300*200)
((500*400)*300)*200
((200*500)*300)*400
400*(200*(300*500))
```

计算机可能没有产生期望的结果，但是至少结果是一致的！

浮点运算有完全不同的数学属性。虽然溢出会产生特殊的值 $+\infty$ ，但是一组正数的乘积总是正的。由于表示的精度有限，浮点运算是不可结合的。例如，在大多数机器上，C表达式 $(3.14+1e20)-1e20$ 求得的值会是0.0，而 $3.14+(1e20-1e20)$ 求得的值会是3.14。整数运算和浮点数运算会有不同的数学属性是因为它们处理数字表示有限性的方式不同——整数的表示虽然只能编码一个相对较小的数值范围，但是这种表示是精确的；而浮点数虽然可以编码一个较大的数值范围，但是这种表示只是近似的。

通过研究数字的实际表示，我们能够理解可以表示的值的范围和不同算术运算的属性。为了使编写的程序能在全部数值范围内正确工作，而且具有可以跨越不同机器、操作系统和编译器组



合的可移植性，了解这些属性是非常重要的。后面我们会讲到，大量计算机的安全漏洞都是由于计算机算术运算的微妙细节引发的。在早期，当人们碰巧触发了程序漏洞，只会给人们带来一些不便；但是现在，有许多的黑客企图利用他们能找到的任何漏洞，不经过授权就进入他人的系统。这就要求程序员有更多的责任和义务，去了解他们的程序如何工作，以及如何被迫产生不良的行为。

计算机用几种不同的二进制表示形式来编码数值。第3章随着进入机器级编程，你需要熟悉这些表示方式。在本章中，我们描述这些编码，并且教你如何推出数字的表示。

通过直接操作数字的位级表示，我们得到了几种进行算术运算的方式。理解这些技术对于理解编译器产生的机器级代码是很重要的，编译器会试图优化算术表达式求值的性能。

我们对这部分内容的处理是基于一组核心的数学原理的。我们从编码的基本定义开始，然后得出一些属性，例如可表示的数字的范围、它们的位级表示以及算术运算的属性。相信从这样一个抽象的观点来分析这些内容，对你来说是很重要的，因为程序员需要对计算机运算与更为人熟悉的整数和实数运算之间的关系有清晰的理解。

怎样阅读本章

如果你觉得等式和公式令人生畏，不要让它阻止你学习本章的内容！为了内容的完整性，我们提供全部的数学概念的推导，但是阅读这些内容的最好方法是在你首次阅读时跳过这些推导。但是，要研究我们提供的例题，并且要做完所有的练习题。这些例题会让你对概念有一些感性的认识，并且练习题让你能够主动学习，帮助你理论联系实际。有了这些例题和练习题作为背景知识，再回头看那些推导，你会发现理解起来会容易许多。同时，请放心，每个掌握了高中代数知识的人都具备理解这些内容所需要的数学技能。

C++ 编程语言建立在 C 语言的基础之上，它们使用完全相同的数字表示和运算。本章中关于 C 的所有内容对 C++ 都有效。另一方面，Java 语言创造了一套新的数字表示和运算标准。C 标准的设计允许多种实现方式，而 Java 标准在数据的格式和编码上是非常精确具体的。本章中多处着重介绍了 Java 支持的表示和运算。

C 编程语言的演变

前面提到过，C 编程语言是贝尔实验室的 Dennis Ritchie 最早开发出来的，目的是和 Unix 操作系统一起使用（Unix 也是贝尔实验室开发的）。在那个时候，大多数系统程序，例如操作系统，为了访问不同数据类型的低级表示，都必须用大量的汇编代码编写。比如说，像 malloc 库函数提供的内存分配那样的功能，用当时的其他高级语言是无法编写的。

Brian Kernighan 和 Dennis Ritchie 的著作的第1版 [57] 记录了最初贝尔实验室的 C 语言版本。随着时间的推移，经过多个标准化组织的努力，C 语言也在不断地演变。1989 年，美国国家标准学会下的一个工作组推出了 ANSI C 标准，对最初的贝尔实验室的 C 语言做了重大修改。ANSI C 与贝尔实验室的 C 有了很大的不同，尤其是函数声明的方式。Brian Kernighan 和 Dennis Ritchie 在著作的第2版 [58] 中描述了 ANSI C，这本书至今仍然被公认为是关于 C 语言最好的参考手册之一。

国际标准化组织接管了对 C 语言进行标准化的任务，在 1990 年推出了几乎和 ANSI C 一样的版本，称为“ISO C90”。该组织在 1999 年又对 C 语言做了更新，得到“ISO C99”。这一版本，引入了一些新的数据类型，对使用不符合英语语言字符的文本字符串提供了支持。

GNU 编译器套装（GNU Compiler Collection，GCC）可以基于不同的命令行选项，依照多个不同版本的 C 语言规则来编译程序，如图 2-1 所示。例如，根据 ISO C99 来编译程序 prog.c，



我们就使用命令行：

```
unix> gcc -std=c99 prog.c
```

编译选项 `-ansi` 和 `-std=c89` 的用法是一样的——会根据 ANSI 或者 ISO C90 标准来编译程序。(C90 有时也称为“C89”，因为它的标准化工作是从 1989 年开始的。) 编译选项 `-std=c99` 会让编译器按照 ISO C99 的规则进行编译。

本书中由于没有指定任何编译选项，程序会按照基于 ISO C90 的 C 语言版本进行编译，但是又包括一些 C99 的特性，一些 C++ 的特性，还有一些是与 GCC 相关的特性。可以显式地用编译选项 `-std=gnu89` 来指定这个 ISO C90 版本。GNU 项目正在开发一个结合了 ISO C99 和其他一些特性的版本，可以通过命令行选项 `-std=gnu99` 来指定。(目前，这个实现还未完成。) 以后，这个版本会成为默认的版本。

2.1 信息存储

大多数计算机使用 8 位的块，或者字节 (byte)，作为最小的可寻址的存储器单位，而不是在存储器中访问单独的位。机器级程序将存储器视为一个非常大的字节数组，称为虚拟存储器 (virtual memory)。存储器的每个字节都由一个唯一的数字来标识，称为它的地址 (address)，所有可能地址的集合称为虚拟地址空间 (virtual address space)。顾名思义，这个虚拟地址空间只是一个展现给机器级程序的概念性映像。实际的实现 (见第 9 章) 是将随机访问存储器 (RAM)、磁盘存储器、特殊硬件和操作系统软件结合起来，为程序提供一个看上去统一的字节数组。

C 版本	GCC 命令行选项
GNU 89	无, <code>-std=gnu89</code>
ANSI, ISO C90	<code>-ansi, -std=c89</code>
ISO C99	<code>-std=c99</code>
GNU 99	<code>-std=gnu99</code>

图 2-1 向 GCC 指定不同的 C 语言版本

接下来的几章，我们将讲述编译器和运行时系统是如何将存储器空间划分为更可管理的单元，以存放不同的程序对象 (program object)，即程序数据、指令和控制信息。可以用各种机制来分配和管理程序不同部分的存储。这种管理完全是在虚拟地址空间里完成的。例如，C 语言中一个指针的值 (无论它是指向一个整数、一个结构或是某个其他程序对象) 都是某个存储块的第一个字节的虚拟地址。C 编译器还把每个指针和类型信息联系起来，这样就可以根据指针值的类型，生成不同的机器级代码来访问存储在指针所指向位置处的值。尽管 C 编译器维护着这个类型信息，但是它生成的实际机器级程序并不包含关于数据类型的信息。每个程序对象可以简单地视为一个字节块，那么程序本身就是一个字节序列。

给 C 语言初学者：C 语言中指针的角色

指针是 C 语言的一个重要特性。它提供了引用数据结构 (包括数组) 的元素的机制。与变量类似，指针也有两个方面：值和类型。它的值表示某个对象的位置，而它的类型表示那个位置上所存储对象的类型 (比如整数或者浮点数)。

2.1.1 十六进制表示法

一个字节由 8 位组成。在二进制表示法中，它的值域是 $00000000_2 \sim 11111111_2$ ；如果用十进制整数表示，它的值域就是 $0_{10} \sim 255_{10}$ 。两种表示法对于描述位模式来说都不是非常方便。二进制表示法太冗长，而十进制表示法与位模式的互相转化又很麻烦。替代的方法是，以 16 为基数，或者叫十六进制 (hexadecimal) 数，来表示位模式。十六进制 (简写为“hex”) 使用数字‘0’～‘9’，以及字符‘A’～‘F’来表示 16 个可能的值。图 2-2 展示了 16 个十六进制数字对应的十进制值和二进制值。用十六进制书写，一个字节的值域为 $00_{16} \sim FF_{16}$ 。



在C语言中，以`0x`或`0X`开头的数字常量被认为是十六进制的值。字符‘A’～‘F’既可以是大写，也可以是小写，甚至是大小写混合。例如，我们可以将数字`FA1D37B16`写作`0xFA1D37B`，或者`0xfald37b`，也可以写作`0xFa1D37b`。在本书中，我们将使用C表示法来表示十六进制值。

十六进制数字	0	1	2	3	4	5	6	7
十进制值	0	1	2	3	4	5	6	7
二进制值	0000	0001	0010	0011	0100	0101	0110	0111
十六进制数字	8	9	A	B	C	D	E	F
十进制值	8	9	10	11	12	13	14	15
二进制值	1000	1001	1010	1011	1100	1101	1110	1111

图 2-2 十六进制表示法。每个十六进制数字都对 16 个值中的一个进行了编码

编写机器级程序的一个常见任务就是在位模式的十进制、二进制和十六进制表示之间人工进行转换。二进制和十六进制之间的转换比较简单直接，因为可以一次执行一个十六进制数字的转换。数字的转换可以参考如图 2-2 所示的表。一个简单的窍门是，记住十六进制数字 A、C 和 F 相对应的十进制值。而对于把十六进制值 B、D 和 E 转换成十进制值时，则可以通过计算它们与前三个值的相对关系来完成。

比如，假设给你一个数字`0x173A4C`，可以通过展开每个十六进制数字，将它转换为二进制格式，如下所示：

十六进制	1	7	3	A	4	C
二进制	0001	0111	0011	1010	0100	1100

这样就得到了二进制表示`000101110011101001001100`。

反过来，如果给定一个二进制数字`1111001010110110110011`，你可以首先把它分为每 4 位一组，再把它转换为十六进制。不过要注意，如果位的总数不是 4 的倍数，最左边的一组可以少于 4 位，前面用 0 补足，然后将每个 4 位组转换为相应的十六进制数字：

二进制	11	1100	1010	1101	1011	0011
十六进制	3	C	A	D	B	3

练习题 2.1 完成下列数字转换：

- A. 将`0x39A7F8`转换为二进制。
- B. 将二进制`110010010111011`转换为十六进制。
- C. 将`0xD5E4C`转换为二进制。
- D. 将二进制`1001101110011110110101`转换为十六进制。

当值 x 是2的非负整数 n 次幂时，也就是 $x=2^n$ ，我们可以很容易地将 x 写成十六进制形式，只要记住 x 的二进制表示就是1后面跟 n 个0。十六进制数字0代表4个二进制0。所以，当 n 表示成 $i+4j$ 的形式，其中 $0 \leq i < 3$ 时，我们可以把 x 写成开头的十六进制数字为1($i=0$)、2($i=1$)、4($i=2$)或者8($i=3$)，后面跟随着 j 个十六进制的0。比如， $x=2048=2^{11}$ ，我们有 $n=11=3+4\times2$ ，从而得到十六进制表示`0x800`。

练习题 2.2 填写下表中的空白项，给出2的不同次幂的二进制和十六进制表示：

n	2^n (十进制)	2^n (十六进制)
9	512	0x200
19		
	16 384	
		0x10000
17		
	32	
		0x80



十进制和十六进制表示之间的转换需要使用乘法或者除法来处理一般情况。将一个十进制数字 x 转换为十六进制，可以反复地用 16 除 x ，得到一个商 q 和一个余数 r ，也就是 $x = q \times 16 + r$ 。然后，我们用十六进制数字表示的 r 作为最低位数字，并且通过对 q 反复进行这个过程得到剩下的数字。例如，考虑十进制 314156 的转换：

$$\begin{aligned} 314156 &= 19634 \times 16 + 12 && (\text{C}) \\ 19634 &= 1227 \times 16 + 2 && (\text{2}) \\ 1227 &= 76 \times 16 + 11 && (\text{B}) \\ 76 &= 4 \times 16 + 12 && (\text{C}) \\ 4 &= 0 \times 16 + 4 && (\text{4}) \end{aligned}$$

从这里，我们能读出十六进制表示为 0x4CB2C。

反过来，将一个十六进制数字转换为十进制数字，我们可以用相应的 16 的幂乘以每个十六进制数字。比如，给定数字 0x7AF，我们计算它对应的十进制值为 $7 \times 16^2 + 10 \times 16 + 15 = 7 \times 256 + 10 \times 16 + 15 = 1792 + 160 + 15 = 1967$ 。

练习题 2.3 一个字节可以用两个十六进制数字来表示。填写下表中缺失的项，给出不同字节模式的十进制、二进制和十六进制值。

十进制	二进制	十六进制
0	0000 0000	0x00
167		
62		
188		
	0011 0111	
	1000 1000	
	1111 0011	
		0x52
		0xAC
		0xE7

十进制和十六进制间的转换

较大数值的十进制和十六进制之间的转换，最好是让计算机或者计算器来完成。例如，下面的 Perl 语言脚本将（命令行给出的）一列数字从十进制转换为十六进制：

```
bin/d2h
1 #!/usr/local/bin/perl
2 # Convert list of decimal numbers into hex
3
4 for ($i = 0; $i < @ARGV; $i++) {
5     printf("%d\t= 0x%x\n", $ARGV[$i], $ARGV[$i]);
6 }
```

bin/d2h

一旦这个文件被设置为可执行的，命令

unix> ./d2h 100 500 751

会产生输出：

100=0x64

500=0x1f4

751=0x2ef



类似地，下面的脚本将十六进制转换为十进制：

```
1 #!/usr/local/bin/perl
2 # Convert list of hex numbers into decimal
3
4 for ($i = 0; $i < @ARGV; $i++) {
5     $val = hex($ARGV[$i]);
6     printf("0x%x = %d\n", $val, $val);
7 }
```

bin/h2d

bin/h2d

练习题 2.4 不将数字转换为十进制或者二进制，试着解答下面的算术题，答案要用十六进制表示。

提示：将执行十进制加法和减法所使用的方法改成以 16 为基数。

- A. 0x503c+0x8=
- B. 0x503c-0x40=
- C. 0x503c+64=
- D. 0x50ea-0x503c=

2.1.2 字

每台计算机都有一个字长（word size），指明整数和指针数据的标称大小（nominal size）。因为虚拟地址是以这样的一个字来编码的，所以字长决定的最重要的系统参数就是虚拟地址空间的最大大小。也就是说，对于一个字长为 w 位的机器而言，虚拟地址的范围为 $0 \sim 2^{w-1}$ ，程序最多访问 2^w 个字节。

今天大多数计算机的字长都是 32 位。这就限定了虚拟地址空间为 4 千兆字节（写作 4GB），也就是说，刚刚超过 4×10^9 字节。虽然对大多数应用而言，这个空间足够大了，但是现在已经有许多大规模的科学和数据库应用需要更大的存储。因此，随着存储器价格的降低，字长为 64 位的高端机器正逐渐变得普遍起来。硬件价格随着时间降低，台式机和笔记本电脑也会变成 64 位字长，所以我们会考虑 w 位字长的通用情况，也会考虑 $w=32$ 和 $w=64$ 的特殊情况。

2.1.3 数据大小

计算机和编译器支持多种不同方式编码的数字格式，如整数和浮点数，以及其他长度的数字。比如，许多机器都有处理单个字节的指令，也有处理表示为 2 字节、4 字节或者 8 字节整数的指令，还有些指令支持表示为 4 字节和 8 字节的浮点数。

C 语言支持整数和浮点数的多种数据格式。C 的数据类型 `char` 表示一个单独的字节。尽管“`char`”是由于它被用来存储文本串中的单个字符这一事实而得名，但它也能用来存储整数值。`C` 的数据类型 `int` 之前还能加上限定词 `short`、`long`，以及最近的 `long long`，以提供各种大小的整数表示。图 2-3 展示了为 C 语言中不同数据类型分配的字节数。准确的字节数依赖于机器和编译器。我们给出的是 32 位和 64 位机器的典型值。可以观察到，“短”整数分配有 2 个字节，而不加限定的 `int` 为 4 个字节。“长”整数使用机器的全字长。ISO C99 引入的“长长”整数数据类型允许 64 位整数。对于 32 位机器来说，编译器必须把这种数据类型的操作编译成执行一系列 32 位操作的代码。

图 2-3 给出了指针（例如，一个被声明为“`char*`”类型的变量）使用机器的全字长。大多数机器还支持两种不同的浮点数格式：单精度（在 C 中声明为 `float`）和双精度（在 C 中声明为 `double`）。格式分别使用 4 字节和 8 字节。



C 声明	32 位机器	64 位机器
char	1	1
short int	2	2
int	4	4
long int	4	8
long long int	8	8
char *	4	8
float	4	4
double	8	8

图 2-3 C 语言中数字数据类型的字节数

给 C 语言初学者：声明指针

对于任何数据类型 T ，声明

$T *p;$

表明 p 是一个指针变量，指向一个类型为 T 的对象。例如，

$char *p;$

表示将一个指针声明为指向一个 $char$ 类型的对象。

程序员应该力图使他们的程序在不同的机器和编译器上是可移植的。可移植性的一个方面就是使程序对不同数据类型的确切大小不敏感。C 语言标准对不同数据类型的数字范围设置了下界（这点在后面还将讲到），但是却没有上界。因为自 1980 年以来 32 位机器一直是标准，许多程序的编写都假设为图 2-3 中列出的 32 位机器对应的字节分配。随着 64 位机器的日益普及，在将这些程序移植到新机器上时，许多隐藏的对字长的依赖性就会显现出来，成为错误。比如，许多程序员假设一个声明为 int 类型的程序对象能被用来存储一个指针。这在大多数 32 位的机器上能正常工作，但是在一台 64 位的机器上却会导致问题。

2.1.4 寻址和字节顺序

对于跨越多字节的程序对象，我们必须建立两个规则：这个对象的地址是什么，以及在存储器中如何排列这些字节。在几乎所有的机器上，多字节对象都被存储为连续的字节序列，对象的地址为所使用字节中最小的地址。例如，假设一个类型为 int 的变量 x 的地址为 $0x100$ ，也就是说，地址表达式 $\&x$ 的值为 $0x100$ 。那么， x 的 4 个字节将被存储在存储器的 $0x100$ 、 $0x101$ 、 $0x102$ 和 $0x103$ 位置。

排列表示一个对象的字节有两个通用的规则。考虑一个 w 位的整数，位表示为 $[x_{w-1}, x_{w-2}, \dots, x_1, x_0]$ ，其中 x_{w-1} 是最高有效位，而 x_0 是最低有效位。假设 w 是 8 的倍数，这些位就能被分组成为字节，其中最高有效字节包含位 $[x_{w-1}, x_{w-2}, \dots, x_{w-8}]$ ，而最低有效字节包含位 $[x_7, x_6, \dots, x_0]$ ，其他字节包含中间的位。某些机器选择在存储器中按照从最低有效字节到最高有效字节的顺序存储对象，而另一些机器则按照从最高有效字节到最低有效字节的顺序存储。前一种规则——最低有效字节在最前面的方式，称为小端法（little endian）。大多数 Intel 兼容机都采用这种规则。后一种规则——最高有效字节在最前面的方式，称为大端法（big endian）。大多数 IBM 和 Sun Microsystems 的机器都采用这种规则。注意我们说的是“大多数”。这些规则并没有严格按照企业界限来划分。比如，IBM 和 Sun 制造的个人计算机使用的是 Intel 兼容的处理器，因此用的就是小端法。许多比较新的微处理器使用双端法（bi-endian），也就是说可以把它们配置成作为大端或者小端的机器运行。

继续我们前面的示例，假设变量 x 类型为 int ，位于地址 $0x100$ 处，它的十六进制值为 $0x01234567$ 。地址范围为 $0x100 \sim 0x103$ 的字节，其排列顺序依赖于机器的类型。



大端法

	0x100	0x101	0x102	0x103	
...	01	23	45	67	...

小端法

	0x100	0x101	0x102	0x103	
...	67	45	23	01	...

注意，在字 0x01234567 中，高位字节的十六进制值为 0x01，而低位字节值为 0x67。

令人吃惊的是，在哪种字节顺序是合适的这个问题上，人们表现得非常情绪化。实际上，术语“little endian”（小端）和“big endian”（大端）出自 Jonathan Swift 的《格列佛游记》(Gulliver's Travels) 一书，其中交战的两个派别无法就应该从哪一端（小端还是大端）打开一个半熟的鸡蛋达成一致。就像鸡蛋的问题一样，没有技术上的原因来选择字节顺序规则，因此，争论沦为关于社会政治问题的争论。只要选择了一种规则并且始终如一地坚持，其实对于哪种字节排序的选择都是任意的。

“端”(endian) 的起源

以下是 Jonathan Swift 在 1726 年关于大小端之争历史的描述：

“……我下面要告诉你是，Lilliput 和 Blefuscu 这两大强国在过去 36 个月里一直在苦战。战争开始是由于以下的原因：我们大家都认为，吃鸡蛋前，原始的方法是打破鸡蛋较大的一端，可是当今皇帝的祖父小时候吃鸡蛋，一次按古法打鸡蛋时碰巧将一个手指弄破了，因此他的父亲，当时的皇帝，就下了一道敕令，命令全体臣民吃鸡蛋时打破鸡蛋较小的一端，违令者重罚。老百姓们对这项命令极为反感。历史告诉我们，由此曾发生过 6 次叛乱，其中一个皇帝送了命，另一个丢了王位。这些叛乱大多都是由 Blefuscu 的国王大臣们煽动起来的。叛乱平息后，流亡的人总是逃到那个帝国去寻求避难。据估计，先后几次有 11 000 人情愿受死也不肯去打破鸡蛋较小的一端。关于这一争端，曾出版过几百本大部著作，不过大端派的书一直是受禁的，法律也规定该派的任何人不得做官。”（此段译文摘自网上蒋剑锋译的《格列佛游记》第一卷第 4 章。）

在他那个时代，Swift 是在讽刺英国 (Lilliput) 和法国 (Blefuscu) 之间持续的冲突。Danny Cohen，一位网络协议的早期开创者，第一次使用这两个术语来指代字节顺序 [25]，后来这个术语被广泛接纳了。

对于大多数应用程序员来说，他们机器所使用的字节顺序是完全不可见的，无论为哪种类型的机器编译的程序都会得到同样的结果。不过有时候，字节顺序会成为问题。首先是在不同类型的机器之间通过网络传送二进制数据时，一个常见的问题是当小端法机器产生的数据被发送到大端法机器或者反方向发送时会发现，接收程序字里的字节成了反序的。为了避免这类问题，网络应用程序的代码编写必须遵守已建立的关于字节顺序的规则，以确保发送方机器将它的内部表示转换成网络标准，而接收方机器则将网络标准转换为它的内部表示。我们将在第 11 章中看到这种转换的例子。

第二种情况是，当阅读表示整数数据的字节序列时字节顺序也很重要。通常在检查机器级程序时会出现这种情况。举一个示例，从某个文件中摘出了下面这行代码，该文件给出了一个针对 Intel IA32 处理器的机器级代码的文本表示：

```
80483bd: 01 05 64 94 04 08      add    %eax, 0x8049464
```

这一行是由反汇编器 (disassembler) 生成的，反汇编器是一种确定可执行程序文件所表示的指令序列的工具。我们将在第 3 章学习有关这些工具的更多知识，以及怎样解释像这样的行。现在，我们只需注意这行表述了十六进制字节串 01 05 64 94 04 08 是一条指令的字节级表示，这条指令把一个字长的数据加到存储在主存地址 0x8049464 的值上。如果取出这个序列的



后 4 个字节：64 94 04 08，按照相反的顺序写出，我们得到 08 04 94 64。去掉开头的 0 得到值 0x8049464，这就是右边的数值。当阅读此类小端法机器生成的机器级程序表示时，经常会将字节按照相反的顺序显示。书写字节序列的自然方式是最低位字节在左边，而最高位字节在右边，这正好和通常书写数字时最高有效位在左边，最低有效位在右边的方式相反。

字节顺序变得可见的第三种情况是当编写规避正常的类型系统的程序时。在 C 语言中，可以使用强制类型转换（cast）来允许以一种数据类型引用一个对象，而这种数据类型与创建这个对象时定义的数据类型不同。大多数应用编程都强烈不推荐这种编码技巧，但是它们对系统级编程来说是非常有用，甚至是必需的。

图 2-4 展示了一段 C 代码，它使用强制类型转换来访问和打印不同程序对象的字节表示。我们用 `typedef` 将数据类型 `byte_pointer` 定义为一个指向类型为 “`unsigned char`” 的对象的指针。这样一个字节指针引用一个字节序列，其中每个字节都被认为是一个非负整数。第一个例程 `show_bytes` 的输入是一个字节序列的地址，它用一个字节指针以及一个字节数来指示。`show_bytes` 打印出每个以十六进制表示的字节。C 格式化指令 “`%.2x`” 表明整数必须用至少两个数字的十六进制格式输出。

```
1 #include <stdio.h>
2
3 typedef unsigned char *byte_pointer;
4
5 void show_bytes(byte_pointer start, int len) {
6     int i;
7     for (i = 0; i < len; i++)
8         printf(" %.2x", start[i]);
9     printf("\n");
10 }
11
12 void show_int(int x) {
13     show_bytes((byte_pointer) &x, sizeof(int));
14 }
15
16 void show_float(float x) {
17     show_bytes((byte_pointer) &x, sizeof(float));
18 }
19
20 void show_pointer(void *x) {
21     show_bytes((byte_pointer) &x, sizeof(void *));
22 }
```

图 2-4 打印程序对象的字节表示这段代码使用强制类型转换来规避类型系统。

很容易定义针对其他数据类型的类似函数

给 C 语言初学者：使用 `typedef` 命名数据类型

C 语言中的 `typedef` 声明提供了一种给数据类型命名的方式。这能够极大地改善代码的可读性，因为深度嵌套的类型声明很难读懂。

`typedef` 的语法与声明变量的语法十分相似，除了它使用的是类型名，而不是变量名。因此，图 2-4 中 `byte_pointer` 的声明和将一个变量声明为类型 “`unsigned char*`” 有相同的形式。

例如，声明：

```
typedef int *int_pointer;
int_pointer ip;
```

将类型 “`int_pointer`” 定义为一个指向 `int` 的指针，并且声明了一个这种类型的变量 `ip`。我们还可以将这个变量直接声明为：



```
int *ip;
```

给 C 语言初学者：使用 printf 格式化输出

printf 函数（还有它的同类 fprintf 和 sprintf）提供了一种打印信息的方式，这种方式对格式化细节有相当大的控制能力。第一个参数是格式串（format string），而其余的参数都是要打印的值。在格式串里，每个以 '%' 开始的字符序列都表示如何格式化下一个参数。典型的示例有：'%d' 是输出一个十进制整数，'%f' 是输出一个浮点数，而 '%c' 是输出一个字符，其编码由参数给出。

给 C 语言初学者：指针和数组

在函数 show_bytes（图 2-4）中，我们看到指针和数组之间紧密的联系，这将在 3.8 节中详细描述。这个函数有一个类型为 byte_pointer（被定义为一个指向 unsigned char 的指针）的参数 start，但是我们在第 8 行上看到数组引用 start[i]。在 C 语言中，我们能够用数组表示法来引用指针，同时我们也能用指针表示法来引用数组元素。在这个例子中，引用 start[i] 表示我们想要读取以 start 指向的位置为起始的第 i 个位置处的字节。

过程 show_int、show_float 和 show_pointer 展示了如何使用程序 show_bytes 来分别输出类型为 int、float 和 void * 的 C 程序对象的字节表示。可以观察到它们仅仅传递给 show_bytes 一个指向它们参数 x 的指针 &x，且这个指针被强制类型转换为“unsigned char *”。这种强制类型转换告诉编译器，程序应该把这个指针看成指向一个字节序列，而不是指向一个原始数据类型的对象。然后，这个指针会被看成是对象使用的最低字节地址。

给 C 语言初学者：指针的创建和间接引用

在图 2-4 的第 13、17 和 21 行，我们看到对 C 和 C++ 中两种独有操作的使用。C 的“取地址”运算符 & 创建一个指针。在这三行中，表达式 &x 创建了一个指向保存变量 x 的位置的指针。这个指针的类型取决于 x 的类型，因此这三个指针的类型分别为 int*、float* 和 void**。（数据类型 void* 是一种特殊类型的指针，没有相关联的类型信息。）

强制类型转换运算符可以将一种数据类型转换为另一种。因此，强制类型转换（byte_pointer）&x 表明无论指针 &x 以前是什么类型，它现在就是一个指向数据类型为 unsigned char 的指针。这里给出的这些强制类型转换不会改变真实的指针，它们只是告诉编译器以新的数据类型来看待被指向的数据。

这些过程使用 C 语言的运算符 sizeof 来确定对象使用的字节数。一般来说，表达式 sizeof(T) 返回存储一个类型为 T 的对象所需要的字节数。使用 sizeof，而不是一个固定的值，是向编写在不同机器类型上可移植的代码迈进了一步。

在几种不同的机器上运行如图 2-5 所示的代码，得到如图 2-6 所示的结果。我们分别使用了以下几种机器：

Linux 32：运行 Linux 的 Intel IA32 处理器

Windows：运行 Windows 的 Intel IA32

Sun：运行 Solaris 的 Sun Microsystems SPARC 处理器

Linux 64：运行 Linux 的 Intel x86-64 处理器

参数 12 345 的十六进制表示为 0x00003039。对于 int 类型的数据，除了字节顺序以外，我们在所有机器上都得到相同的结果。特别地，我们可以看到在 Linux 32、Windows 和 Linux 64 上，最低有效字节值 0x39 最先输出，这说明它们是小端法机器；而在 Sun 上却最后输出，这说明 Sun 是大端法机器。同样地，float 数据的字节，除了字节顺序以外，也都是相同的。另一方



面，指针值却是完全不同的。不同的机器 / 操作系统配置使用不同的存储分配规则。一个值得注意的特性是 Linux 32、Windows 和 Sun 的机器使用 4 字节地址，而 Linux 64 使用 8 字节地址。

```
code/data/show-bytes.c
1 void test_show_bytes(int val) {
2     int ival = val;
3     float fval = (float) ival;
4     int *pval = &ival;
5     show_int(ival);
6     show_float(fval);
7     show_pointer(pval);
8 }
```

code/data/show-bytes.c

图 2-5 字节表示的示例。这段代码打印示例数据对象的字节表示

机器	值	类型	字节 (十六进制)
Linux 32	12 345	int	39 30 00 00
Windows	12 345	int	39 30 00 00
Sun	12 345	int	00 00 30 39
Linux 64	12 345	int	39 30 00 00
Linux 32	12 345.0	float	00 e4 40 46
Windows	12 345.0	float	00 e4 40 46
Sun	12 345.0	float	46 40 e4 00
Linux 64	12 345.0	float	00 e4 40 46
Linux 32	&ival	int *	e4 f9 ff bf
Windows	&ival	int *	b4 cc 22 00
Sun	&ival	int *	ef ff fa 0c
Linux 64	&ival	int *	b8 11 e5 ff ff 7f 00 00

图 2-6 不同数据值的字节表示。除了字节顺序以外，int 和 float 的结果是一样的。指针值与机器相关

可以观察到，尽管浮点型和整型数据都是对数值 12 345 编码，但是它们有非常不同的字节模式：整型为 0x00003039，而浮点数为 0x4640E400。一般而言，这两种格式使用不同的编码方法。如果我们将这些十六进制模式扩展为二进制形式，并且适当地将它们移位，我们就会发现一个有 13 个相匹配的位的序列，用一串星号标识出来：

```
0 0 0 0 3 0 3 9
000000000000000011000000111001
*****
4 6 4 0 E 4 0 0
01000110010000001110010000000000
```

这并不是巧合。当我们研究浮点数格式时，还将再回到这个例子。

练习题 2.5 思考下面对 show_bytes 的三次调用：

```
int val=0x87654321;
byte_pointer valp=(byte_pointer)&val;
show_bytes(valp, 1); /* A.* /
```



```
show_bytes(valp, 2); /* B.*/
show_bytes(valp, 3); /* C.*/
指出在小端法机器和大端法机器上，每次调用的输出值。
```

- A. 小端法： 大端法：
- B. 小端法： 大端法：
- C. 小端法： 大端法：

练习题 2.6 使用 show_int 和 show_float，我们确定整数 3510593 的十六进制表示为 0x00359141，而浮点数 3510593.0 的十六进制表示为 0x4A564504。

- A. 写出这两个十六进制值的二进制表示。
- B. 移动这两个二进制串的相对位置，使得它们相匹配的位数最多。有多少位相匹配呢？
- C. 串中的什么部分不相匹配？

2.1.5 表示字符串

C 语言中字符串被编码为一个以 null（其值为 0）字符结尾的字符数组。每个字符都由某个标准编码来表示，最常见的是 ASCII 字符码。因此，如果我们以参数“12345”和 6（包括终止符）来运行例程 show_bytes，我们得到结果 31 32 33 34 35 00。请注意，十进制数字 x 的 ASCII 码正好是 0x3x，而终止字节的十六进制表示为 0x00。在使用 ASCII 码作为字符码的任何系统上都将得到相同的结果，与字节顺序和字大小规则无关。因而，文本数据比二进制数据具有更强的平台独立性。

生成一张 ASCII 表

通过执行命令 man ascii，你可以得到一张 ASCII 字符码的表。

练习题 2.7 下面对 show_bytes 的调用将输出什么结果？

```
const char *s = "abcdef";
show_bytes((byte_pointer)s, strlen(s));
```

注意字母‘a’～‘z’的 ASCII 码为 0x61～0x7A。

文字编码的 Unicode 标准

ASCII 字符集适合于编码英语文档，但是在表达一些特殊字符方面却没有太多办法，它完全不适合编码希腊语、俄语和中文这样语言的文档。近几年，开发出很多方法来对不同语言的文字编码。Unicode 联合会（Unicode Consortium）修订了最复杂且最普遍接受的文字编码标准。当前的 Unicode 标准（5.0 版）的字库包括近 100 000 个字符，支持的语言范围广，从阿尔巴尼亚语到 Xamtanga（埃塞俄比亚 Xamir 人所说的语言）。

基本编码，也称为 Unicode 的“统一字符集”，使用 32 位来表示字符。这好像是要求文本串中每个字符要占用 4 个字节。不过，可以用一些替代编码，常见的字符只需要 1 个或 2 个字节，而不太常用的字符需要多一些的字节数。特别地，UTF-8 表示将每个字符编码为一个字节序列，这样标准 ASCII 字符还是使用和它们在 ASCII 中一样的单字节编码，这也就意味着所有的 ASCII 字节序列用 ASCII 码表示和用 UTF-8 表示是一样的。

Java 编程语言使用 Unicode 来表示字符串。对于 C 语言也有支持 Unicode 的程序库。

2.1.6 表示代码

考虑下面的 C 函数：

```
1 int sum(int x, int y) {
2     return x + y;
3 }
```



当我们在示例机器上编译时，生成如下字节表示的机器代码：

Linux 32: 55 89 e5 8b 45 0c 03 45 08 c9 c3
Windows: 55 89 e5 8b 45 0c 03 45 08 5d c3
Sun: 81 c3 e0 08 90 02 00 09
Linux 64: 55 48 89 e5 89 7d fc 89 75 f8 03 45 fc c9 c3

我们发现指令编码是不同的。不同的机器类型使用不同的且不兼容的指令和编码方式。即使是完全一样的进程运行在不同的操作系统上也会有不同的编码规则，因此二进制代码是不兼容的。二进制代码很少能在不同机器和操作系统组合之间移植。

计算机系统的一个基本概念就是从机器的角度来看，程序仅仅只是字节序列。机器没有关于初始源程序的任何信息，除了可能有些用来帮助调试的辅助表以外。在第3章学习机器级编程时，我们将更清楚地理解这一点。

2.1.7 布尔代数简介

二进制值是计算机编码、存储和操作信息的核心，所以围绕数值0和1的研究已经演化出了丰富的数学知识体系。这起源于1850年前后乔治·布尔（George Boole, 1815—1864）的工作，因此也称为布尔代数（Bool algebra）。布尔注意到通过将逻辑值TRUE（真）和FALSE（假）编码为二进制值1和0，能够设计出一种代数，以研究逻辑推理的基本原则。

最简单的布尔代数是在二元集合{0, 1}基础上的定义。图2-7定义了这种布尔代数中的几种运算。我们用来表示这些运算的符号是和C语言的位级运算使用的符号相匹配的，这些将在后面讨论到。布尔运算 \sim 对应于逻辑运算NOT，在命题逻辑中用符号 \neg 表示。也就是说，当P不是真的时候，我们就说 $\neg P$ 是真的，反之亦然。相应地，当P等于0时， $\neg P$ 等于1，反之亦然。布尔运算&对应于逻辑运算AND，在命题逻辑中用符号 \wedge 表示。当P和Q都为真时，我们说 $P \wedge Q$ 为真。相应地，只有当 $p=1$ 且 $q=1$ 时， $p \wedge q$ 才等于1。布尔运算|对应于逻辑运算OR，在命题逻辑中用符号 \vee 表示。当P或者Q为真时，我们说 $P \vee Q$ 成立。相应地，当 $p=1$ 或者 $q=1$ 时， $p \vee q$ 等于1。布尔运算^对应于逻辑运算异或，在命题逻辑中用符号 \oplus 表示。当P或者Q为真但不同时为真时，我们说 $P \oplus Q$ 成立。相应地，当 $p=1$ 且 $q=0$ ，或者 $p=0$ 且 $q=1$ 时， $p \oplus q$ 等于1。

\sim	&		\wedge
0 1	0 0 0	0 0 1	0 0 1
1 0	1 0 1	1 1 1	1 1 0

图2-7 布尔代数的运算

后来创立信息理论领域的Claude Shannon（1916—2001）首先建立了布尔代数和数字逻辑之间的联系。在1937年，他在硕士论文中表明了布尔代数可以用来设计和分析机电继电器网络。尽管那时计算机技术已经取得了相当的发展，但是布尔代数仍然在数字系统的设计和分析中扮演着重要的角色。

我们可以将上述4个布尔运算扩展到位向量的运算，位向量就是有固定长度为w、由0和1组成的串。位向量的运算可以定义成参数的每个对应元素之间的运算。假设a和b分别表示位向量 $[a_{w-1}, a_{w-2}, \dots, a_0]$ 和 $[b_{w-1}, b_{w-2}, \dots, b_0]$ 。我们将 $a \& b$ 也定义为一个长度为w的位向量，其中第i个元素等于 $a_i \& b_i$ ， $0 \leq i < w$ 。可以用类似的方式将运算|、^和 \sim 扩展到位向量上。

举个例子，假设 $w=4$ ，参数 $a=[0110]$ ， $b=[1100]$ 。那么4种运算 $a \& b$ 、 $a | b$ 、 $a ^ b$ 和 $\sim b$ 分别得到以下结果：

$$\begin{array}{rcl} 0110 & 0110 & 0110 \\ \& 1100 & | 1100 & \sim 1100 \\ \hline 0100 & 1110 & 1010 & 0011 \end{array}$$

练习题2.8 填写下表，给出位向量的布尔运算的求值结果。



运算	结果
a	[01101001]
b	[01010101]
$\sim a$	_____
$\sim b$	_____
$a \& b$	_____
$a b$	_____
$a \sim b$	_____

DATA:BOOL：关于布尔代数和布尔环的更多内容

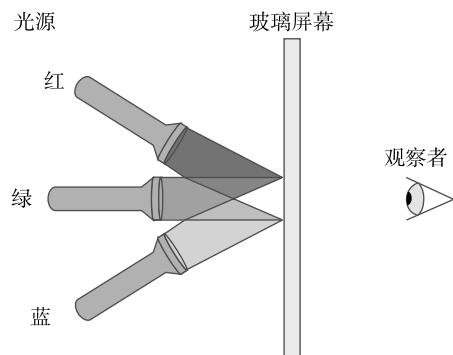
对于任意整数 $w > 0$, 长度为 w 的位向量上的布尔运算 $|$ 、 $\&$ 和 \sim 形成了一个布尔代数。最简单的情况是 $w=1$ 时, 只有 2 个元素; 但是对于更普遍的情况下, 有 2^w 个长度为 w 的位向量。布尔代数和整数算术运算有很多相似之处。例如, 乘法对加法的分配律, 写做 $a \cdot (b+c) = (a \cdot b) + (a \cdot c)$, 而布尔运算 $\&$ 对 $|$ 的分配律, 写做 $a \& (b \cdot c) = (a \& b) | (a \& c)$ 。此外, 布尔运算 $|$ 对 $\&$ 也有分配律, 写做 $a | (b \& c) = (a \cdot b) \& (a \cdot c)$, 但是对于整数我们不能说 $a + (b \cdot c) = (a+b) \cdot (a+c)$ 。

当考虑长度为 w 的位向量上的 \wedge 、 $\&$ 和 \sim 运算时, 会得到一种不同的数学形式, 我们称为布尔环 (Boolean ring)。布尔环与整数运算有很多相同的属性。例如, 整数运算的一个属性是每个值 x 都有一个加法逆元 (additive inverse) $-x$, 使得 $x + (-x) = 0$ 。布尔环也有类似的属性, 这里的“加法”运算是 \wedge , 不过这时每个元素的加法逆元是它自己本身。也就是说, 对于任何值 a 来说, $a \wedge a = 0$, 这里我们用 0 来表示全 0 的位向量。可以看到对单个位来说这是成立的, 即 $0 \wedge 0 = 1 \wedge 1 = 0$, 将这个扩展到位向量也是成立的。当我们重新排列组合顺序, 这个属性也仍然成立, 因此有 $(a \wedge b) \wedge a = b$ 。这个属性会引起一些很有趣的结果和聪明的技巧, 在练习题 2.10 中我们会有所探讨。

位向量一个很有用的应用就是表示有限集合。我们可以用位向量 $[a_{w-1}, \dots, a_1, a_0]$ 编码任何子集 $A = \{0, 1, \dots, w-1\}$, 其中 $a_i = 1$ 当且仅当 $i \in A$ 。例如, (记住我们是把 a_{w-1} 写在左边, 而将 a_0 写在右边), 位向量 $a = [01101001]$ 表示集合 $A = \{0, 3, 5, 6\}$, 而 $b = [01010101]$ 表示集合 $B = \{0, 2, 4, 6\}$ 。使用这种编码集合的方法, 布尔运算 $|$ 和 $\&$ 分别对应于集合的并和交, 而 \sim 对应于集合的补。还是用前面那个例子, 运算 $a \& b$ 得到位向量 $[01000001]$, 而 $A \cap B = \{0, 6\}$ 。

在大量实际应用中, 我们都能看到用位向量来对集合编码。例如, 在第 8 章, 我们会看到有很多不同的信号会中断程序执行。我们能够通过指定一个位向量掩码, 有选择地使能或是不能屏蔽一些信号, 其中某一位位置上为 1 时, 表明信号 i 是有效的, 而 0 表明该信号是被屏蔽的。因而, 这个掩码表示的就是设置为有效信号的集合。

练习题 2.9 通过混合三种不同颜色的光 (红色、绿色和蓝色), 计算机可以在视频屏幕或者液晶显示器上产生彩色的画面。设想一种简单的方法, 使用三种不同颜色的光, 每种光都能打开或关闭, 投射到玻璃屏幕上, 如图所示:





那么基于光源 R (红)、G (绿)、B (蓝) 的关闭 (0) 或打开 (1)，我们就能够创建 8 种不同的颜色：

R	G	B	颜色	R	G	B	颜色
0	0	0	黑色	1	0	0	红色
0	0	1	蓝色	1	0	1	红紫色
0	1	0	绿色	1	1	0	黄色
0	1	1	蓝绿色	1	1	1	白色

这些颜色中的每一种都能用一个长度为 3 的位向量来表示，我们可以对它们进行布尔运算。

A. 一种颜色的补是通过关掉打开的光源，且打开关闭的光源而形成的。那么上面列出的 8 种颜色每一种的补是什么？

B. 描述下列颜色应用布尔运算的结果：

$$\begin{array}{lll} \text{蓝色} & | & \text{绿色} = \\ \text{黄色} & \& \text{蓝绿色} = \\ \text{红色} & ^\wedge & \text{红紫色} = \end{array}$$

2.1.8 C 语言中的位级运算

C 语言的一个很有用的特性就是它支持按位布尔运算。事实上，我们在布尔运算中使用的那些符号就是 C 语言所使用的：| 就是 OR (或)，& 就是 AND (与)，~ 就是 NOT (取反)，而 ^ 就是 EXCLUSIVE-OR (异或)。这些运算能运用到任何“整型”的数据类型上，也就是那些声明为 char 或者 int 的数据类型，无论它们有没有像 short、long、long long 或者 unsigned 这样的限定词。以下是一些对 char 数据类型表达式求值的例子。

C 的表达式	二进制表达式	二进制结果	十六进制结果
$\sim 0x41$	$\sim [0100 0001]$	[1011 1110]	0xBE
$\sim 0x00$	$\sim [0000 0000]$	[1111 1111]	0xFF
$0x69 \& 0x55$	$[0110 1001] \& [0101 0101]$	[0100 0001]	0x41
$0x69 \mid 0x55$	$[0110 1001] \mid [0101 0101]$	[0111 1101]	0x7D

正如示例说明的那样，确定一个位级表达式的结果最好的方法，就是将十六进制的参数扩展成二进制表示并执行二进制运算，然后再转换回十六进制。

练习题 2.10 对于任一位向量 a ，有 $a \wedge a = 0$ 。应用这一属性，考虑下面的程序：

```
1 void inplace_swap(int *x, int *y) {
2     *y = *x ^ *y; /* Step 1 */
3     *x = *x ^ *y; /* Step 2 */
4     *y = *x ^ *y; /* Step 3 */
5 }
```

正如程序名字所暗示的那样，我们认为这个过程的效果是交换指针变量 x 和 y 所指向的存储位置处存放的值。注意，与通常的交换两个数值的技术不一样，当移动一个值时，我们不需要第三个位置来临时存储另一个值。这种交换方式并没有性能上的优势，它仅仅是一个智力游戏。

以指针 x 和 y 指向的位置存储的值分别是 a 和 b 作为开始，填写下表，给出在程序的每一步之后，存储在这两个位置中的值。利用 \wedge 的属性证明达到了所希望的效果。回想一下，每个元素就是它自身的加法逆元 ($a \wedge a = 0$)。

步骤	*x	*y
初始	a	b
第一步		
第二步		
第三步		



练习题 2.11 在练习题 2.10 中的 `inplace_swap` 函数的基础上，你决定写一段代码，实现将一个数组中的元素头尾两端依次对调。你写出下面这个函数：

```
1 void reverse_array(int a[], int cnt) {  
2     int first, last;  
3     for (first = 0, last = cnt-1;  
4         first <= last;  
5         first++, last--)  
6         inplace_swap(&a[first], &a[last]);  
7 }
```

当你对一个包含元素 1、2、3 和 4 的数组使用这个函数时，正如预期的那样，现在数组的元素变成了 4、3、2 和 1。不过，当你对一个包含元素 1、2、3、4 和 5 的数组使用这个函数时，你会很惊奇地看到得到数字的元素为 5、4、0、2 和 1。实际上，你会发现这段代码对所有偶数长度的数组都能正确地工作，但是当数组的长度为奇数时，它就会把中间的元素设置成 0。

- A. 对于一个长度为奇数的数组，长度 $cnt=2k+1$ ，函数 `reverse_array` 最后一次循环中，变量 `first` 和 `last` 的值分别是什么？
- B. 为什么这时调用函数 `xor_swap` 会将数组元素设置为 0？
- C. 对 `reverse_array` 的代码做哪些简单改动就能消除这个问题？

位级运算的一个常见用法就是实现掩码运算，这里掩码是一个位模式，表示从一个字中选出的位的集合。让我们来看一个例子，掩码 0xFF（最低的 8 位为 1）表示一个字的低位字节。位级运算 `x&0xFF` 生成一个由 `x` 的最低有效字节组成的值，而其他的字节就被置为 0。比如，对于 `x=0x89ABCDEF`，其表达式将得到 0x000000EF。表达式 `~0` 将生成一个全 1 的掩码，不管机器的字大小是多少。尽管对于一个 32 位机器来说，同样的掩码可以写成 0xFFFFFFFF，但是这样的代码不是可移植的。



练习题 2.12 对于下面的值，写出变量 `x` 的 C 语言表达式。你的代码应该对任何字长 $w \geq 8$ 都能工作。我们给出了当 $x=0x87654321$ 以及 $w=32$ 时表达式求值的结果，仅供参考。

- A. `x` 的最低有效字节，其他位均置为 0。[0x00000021]。
- B. 除了 `x` 的最低有效字节外，其他的位都取补，最低有效字节保持不变。[0x789ABC21]。
- C. `x` 的最低有效字节设置成全 1，其他字节都保持不变。[0x876543FF]。



练习题 2.13 从 20 世纪 70 年代末到 80 年代末，Digital Equipment 的 VAX 计算机是一种非常流行的机型。它没有布尔运算 AND 和 OR 指令，只有 `bis`（位设置）和 `bic`（位清除）这两种指令。两种指令的输入都是一个数据字 `x` 和一个掩码字 `m`。它们生成一个结果 `z`，`z` 是由根据掩码 `m` 的位来修改 `x` 的位得到的。使用 `bis` 指令，这种修改就是在 `m` 为 1 的每个位置上，将 `z` 对应的位设置为 1。使用 `bic` 指令，这种修改就是在 `m` 为 1 的每个位置，将 `z` 对应的位设置为 0。

为了清楚因为这些运算与 C 语言位级运算的关系，假设我们有两个函数 `bis` 和 `bic` 来实现位设置和位清除操作。只想用这两个函数，而不使用任何其他 C 语言运算，来实现按位 | 和 ^ 运算。填写下列代码中缺失的代码。提示：写出 `bis` 和 `bic` 运算的 C 语言表达式。

```
/* Declarations of functions implementing operations bis and bic */  
int bis(int x, int m);  
int bic(int x, int m);  
  
/* Compute x|y using only calls to functions bis and bic */  
int bool_or(int x, int y) {  
    int result = _____;  
    return result;  
}  
  
/* Compute x^y using only calls to functions bis and bic */
```



```
int bool_xor(int x, int y) {
    int result = _____;
    return result;
}
```

2.1.9 C 语言中的逻辑运算

C 语言还提供了一组逻辑运算符 `||`、`&&` 和 `!`，分别对应于命题逻辑中的 OR、AND 和 NOT 运算。逻辑运算很容易和位级运算相混淆，但是它们的功能是完全不同的。逻辑运算认为所有非零的参数都表示 TRUE，而参数 0 表示 FALSE。它们返回 1 或者 0，分别表示结果为 TRUE 或者为 FALSE。以下是一些表达式求值的示例。

表达式	结果
<code>!0x41</code>	<code>0x00</code>
<code>!0x00</code>	<code>0x01</code>
<code>!!0x41</code>	<code>0x01</code>
<code>0x69&&0x55</code>	<code>0x01</code>
<code>0x69 0x55</code>	<code>0x01</code>

可以观察到，按位运算只有在特殊情况下，也就是参数被限制为 0 或者 1 时，才和与其对应的逻辑运算有相同的行为。

逻辑运算符 `&&` 和 `||` 与它们对应的位级运算 `&` 和 `|` 之间第二个重要的区别是，如果对第一个参数求值就能确定表达式的结果，那么逻辑运算符就不会对第二个参数求值。因此，例如，表达式 `a&&5/a` 将不会造成被零除，而表达式 `p&&*p++` 也不会导致间接引用空指针。

练习题 2.14 假设 `x` 和 `y` 的字节值分别为 `0x66` 和 `0x39`。填写下表，指明各个 C 表达式的字节值。

表达式	值	表达式	值
<code>x & y</code>		<code>x && y</code>	
<code>x y</code>		<code>x y</code>	
<code>~x ~y</code>		<code>!x !y</code>	
<code>x & !y</code>		<code>x && ~y</code>	

练习题 2.15 只使用位级和逻辑运算，编写一个 C 表达式，它等价于 `x==y`。换句话说，当 `x` 和 `y` 相等时它将返回 1，否则就返回 0。

2.1.10 C 语言中的移位运算

C 语言还提供了一组移位运算，以便向左或者向右移动位模式。对于一个位表示为 $[x_{n-1}, x_{n-2}, \dots, x_0]$ 的操作数 `x`，C 表达式 `x<<k` 会生成一个值，其位表示为 $[x_{n-k-1}, x_{n-k-2}, \dots, x_0, 0, \dots, 0]$ 。也就是说，`x` 向左移动 `k` 位，丢弃最高的 `k` 位，并在右端补 `k` 个 0。移位量应该是一个 $0 \sim n-1$ 之间的值。移位运算是从左至右可结合的，所以 `x<<j<<k` 等价于 `(x<<j)<<k`。

有一个相应的右移运算 `x>>k`，但是它的行为有点微妙。一般而言，机器支持两种形式的右移：逻辑右移和算术右移。逻辑右移在左端补 `k` 个 0，得到的结果是 $[0, \dots, 0, x_{n-1}, x_{n-2}, \dots, x_k]$ 。算术右移是在左端补 `k` 个最高有效位的值，得到的结果是 $[x_{n-1}, \dots, x_{n-1}, x_{n-1}, x_{n-2}, \dots, x_k]$ 。这种做法看上去可能有点奇特，但是我们会发现它对有符号整数数据的运算非常有用。



让我们来看一个例子，下面的表给出了对某些实例 8 位数据做不同的移位操作得到的结果。

操作	值
参数 x	[01100011] [10010101]
$x \ll 4$	[00110000] [01010000]
$x \gg 4$ (逻辑右移)	[00000110] [00001001]
$x \gg 4$ (算术右移)	[00000110] [11111001]

斜体的数字表示的是最右端（左移）或最左端（右移）填充的值。可以看到除了一个条目之外，其他的都涉及填充 0。唯一的例外是算术右移 [10010101] 的情况。因为操作数的最高位是 1，填充的值就是 1。

C 语言标准并没有明确定义应该使用哪种类型的右移。对于无符号数据（也就是以限定词 `unsigned` 声明的整型对象），右移必须是逻辑的。而对于有符号数据（默认的声明的整型对象），算术的或者逻辑的右移都可以。不幸的是，这就意味着任何假设一种或者另一种右移形式的代码都潜在着可移植性问题。然而，实际上，几乎所有的编译器 / 机器组合都对有符号数据使用算术右移，且许多程序员也都假设机器会使用这种右移。

另一方面，Java 对于如何进行右移有明确的定义。表达式 `x>>k` 会将 `x` 算术右移 `k` 个位置，而 `x>>>k` 会对 `x` 做逻辑右移。

移动 k 位，这里 k 很大

对于一个由 w 位组成的数据类型，如果要移动 $k > w$ 位会得到什么结果呢？例如，在一个 32 位机器上计算下面的表达式会得到什么结果：

```
int lval = 0xFEDCBA98 << 32;
int aval = 0xFEDCBA98 >> 36;
unsigned uval = 0xFEDCBA98u >> 40;
```

C 语言标准很小心地规避了说明在这种情况下该如何做。在许多机器上，当移动一个 w 位的值时，移位指令只考虑位移量的低 $\log_2 w$ 位，因此实际上位移量就是通过计算 $k \bmod w$ 得到的。例如，在一台采用这个规则的 32 位机器上，上面三个移位运算分别是移动 0、4 和 8 位，得到结果：

```
lval 0xFEDCBA98
aval 0xFFEDCBA9
uval 0x00FEDCBA
```

不过这种行为对于 C 程序来说是没有保证的，所以移位数量应该保持小于字长。

另一方面，Java 特别要求位移数量应该按照我们前面所讲的求模的方法来计算。

与移位运算有关的操作符优先级问题

常常有人会写这样的表达式 `1<<2+3<<4`，其本意是 $(1 \ll 2) + (3 \ll 4)$ 。但是在 C 语言中，前面的表达式等价于 `1<<(2+3)<<4`，这是由于加法（和减法）的优先级比移位运算要高。然后，按照从左至右结合性规则，括号应该是这样打的 `(1<<(2+3))<<4`，因此得到的结果是 512，而不是期望的 52。

在 C 表达式中搞错优先级是一种常见的程序错误，而且常常很难检查出来。所以当你拿不准的时候，请加上括号！

练习题 2.16 填写下表，说明不同移位运算对单字节数的影响。思考移位运算的最好方式是使用二进



制表示。将最初的值转换为二进制执行移位运算，然后再转换回十六进制。每个答案都应该是 8 个二进制数字或者 2 个十六进制数字。

x		x<<3		x>>2 (逻辑的)		x>>2 (算术的)	
十六进制	二进制	二进制	十六进制	二进制	十六进制	二进制	十六进制
0xC3							
0x75							
0x87							
0x66							

2.2 整数表示

在本节中，我们描述用位来编码整数的两种不同的方式：一种只能表示非负数，而另一种能够表示负数、零和正数。后面我们将会看到它们的数学属性和机器级实现方面密切关联。我们还会研究扩展或者收缩一个已编码整数以适应不同长度表示的效果。

2.2.1 整型数据类型

C 语言支持多种整型数据类型——表示有限范围的整数。这些类型如图 2-8 和图 2-9 所示，其中还给出了“典型的”32 位和 64 位机器的取值范围。每种类型都能用关键字来指定大小，这些关键字包括 `char`、`short`、`long` 或者 `long long`，同时还可以指示被表示的数字是非负数（声明为 `unsigned`），或者可能是负数（默认）。如图 2-3 所示，这些不同大小的分配的字节数会根据机器的字长和编译器有所不同。根据字节分配，不同的大小所能表示的值的范围是不同的。这里给出来的唯一一个与机器相关的取值范围是大小指示符 `long` 的。大多数 64 位机器使用 8 个字节表示，比 32 位机器上使用的 4 个字节表示的取值范围大很多。

图 2-8 和图 2-9 中一个值得注意的特点是取值范围不是对称的——负数的范围比整数的范围大 1。当我们考虑如何表示负数的时候，会看到为什么会这样。

C 数据类型	最小值	最大值
<code>char</code>	-128	127
<code>unsigned char</code>	0	255
<code>short [int]</code>	-32 768	32 767
<code>unsigned short [int]</code>	0	65 535
<code>int</code>	-2 147 483 648	2 147 483 647
<code>unsigned [int]</code>	0	4 294 967 295
<code>long [int]</code>	-2 147 483 648	2 147 483 647
<code>unsigned long [int]</code>	0	4 294 967 295
<code>long long [int]</code>	-9 223 372 036 854 775 808	9 223 372 036 854 775 807
<code>unsigned long long [int]</code>	0	18 446 744 073 709 551 615

图 2-8 32 位机器上 C 语言的整型数据类型的典型取值范围（方括号中的文字是可选的）

C 语言标准定义了每种数据类型必须能够表示的最小的取值范围。如图 2-10 所示，它们的取值范围与图 2-8 和图 2-9 所示的典型实现一样或者小一些。特别地，我们看到它们只要求正数和负数的取值范围是对称的。此外，数据类型 `int` 可以用 2 个字节的数字来实现，而这几乎退回到了 16 位机器的时代。还看到，`long` 的大小可以用 4 个字节的数字来实现，而实际上也常常是这样。数据类型 `long long` 是在 ISO C99 中引入的，它需要至少 8 个字节表示。



C 数据类型	最小值	最大值
char	-128	127
unsigned char	0	255
short [int]	-32 768	32 767
unsigned short [int]	0	65 535
int	-2 147 483 648	2 147 483 647
unsigned [int]	0	4 294 967 295
long [int]	-9 223 372 036 854 775 808	9 223 372 036 854 775 807
unsigned long [int]	0	18 446 744 073 709 551 615
long long [int]	-9 223 372 036 854 775 808	9 223 372 036 854 775 807
unsigned long long [int]	0	18 446 744 073 709 551 615

图 2-9 64 位机器上 C 语言的整型数据类型的典型取值范围（方括号中的文字是可选的）

C 数据类型	最小值	最大值
char	-127	127
unsigned char	0	255
short [int]	-32 767	32 767
unsigned short [int]	0	65 535
int	-32 767	32 767
unsigned [int]	0	65 535
long [int]	-2 147 483 647	2 147 483 647
unsigned long [int]	0	4 294 967 295
long long [int]	-9 223 372 036 854 775 807	9 223 372 036 854 775 807
unsigned long long [int]	0	18 446 744 073 709 551 615

图 2-10 C 语言的整型数据类型的保证的取值范围（方括号中的文字是可选的）

注：C 语言标准要求这些数据类型必须至少具有这样的取值范围。

给 C 语言初学者：C、C++ 和 Java 中的有符号和无符号数

C 和 C++ 都支持有符号（默认）和无符号数。Java 只支持有符号数。

2.2.2 无符号数的编码

假设一个整数数据类型有 w 位。我们可以将位向量写成 \vec{x} ，表示整个向量，或者写成 $[x_{w-1}, x_{w-2}, \dots, x_0]$ ，表示向量中的每一位。把 \vec{x} 看做一个二进制表示的数，就获得了 \vec{x} 的无符号表示。我们用一个函数 $B2U_w$ (Binary to Unsigned) 的缩写，长度为 w 来表示：

$$B2U_w(\vec{x}) \doteq \sum_{i=0}^{w-1} x_i 2^i \quad (2-1)$$

在这个等式中，符号 “ \doteq ” 表示左边被定义为等于右边。函数 $B2U_w$ 将一个长度为 w 的 0、1 串映射到非负整数。举一个示例，图 2-11 展示的是下面几种情况下 $B2U$ 给出的从位向量到整数的映射。

$$\begin{aligned} B2U_4([0001]) &= 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 0 + 0 + 0 + 1 = 1 \\ B2U_4([0101]) &= 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 0 + 4 + 0 + 1 = 5 \\ B2U_4([1011]) &= 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 8 + 0 + 2 + 1 = 11 \\ B2U_4([1111]) &= 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 8 + 4 + 2 + 1 = 15 \end{aligned} \quad (2-2)$$

在图中，我们用长度为 2^i 的指向右侧箭头的条表示每个位的位置 i 。每个位向量对应的数值



就等于所有值为 1 的位对应的条的长度之和。

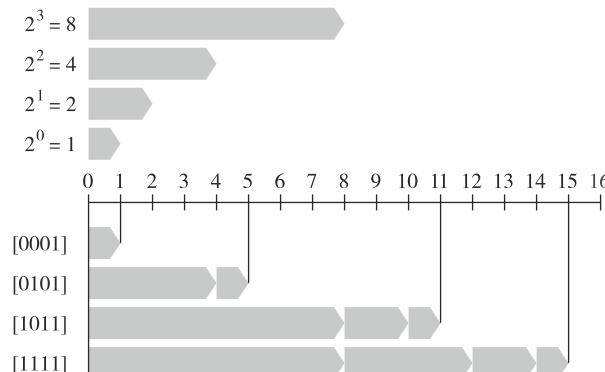


图 2-11 $w=4$ 的无符号数示例。当二进制表示中为 i 为 1，数值就会相应的加上 2^i

让我们来考虑一下 w 位所能表示的值的范围。最小值是用位向量 $[00\cdots 0]$ 表示，也就是整数值 0，而最大值是用位向量 $[11\cdots 1]$ 表示，也就是整数值 $UMax_w \doteq \sum_{i=0}^{w-1} 2^i = 2^w - 1$ 。以 $w=4$ 位为例， $UMax_4 = B2U_4([1111]) = 2^4 - 1 = 15$ 。因此，函数 $B2U_w$ 能够被定义为一个映射 $B2U_w: \{0, 1\}^w \rightarrow \{0, \dots, 2^w - 1\}$ 。

无符号数的二进制表示有一个很重要的属性，就是每个介于 $0 \sim 2^w - 1$ 之间的数都有唯一一个 w 位的值编码。例如，十进制值 11 作为无符号数，只有一个 4 位的表示，即 $[1011]$ 。这个属性用数学语言来描述就是函数 $B2U_w$ 是一个双射——对于每一个长度为 w 的位向量，都有一个唯一的值与之对应；反过来，在 $0 \sim 2^w - 1$ 之间的每一个整数都有一个唯一的长度为 w 的位向量二进制表示与之对应。

2.2.3 补码编码

对于许多应用，我们还希望表示负数值。最常见的有符号数的计算机表示方式就是补码 (two's-complement) 形式。在这个定义中，将字的最高有效位解释为负权 (negative weight)。我们用函数 $B2T_w$ (Binary to Two's-complement 的缩写，长度为 w) 来表示：

$$B2T_w(\vec{x}) \doteq -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i \quad (2-3)$$

最高有效位 x_{w-1} 也称为符号位，它的“权重”为 -2^{w-1} ，是无符号表示中权重的负数。符号位被设置为 1 时，表示值为负，而当设置为 0 时，值为非负。这里来看一个示例，图 2-12 展示的是下面几种情况下 $B2T$ 给出的从位向量到整数的映射。

$$\begin{aligned} B2T_4([0001]) &= -0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 0 + 0 + 0 + 1 = 1 \\ B2T_4([0101]) &= -0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 0 + 4 + 0 + 1 = 5 \\ B2T_4([1011]) &= -1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = -8 + 0 + 2 + 1 = -5 \\ B2T_4([1111]) &= -1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = -8 + 4 + 2 + 1 = -1 \end{aligned} \quad (2-4)$$

在这个图中，我们用向左指的条表示符号位具有负权重。于是，与一个位向量相关联的数值是由可能的向左指的灰色条和向右指的蓝色条加起来决定的。

我们可以看到，图 2-11 和图 2-12 中的位模式都是一样的，对等式 (2-2) 和等式 (2-4) 来说也是一样，但是当最高有效位是 1 时，数值是不同的，这是因为在一种情况中，最高有效位的权重是 +8，而在另一种情况中，它的权重是 -8。

让我们来考虑一下 w 位补码所能表示的值的范围。它能表示的最小值是位向量 $[10\cdots 0]$ (也就



是设置这个位为负权，但是清除其他所有的位），其整数值为 $TMin_w \doteq -2^{w-1}$ 。而最大值是位向量 $[01\dots1]$ （清除具有负权的位，而设置其他所有的位），其整数值为 $TMax_w \doteq \sum_{i=0}^{w-2} 2^i = 2^{w-1} - 1$ 。以长度为 4 为例，我们有 $TMin_4 = B2T_4([1000]) = -2^3 = -8$ ，而 $TMax_4 = B2T_4([0111]) = 2^2 + 2^1 + 2^0 = 4 + 2 + 1 = 7$ 。

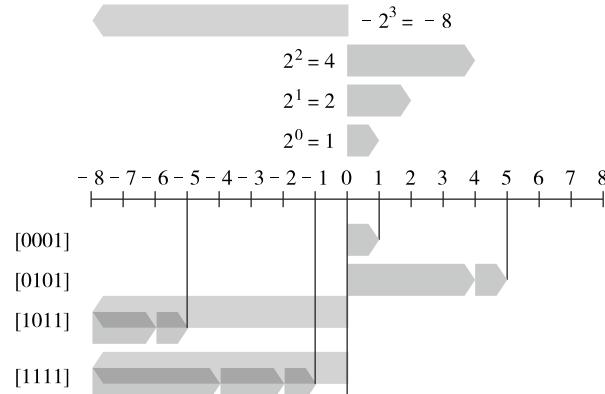


图 2-12 $w=4$ 的补码示例。把位 3 作为符号位，因此当它为 1 时，对数值的影响是 $-2^3 = -8$ 。

这个权重在图中用带向左箭头的条表示

我们可以看出 $B2T_w$ 是一个从长度为 w 的位模式到 $TMin_w$ 和 $TMax_w$ 之间数字的映射，写做 $B2T_w: \{0, 1\}^w \rightarrow \{-2^{w-1}, \dots, 2^{w-1}-1\}$ 。同无符号表示一样，在可表示的取值范围内的每个数字都有一个唯一的 w 位的补码编码。用数学语言来说就是 $B2T_w$ 是一个双射——每个长度为 w 的位向量都对应一个唯一的值；反过来，每个介于 -2^{w-1} 和 $2^{w-1}-1$ 之间的整数都有一个唯一的长度为 w 的位向量二进制表示。

练习题 2.17 假设 $w=4$ ，我们能给每个可能的十六进制数字赋予一个数值，假设用一个无符号或者补码表示。请根据这些表示，通过写出等式 (2-1) 和等式 (2-3) 所示的求和公式中的 2 的非零次幂，填写下表：

\vec{x}		$B2U_4(\vec{x})$	$B2T_4(\vec{x})$
十六进制	二进制		
0xE	[1110]	$2^3+2^2+2^1=14$	$-2^3+2^2+2^1=-2$
0x0			
0x5			
0x8			
0xD			
0xF			

图 2-13 展示了针对不同字长，几个重要数字的位模式和数值。前三个给出的是可表示的整数的范围，用 $UMax_w$ 、 $TMin_w$ 和 $TMax_w$ 来表示。在后面的讨论中，我们还会经常引用到这三个特殊的值。如果可以从上下文中推断出 w ，或者 w 不是讨论的主要内容时，我们会省略下标 w ，直接引用 $UMax$ 、 $TMin$ 和 $TMax$ 。

关于这些数字，有几点值得注意。第一，从图 2-8 和图 2-9 可以看到，补码的范围是不对称的： $|TMin| = |TMax| + 1$ ，也就是说， $TMin$ 没有与之对应的正数。正如我们将会看到的，这导致补码运算的某些特殊的属性，并且容易造成程序中细微的错误。之所以会有这样的不对称性，是因为一半的位模式（符号位设置为 1 的数）表示负数，而一半的数（符号位设置为 0 的数）表示



非负数。因为 0 是非负数，也就意味着能表示的正数比负数少一个。第二，最大的无符号数值刚好比补码的最大值的两倍大一点： $UMax_w = 2 TMax_w + 1$ 。补码表示中所有表示负数的位模式在无符号表示中都变成了正数。图 2-13 也给出了常数 -1 和 0 的表示。注意 -1 和 $UMax$ 有同样的位表示——一个全 1 的串。数值 0 在两种表示方式中都是全 0 的串。

数	字长 w			
	8	16	32	64
$UMax_w$	0xFF 255	0xFFFF 65 535	0xFFFFFFFF 4 294 967 295	0xFFFFFFFFFFFFFF 18 446 744 073 709 551 615
$TMin_w$	0x80 -128	0x8000 -32 768	0x80000000 -2 147 483 648	0x8000000000000000 -9 223 372 036 854 775 808
$TMax_w$	0x7F 127	0x7FFF 32 767	0x7FFFFFFF 2 147 483 647	0x7FFFFFFF 9 223 372 036 854 775 807
-1	0xFF	0xFFFF	0xFFFFFFFF	0xFFFFFFFFFFFFFF 0xFFFFFFFFFFFFFF
0	0x00	0x0000	0x00000000	0x0000000000000000

图 2-13 重要的数字。图中给出了数字和十六进制表示

C 语言标准并没有要求用补码形式来表示有符号整数，但是几乎所有的机器都是这么做的。程序员如果希望代码具有最大可移植性，能够在所有可能的机器上运行，那么除了图 2-10 所示的那些范围之外，我们不应该假设任何可表示的数值范围，也不应该假设有符号数会使用何种特殊的表示方式。另一方面，许多程序的书写都假设用补码来表示有符号数，并且具有图 2-8 和图 2-9 所示的“典型的”取值范围，这些程序在大量的机器和编译器上也有可移植性。C 库中的文件 `<limits.h>` 定义了一组常量，来限定编译器运行的这台机器的不同整型数据类型的取值范围。比如，它定义了常量 `INT_MAX`、`INT_MIN` 和 `UINT_MAX`，它们描述了有符号和无符号整数的范围。对于一个补码的机器，数据类型 `int` 有 w 位，这些常量就对应于 $TMax_w$ 、 $TMin_w$ 和 $UMax_w$ 的值。

确定大小的整数类型

对于某些程序来说，用某个确定大小的表示来编码数据类型非常重要。例如，当编写程序，使得机器能够按照一个标准协议在因特网上通信时，让数据类型与协议指定的数据类型兼容是非常重要的。我们前面看到了，某些 C 数据类型，特别是 `long` 型，在不同的机器上有不同的取值范围，而实际上 C 语言标准只指定了每种数据类型的最小范围，而不是确定的范围。虽然我们可以选择与大多数机器上的标准表示兼容的数据类型，但是这也不能保证可移植性。

ISO C99 标准在文件 `stdint.h` 中引入了另一类整数类型。这个文件定义了一组数据类型，它们的声明形如 `intN_t` 和 `uintN_t`，指定的是 N 位有符号和无符号整数。 N 的具体值与实现相关，但是大多数编译器允许的值为 8、16、32 和 64。因此，通过将它的类型声明为 `uint16_t`，我们可以无歧义地声明一个 16 位无符号变量，而如果声明为 `int32_t`，就是一个 32 位有符号变量。

这些数据类型对应着一组宏，定义了每个 N 的值对应的最小值和最大值。这些宏名字形如 `INTN_MIN`、`INTN_MAX` 和 `UINTN_MAX`。

关于整数数据类型的取值范围和表示，Java 标准是非常明确的。它要求采用补码表示，取值范围与图 2-9 中 64 位的情况一样。在 Java 中，单字节数据类型称为 `byte`，而不是 `char`，而且没有 `long long` 数据类型。这些非常具体的要求都是为了保证无论在什么机器上，Java 程序运



行的表现都能完全一样。

有符号数的其他表示方法

有符号数还有两种标准的表示方法：

反码 (Ones' Complement)：除了最高有效位的权是 $-(2^{w-1}-1)$ 而不是 -2^{w-1} ，它和补码是一样的：

$$B2O_w(\vec{x}) \doteq -x_{w-1}(2^{w-1}-1) + \sum_{i=0}^{w-2} x_i 2^i$$

原码 (Sign-Magnitude)：最高有效位是符号位，用来确定剩下的位应该取负权还是正权：

$$B2S_w(\vec{x}) \doteq (-1)^{x_{w-1}} \cdot \left(\sum_{i=0}^{w-2} x_i 2^i \right)$$

这两种表示方法都有一个奇怪的属性，那就是对于数字 0 有两种不同的编码方式。这两种表示方法，把 [00…0] 都解释为 +0。而值 -0 在原码中表示为 [10…0]，在反码中表示为 [11…1]。虽然过去生产过基于反码表示的机器，但是几乎所有的现代机器都使用补码。我们将看到在浮点数中有使用原码编码。

请注意补码 (Two's complement) 和反码 (Ones' complement) 中撇号的位置是不同的。术语补码来源于这样一个情况，对于非负数 x ，我们用 $2^w - x$ (这里只有一个 2) 来计算 $-x$ 的 w 位表示。术语反码来源于这样一个属性，我们用 [111…1] $-x$ (这里有很多个 1) 来计算 $-x$ 的反码表示。

作为一个示例，考虑下面的代码：

```
1 short x = 12345;
2 short mx = -x;
3
4 show_bytes((byte_pointer) &x, sizeof(short));
5 show_bytes((byte_pointer) &mx, sizeof(short));
```

权	12 345		-12 345		53 191	
	位	值	位	值	位	值
1	1	1	1	1	1	1
2	0	0	1	2	1	2
4	0	0	1	4	1	4
8	1	8	0	0	0	0
16	1	16	0	0	0	0
32	1	32	0	0	0	0
64	0	0	1	64	1	64
128	0	0	1	128	1	128
256	0	0	1	256	1	256
512	0	0	1	512	1	512
1 024	0	0	1	1 024	1	1 024
2 048	0	0	1	2 048	1	2 048
4 096	1	4096	0	0	0	0
8 192	1	8192	0	0	0	0
16 384	0	0	1	16 384	1	16 384
±32 768	0	0	1	-32 768	1	32 768
总计	12 345		-12 345		53 191	

图 2-14 12 345 和 -12 345 的补码表示，以及 53 191 的无符号表示。注意后面两个数有相同的位表示



当在大端法机器上运行时，这段代码的输出为 30 39 和 cf c7，指明 x 的十六进制表示为 0x3039，而 mx 的十六进制表示为 0xCFC7。将它们展开为二进制，我们得到 x 的位模式为 [0011000000111001]，而 mx 的位模式为 [1100111111000111]。如图 2-14 所示，等式 (2-3) 对这两个位模式生成的值为 12345 和 -12345。

练习题 2.18 在第 3 章，我们将看到由反汇编器生成的列表，反汇编器是一种将可执行程序文件转换回可读性更好的 ASCII 码形式的程序。这些文件包含许多十六进制数字，都是用典型的补码形式来表示这些值。能够认识这些数字并理解它们的意义（例如，它们是正数还是负数），是一项重要的技巧。

在下面的列表中，对于标号为 A ~ J（标记在右边）的那些行，将指令名（sub、mov 和 add）右边显示的（用 32 位补码形式表示的）十六进制值转换为等价的十进制值。

8048337: 81 ec b8 01 00 00	sub	\$0x1b8,%esp	A.
804833d: 8b 55 08	mov	0x8(%ebp),%edx	
8048340: 83 c2 14	add	\$0x14,%edx	B.
8048343: 8b 85 58 fe ff ff	mov	0xfffffe58(%ebp),%eax	C.
8048349: 03 02	add	(%edx),%eax	
804834b: 89 85 74 fe ff ff	mov	%eax,0xfffffe74(%ebp)	D.
8048351: 8b 55 08	mov	0x8(%ebp),%edx	
8048354: 83 c2 44	add	\$0x44,%edx	E.
8048357: 8b 85 c8 fe ff ff	mov	0xfffffec8(%ebp),%eax	F.
804835d: 89 02	mov	%eax,(%edx)	
804835f: 8b 45 10	mov	0x10(%ebp),%eax	G.
8048362: 03 45 0c	add	0xc(%ebp),%eax	H.
8048365: 89 85 ec fe ff ff	mov	%eax,0xfffffeec(%ebp)	I.
804836b: 8b 45 08	mov	0x8(%ebp),%eax	
804836e: 83 c0 20	add	\$0x20,%eax	J.
8048371: 8b 00	mov	(%eax),%eax	

2.2.4 有符号数和无符号数之间的转换

C 语言允许在各种不同的数字数据类型之间做强制类型转换。例如，假设变量 x 声明为 int，u 声明为 unsigned。表达式 (unsigned)x 会将 x 的值转换成一个无符号数值，而 (int)u 将 u 的值转换成一个有符号整数。将有符号数强制类型转换成无符号数，或者反过来，会得到什么结果呢？从数学的角度来说，可以想象到几种不同的规则。很明显，对于在两种形式中都能表示的值，我们是想要保持不变的。另一方面，将负数转换成无符号数可能会得到 0。如果转换的无符号数太大以至于超出了补码能够表示的范围，可能会得到 TMax。不过，对于大多数 C 语言的实现来说，对这个问题的回答都是从位级角度来看的，而不是数的角度。

比如说，考虑下面的代码：

```
1 short int v = -12345;
2 unsigned short uv = (unsigned short) v;
3 printf("v = %d, uv = %u\n", v, uv);
```

在一台采用补码的机器上，上述代码会产生如下输出：

```
v = -12345, uv = 53191
```

我们看到，强制类型转换的结果保持位值不变，只是改变了解释这些位的方式。在图 2-14 中我们看到过，-12345 的 16 位补码表示与 53191 的 16 位无符号表示是完全一样的。将 short int 强制类型转换为 unsigned short 改变数值，但是不改变位表示。

类似地，考虑下面的代码：

```
1 unsigned u = 4294967295u; /* UMax_32 */
2 int tu = (int) u;
3 printf("u = %u, tu = %d\n", u, tu);
```



在一台采用补码的机器上，上述代码会产生如下输出：

```
u = 4294967295, tu = -1
```

从图 2-13 我们可以看到，对于 32 位字长来说，无符号形式的 4 294 967 295 ($U_{Max_{32}}$) 和补码形式的 -1 的位模式是完全一样的。将 `unsigned int` 强制类型转换成 `int`，底层的位表示保持不变。

对大多数 C 语言的实现而言，处理同样字长的有符号数和无符号数之间相互转换的一般规则是：数值可能会改变，但是位模式不变。下面我们用更数学化的形式来描述这个规则。既然 $B2U_w$ 和 $B2T_w$ 都是双射，它们就有定义明确的逆映射。将 $U2B_w$ 定义为 $B2U_w^{-1}$ ，而将 $T2B_w$ 定义为 $B2T_w^{-1}$ 。这些函数给出了一个数值的无符号或者补码的位模式。也就是说，给定 $0 \leq x < 2^w$ 范围内的一个整数 x ，函数 $U2B_w(x)$ 会给出 x 的唯一的 w 位无符号表示。相似地，当 x 满足 $-2^{w-1} \leq x < 2^{w-1}$ ，函数 $T2B_w(x)$ 会给出 x 的唯一的 w 位补码表示。可以观察到，对于在范围 $0 \leq x < 2^{w-1}$ 内的值，这两个函数将生成同样的位模式——最高位是 0，因此这个位是正权还是负权就没有关系了。

现在，将函数 $U2T_w$ 定义为 $U2T_w(x) \doteq B2T_w(U2B_w(x))$ 。这个函数的输入是一个 $0 \sim 2^w - 1$ 之间的数，结果得到一个 $-2^{w-1} \sim 2^{w-1} - 1$ 之间的值，这里两个数有相同的位模式，除了参数是无符号的，而结果是以补码表示的。类似地，对于 $-2^{w-1} \sim 2^{w-1} - 1$ 之间的值 x ，函数 $T2U_w$ 定义为 $T2U_w(x) \doteq B2U_w(T2B_w(x))$ ，生成一个数的无符号表示和 x 的补码表示相同。

继续我们前面的例子，从图 2-14 中，我们看到 $T2U_{16}(-12345) = 53191$ ，并且 $U2T_{16}(53191) = -12345$ 。也就是说，十六进制表示写做 0xCFC7 的 16 位位模式既是 -12345 的补码表示，又是 53191 的无符号表示。类似地，从图 2-13 我们看到 $T2U_{32}(-1) = 4 294 967 295$ ，并且 $U2T_{32}(4 294 967 295) = -1$ 。也就是说，无符号表示中的 U_{Max} 有着和补码表示的 -1 相同的位模式。

接下来，我们看到函数 $U2T$ 描述了从无符号数到补码的转换，而 $T2U$ 描述的是补码到无符号的转换。这两个函数描述了在大多数 C 语言实现中这两种数据类型之间的强制类型转换效果。

练习题 2.19 利用你解答练习题 2.17 时填写的表格，填写下列描述函数 $T2U_4$ 的表格。

x	$T2U_4(x)$
-8	
-3	
-2	
-1	
0	
5	

为了更好地理解一个有符号数字 x 和与之对应的无符号数 $T2U_w(x)$ 之间的关系，我们可以利用它们有相同的位表示这一事实，推导出一个数字关系。比较等式 (2-1) 和等式 (2-3)，可以发现对于位模式 \vec{x} ，如果我们计算 $B2U_w(\vec{x}) - B2T_w(\vec{x})$ 之差，从 0 到 $w-2$ 的位的加权和将互相抵消掉，剩下一个值： $B2U_w(\vec{x}) - B2T_w(\vec{x}) = x_{w-1}(2^{w-1} - 2^{w-1}) = x_{w-1}2^w$ 。这就得到一个关系： $B2U_w(\vec{x}) = x_{w-1}2^w + B2T_w(\vec{x})$ 。如果令 $\vec{x} = T2B_w(x)$ ，我们就得到以下公式

$$B2U_w(T2B_w(x)) = T2U_w(x) = x_{w-1}2^w + x \quad (2-5)$$

这个关系对于证明无符号和补码运算之间的关系是很有用的。在 x 的补码表示中，位 x_{w-1} 决定了 x 是否为负，得到

$$T2U_w(x) = \begin{cases} x+2^w, & x < 0 \\ x, & x \geq 0 \end{cases} \quad (2-6)$$



比如说，图 2-15 比较当 $w=4$ 时，函数 $B2U$ 和 $B2T$ 是如何将数值变成位模式的。对补码来说，最高有效位是符号位，我们用带向左箭头的条来表示。对于无符号数来说，最高有效位有正权重，我们用带向右箭头的条来表示。从补码变为无符号数，最高有效位的权重从 -8 变为 $+8$ 。因此，补码表示的负数如果看成无符号数，值会增加 $2^4=16$ 。因而， -5 变成了 $+11$ ，而 -1 变成了 $+15$ 。

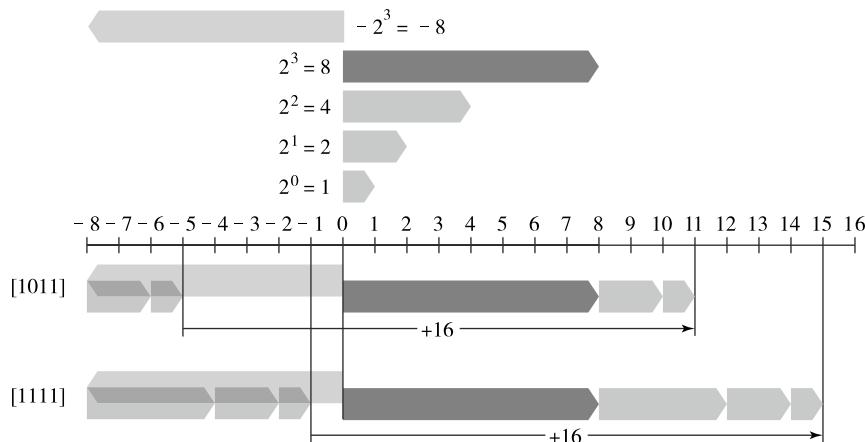


图 2-15 比较当 $w=4$ 时无符号表示和补码表示（对补码和无符号数来说，最高有效位的权重分别是 -8 和 $+8$ ，因而产生一个差为 16 ）

图 2-16 说明了函数 $T2U$ 的一般行为。如图所示，当将一个有符号数映射为它相应的无符号数时，负数就被转换成了大的正数，而非负数会保持不变。

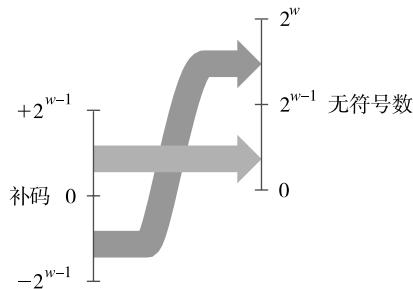


图 2-16 从补码到无符号数的转换。函数 $T2U$ 将负数转换为大的正数

练习题 2.20 请说明在解答练习题 2.19 时生成的表格中，你是如何将等式 (2-6) 应用到其中各项的。

反过来看，我们希望推导出一个无符号数 u 和与之对应的有符号数 $U2T_w(u)$ 之间的关系。如果设 $\bar{u}=U2B_w(u)$ ，我们得到以下公式

$$B2T_w(U2B_w(u)) = U2T_w(u) = -u_{w-1}2^w + u \quad (2-7)$$

在 u 的无符号表示中，位 u_{w-1} 决定了 u 是否大于或者等于 2^{w-1} ，得到

$$U2T_w(u) = \begin{cases} u, & x < 2^{w-1} \\ u - 2^w, & u \geq 2^{w-1} \end{cases} \quad (2-8)$$

图 2-17 说明了这个行为。对于小的数 ($< 2^{w-1}$)，从无符号到有符号的转换将保留数字的原值。对于大的数 ($\geq 2^{w-1}$)，数字将被转换为一个负数值。

总结一下，我们考虑无符号与补码表示之间互相转换的结果。对于在 $0 \leq x < 2^{w-1}$ 范围之内的值 x 而言，我们得到 $T2U_w(x) = x$ 和 $U2T_w(x) = x$ 。也就是说，在这个范围内的数字有相同的无符号和补码表示。对于这个范围以外的数值，转换需要加上或者减去 2^w 。例如，我们有 $T2U_w(-1) = -1 + 2^w = UMax_w$ ——最靠近 0 的负数映射为最大的无符号数。在另一个极端，我们



可以看到 $T2U_w(TMin_w) = -2^{w-1} + 2^w = 2^{w-1} = TMax_w + 1$ ——最小的负数映射为一个刚好在补码的正数范围之外的无符号数。使用图 2-14 的示例，我们能得到 $T2U_{16}(-12345) = 65563 + -12345 = 53191$ 。

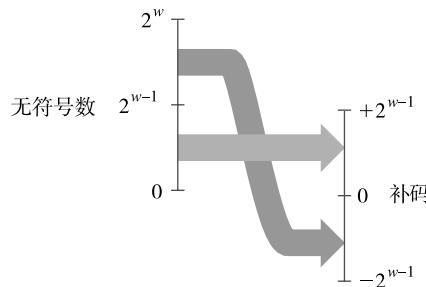


图 2-17 从无符号数到补码的转换。函数 $U2T$ 把大于 $2^{w-1} - 1$ 的数字转换为负值

2.2.5 C 语言中的有符号数与无符号数

如图 2-8 和图 2-9 所示，C 语言支持所有整型数据类型的有符号和无符号运算。尽管 C 语言标准没有指定有符号数要采用某种表示，但是几乎所有的机器都使用补码。通常，大多数数字都默认认为是有符号的。例如，当声明一个像 12345 或者 0x1A2B 这样的常量时，这个值就认为是有符号的。要创建一个无符号常量，必须加上后缀字符 ‘U’ 或者 ‘u’。例如，12345U 或者 0x1A2Bu。

C 语言允许无符号数和有符号数之间的转换。转换的原则是底层的位表示保持不变。因此，在一台采用补码的机器上，当从无符号数转换为有符号数时，效果就是应用函数 $U2T_w$ ，而从有符号数转换为无符号数时，就是应用函数 $T2U_w$ ，其中 w 表示数据类型的位数。

显式的强制类型转换就会导致转换发生，就像下面的代码：

```
1 int tx, ty;
2 unsigned ux, uy;
3
4 tx = (int) ux;
5 uy = (unsigned) ty;
```

另外，当一种类型的表达式被赋值给另外一种类型的变量时，转换是隐式发生的，就像下面的代码：

```
1 int tx, ty;
2 unsigned ux, uy;
3
4 tx = ux; /* Cast to signed */
5 uy = ty; /* Cast to unsigned */
```

当用 `printf` 输出数值时，分别用指示符 %d、%u 和 %x 以有符号十进制、无符号十进制和十六进制格式输出一个数字。注意 `printf` 没有使用任何类型信息，所以它可以用指示符 %u 来输出类型为 `int` 的数值，也可以用指示符 %d 输出类型为 `unsigned` 的数值。例如，考虑下面的代码：

```
1 int x = -1;
2 unsigned u = 2147483648; /* 2 to the 31st */
3
4 printf("x = %u = %d\n", x, x);
5 printf("u = %u = %d\n", u, u);
```



当在一个 32 位机器上运行时，它的输出如下：

```
x = 4294967295 = -1  
u = 2147483648 = -2147483648
```

在这两种情况下，`printf` 首先将这个字当作一个无符号数输出，然后把它当作一个有符号数输出。以下是实际运行中的转换函数： $T2U_{32}(-1) = UMax_{32} = 2^{32}-1$ 和 $U2T_{32}(2^{31}) = 2^{31} - 2^{32} = -2^{31} = TMin_{32}$ 。

由于 C 语言对同时包含有符号和无符号数表达式的这种处理方式，出现了一些奇特的行为。当执行一个运算时，如果它的一个运算数是有符号的而另一个是无符号的，那么 C 语言会隐式地将有符号参数强制类型转换为无符号数，并假设这两个数都是非负的，来执行这个运算。就像我们将要看到的，这种方法对于标准的算术运算来说并无多大差异，但是对于像 `<` 和 `>` 这样的关系运算符来说，它会导致非直观的结果。图 2-18 展示了一些关系表达式的示例以及它们得到的求值结果，这里假设使用的是台采用补码的 32 位机器。考虑比较式 `-1 < 0U`。因为第二个运算数是无符号的，第一个运算数就会被隐式地转换为无符号数，因此表达式就等价于 `4294967295U < 0U`（回想 $T2U_w(-1) = UMax_w$ ），这个答案显然是错的。其他那些示例也可以通过相似的分析来理解。

表达式	类 型	求 值
<code>0 == 0U</code>	无符号	1
<code>-1 < 0</code>	有符号	1
<code>-1 < 0U</code>	无符号	0*
<code>2147483647 > -2147483647-1</code>	有符号	1
<code>2147483647U > -2147483647-1</code>	无符号	0*
<code>2147483647 > (int) 2147483648U</code>	有符号	1*
<code>-1 > -2</code>	有符号	1
<code>(unsigned) -1 > -2</code>	无符号	1

图 2-18 C 语言的升级规则的效果

注：非直观的情况标注了“*”。当一个运算数是无符号的时候，另一个运算数也被隐式强制转换为无符号。将 $TMin_{32}$ 写成 $-2147483647-1$ 的原因请参见网络旁注 DATA:TMIN。

练习题 2.21 假设在采用补码运算的 32 位机器上对这些表达式求值，按照图 2-18 的格式填写下表，描述强制类型转换和关系运算的结果。

表达式	类 型	求 值
<code>-2147483647-1 == 2147483648U</code>		
<code>-2147483647-1 < 2147483647</code>		
<code>-2147483647-1U < 2147483647</code>		
<code>-2147483647-1 < -2147483647</code>		
<code>-2147483647-1U < -2147483647</code>		

网络旁注 DATA:TMIN C 语言中 $TMin$ 的写法

在图 2-18 和练习题 2.21 中，我们很小心地将 $TMin_{32}$ 写成 $-2147483647-1$ 。为什么不简单地写成 -2147483648 或者 $0x80000000$ ？看一下 C 头文件 `limits.h`，注意到它们使用了跟我们写 $TMin_{32}$ 和 $TMax_{32}$ 类似的方法：

```
/* Minimum and maximum values a 'signed int' can hold. */  
#define INT_MAX    2147483647  
#define INT_MIN   (-INT_MAX - 1)
```



不幸的是，补码表示的不对称性和C语言转换规则之间这种奇怪的交互，迫使我们用这种不寻常的方式来写 $TMin_{32}$ 。虽然理解这个问题需要我们钻研C语言标准的一些比较隐晦的角落，但是它能够帮助我们充分领会整数数据类型和表示的一些细微之处。

2.2.6 扩展一个数字的位表示

一种常见的运算是在不同字长的整数之间转换，同时又保持数值不变。当然，当目标数据类型太小以至于不能表示想要的值时，这根本就是不可能的。然而，从一个较小的数据类型转换到一个较大的类型，这应该总是可能的。将一个无符号数转换为一个更大的数据类型，我们只需要简单地在表示的开头添加0，这种运算称为零扩展(zero extension)。将一个补码数字转换为一个更大的数据类型可以执行符号扩展(sign extension)，规则是在表示中添加最高有效位的值的副本。由此可知，如果我们原始值的位表示为 $[x_{w-1}, x_{w-2}, \dots, x_0]$ ，那么扩展后的表示就为 $[x_{w-1}, \dots, x_{w-1}, x_{w-1}, x_{w-2}, \dots, x_0]$ 。(我们用浅灰色标出符号位 x_{w-1} 来突出它们在符号扩展中的角色。)

例如，考虑下面的代码：

```
1 short sx = -12345;      /* -12345 */
2 unsigned short usx = sx; /* 53191 */
3 int x = sx;             /* -12345 */
4 unsigned ux = usx;       /* 53191 */
5
6 printf("sx = %d:\t", sx);
7 show_bytes((byte_pointer) &sx, sizeof(short));
8 printf("usx = %u:\t", usx);
9 show_bytes((byte_pointer) &usx, sizeof(unsigned short));
10 printf("x = %d:\t", x);
11 show_bytes((byte_pointer) &x, sizeof(int));
12 printf("ux = %u:\t", ux);
13 show_bytes((byte_pointer) &ux, sizeof(unsigned));
```

在采用补码表示的32位大端法机器上运行这段代码时，打印出如下输出：

```
sx = -12345: cf c7
usx = 53191: cf c7
x = -12345: ff ff cf c7
ux = 53191: 00 00 cf c7
```

我们看到，尽管-12 345的补码表示和53 191的无符号表示在16位字长时是相同的，但是在32位字长时却是不同的。特别地，-12 345的十六进制表示为0xFFFFFCFC7，而53 191的十六进制表示为0x0000CFC7。前者使用的是符号扩展——最开头加了16位，都是最高有效位1，表示为十六进制就是0xFFFF。后者开头使用16个0来扩展，表示为十六进制就是0x0000。

图2-19给出了从字长 $w=3$ 到 $w=4$ 的符号扩展的结果。位向量[101]表示值 $-4+1=-3$ 。对它应用符号扩展，得到位向量[1101]，表示的值 $-8+4+1=-3$ 。我们可以看到，对于 $w=4$ ，最高两位的组合值是 $-8+4=-4$ ，与 $w=3$ 时符号位的值相同。类似地，位向量[111]和[1111]都表示值-1。

如何证明符号扩展工作是否正确呢？我们想要证明的是

$$B2T_{w+k}(\underbrace{[x_{w-1}, \dots, x_{w-1},}_{k\text{次}} x_{w-1}, x_{w-2}, \dots, x_0]) = B2T_w([x_{w-1}, x_{w-2}, \dots, x_0])$$

这里，在表达式的左边，我们增加了 k 位 x_{w-1} 的副本。下面的证明是对 k 进行的归纳。也就是说，如果我们能够证明符号扩展一位保持数值不变，那么符号扩展任意位都能保持这种属性。因此，证明的任务就变为证明以下等式：



$B2T_{w+1}([x_{w-1}, x_{w-1}, x_{w-2}, \dots, x_0]) = B2T_w([x_{w-1}, x_{w-2}, \dots, x_0])$
用等式 (2-3) 展开左边的表达式, 得到:

$$\begin{aligned} B2T_{w+1}([x_{w-1}, x_{w-1}, x_{w-2}, \dots, x_0]) &= -x_{w-1}2^w + \sum_{i=0}^{w-1} x_i 2^i \\ &= -x_{w-1}2^w + x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i \\ &= -x_{w-1}(2^w - 2^{w-1}) + \sum_{i=0}^{w-2} x_i 2^i \\ &= -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i \\ &= B2T_w([x_{w-1}, x_{w-2}, \dots, x_0]) \end{aligned}$$

我们使用的关键属性是 $2^w - 2^{w-1} = 2^{w-1}$ 。因此, 加上一个权值为 -2^w 的位, 和将一个权值为 -2^{w-1} 的位转换为一个权值为 2^{w-1} 的位, 这两项运算的综合效果就会保持原始的数值。

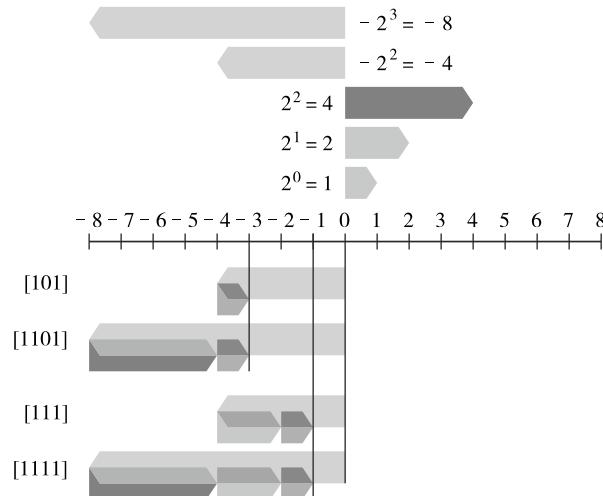


图 2-19 从 $w=3$ 到 $w=4$ 的符号扩展示例。对于 $w=4$, 最高两位组合权重为 $-8 + 4 = -4$, 与 $w=3$ 时的符号位的权重一样

练习题 2.22 应用等式 (2-3), 证明下列每个位向量都是 -5 的补码表示。

- A. [1011]
- B. [11011]
- C. [111011]

可以看到第二个和第三个位向量可以通过对第一个位向量做符号扩展得到。

值得一提的是, 从一个数据大小到另一个数据大小的转换, 以及无符号和有符号数字之间的转换的相对顺序能够影响一个程序的行为。考虑下面的代码:

```
1     short sx = -12345;      /* -12345 */
2     unsigned uy = sx;        /* Mystery! */
3
4     printf("uy = %u:\t", uy);
5     show_bytes((byte_pointer) &uy, sizeof(unsigned));
```



在一台大端法机器上，这部分代码产生如下输出：

```
uy = 4294954951: ff ff cf c7
```

这表明当把 `short` 转换成 `unsigned` 时，我们先要改变大小，之后再完成从有符号到无符号的转换。也就是说 `(unsigned) sx` 等价于 `(unsigned)(int) sx`，求值得到 4 294 954 951，而不等价于 `(unsigned)(unsigned short) sx`，后者求值得到 53 191。事实上，这个规则是 C 语言标准要求的。

练习题 2.23 考虑下面的 C 函数：

```
int fun1(unsigned word) {
    return (int) ((word << 24) >> 24);
}

int fun2(unsigned word) {
    return ((int) word << 24) >> 24;
}
```

假设在一个采用补码运算的 32 位字长的机器上执行这些函数。还假设有符号数值的右移是算术右移，而无符号数值的右移是逻辑右移。

A. 填写下表，说明这些函数对几个示例参数的结果。你会发现用十六进制表示来做会更方便，只要记住十六进制数字 8 到 F 的最高有效位等于 1。

w	fun1(w)	fun2(w)
0x00000076		
0x87654321		
0x000000C9		
0xEDCBA987		

B. 用语言来描述这些函数执行的有用的计算。

2.2.7 截断数字

假设我们不用额外的位来扩展一个数值，而是减少表示一个数字的位数。例如下面代码中这种情况：

```
1     int x = 53191;
2     short sx = (short) x; /* -12345 */
3     int y = sx;           /* -12345 */
```

在一台典型的 32 位机器上，当把 `x` 强制类型转换为 `short` 时，我们就将 32 位的 `int` 截断为 16 位的 `short int`。就像前面所看到的，这个 16 位的位模式就是 -12 345 的补码表示。当我们把它强制类型转换回 `int` 时，符号扩展把高 16 位设置为 1，从而生成 -12 345 的 32 位补码表示。

将一个 w 位的数 $\vec{x}=[x_{w-1}, x_{w-2}, \dots, x_0]$ 截断为一个 k 位数字时，我们会丢弃高 $w-k$ 位，得到一个位向量 $\vec{x}'=[x_{k-1}, x_{k-2}, \dots, x_0]$ 。截断一个数字可能会改变它的值——溢出的一种形式。我们现在来研究什么样的数值会产生这种情况。对于一个无符号数字 x ，截断它到 k 位的结果就相当于计算 $x \bmod 2^k$ 。通过对等式 (2-1) 应用取模运算就可以得到：



$$\begin{aligned}B2U_w([x_{w-1}, x_{w-2}, \dots, x_0]) \bmod 2^k &= \left[\sum_{i=0}^{w-1} x_i 2^i \right] \bmod 2^k \\&= \left[\sum_{i=0}^{k-1} x_i 2^i \right] \bmod 2^k \\&= \sum_{i=0}^{k-1} x_i 2^i \\&= B2U_k([x_{k-1}, x_{k-2}, \dots, x_0])\end{aligned}$$

在这段推导中，我们利用的属性是：对于任何 $i < k$, $2^i \bmod 2^k = 0$ 和 $\sum_{i=0}^{k-1} x_i 2^i = \sum_{i=0}^{k-1} 2^i = 2^k - 1 < 2^k$ 。

对于一个补码数字 x , 相似的推理表明 $B2T_w([x_{w-1}, x_{w-2}, \dots, x_0]) \bmod 2^k = B2U_k[x_{k-1}, x_{k-2}, \dots, x_0]$ 。也就是说， $x \bmod 2^k$ 能够被一个位级表示为 $[x_{k-1}, \dots, x_0]$ 的无符号数表示。不过，一般而言，我们将被截断的数字视为有符号的。这将得到数值 $U2T_k(x \bmod 2^k)$ 。

总而言之，无符号数的截断结果是：

$$B2U_k([x_{k-1}, x_{k-2}, \dots, x_0]) = B2U_w([x_{w-1}, x_{w-2}, \dots, x_0]) \bmod 2^k, \quad (2-9)$$

而补码数字的截断结果是：

$$B2T_k([x_{k-1}, x_{k-2}, \dots, x_0]) = U2T_k(B2U_w([x_{w-1}, x_{w-2}, \dots, x_0]) \bmod 2^k) \quad (2-10)$$

练习题 2.24 假设将一个 4 位数值（用十六进制数字 0 ~ F 表示）截断到一个 3 位数值（用十六进制数字 0 ~ 7 表示）。填写下表，根据那些位模式的无符号和补码解释，说明这种截断对某些情况的结果。

十六进制		无符号		补码	
原始值	截断值	原始值	截断值	原始值	截断值
0	0	0		0	
2	2	2		2	
9	1	9		-7	
B	3	11		-5	
F	7	15		-1	

解释如何将等式 (2-9) 和等式 (2-10) 应用到这些示例上。

2.2.8 关于有符号数与无符号数的建议

就像我们看到的那样，有符号数到无符号数的隐式强制类型转换导致了某些非直观的行为。而这些非直观的特性经常导致程序错误，并且这种包含隐式强制类型转换细微差别的错误很难被发现。因为这种强制类型转换是在代码中没有明确指示的情况下发生的，程序员经常忽视了它的影响。

下面两个练习题说明了某些由于隐式强制类型转换和无符号数据类型造成的细微错误。

练习题 2.25 考虑下列代码，这段代码试图计算数组 a 中所有元素的和，其中元素的数量由参数 length 给出。

```
1  /* WARNING: This is buggy code */
2  float sum_elements(float a[], unsigned length) {
3      int i;
4      float result = 0;
5
6      for (i = 0; i <= length-1; i++)
7          result += a[i];
8      return result;
9 }
```

当参数 length 等于 0 时，运行这段代码应该返回 0.0。但实际上，运行时会遇到一个存储器错误。请解释为什么会发生这样的情况，并且说明如何修改代码。



练习题 2.26 现在给你一个任务，写一个函数用来判定一个字符串是否比另一个更长。前提是你要用字符串库函数 `strlen`，它的声明如下：

```
/* Prototype for library function strlen */
size_t strlen(const char *s);
```

最开始你写的函数是这样的：

```
/* Determine whether string s is longer than string t */
/* WARNING: This function is buggy */
int strlonger(char *s, char *t) {
    return strlen(s) - strlen(t) > 0;
}
```

当你在一些示例数据上测试这个函数时，一切似乎都是正确的。进一步研究发现在头文件 `stdio.h` 中数据类型 `size_t` 是定义成 `unsigned int` 的。

- A. 在什么情况下，这个函数会产生不正确的结果？
- B. 解释为什么会出现这样不正确的结果。
- C. 说明如何修改这段代码好让它能可靠地工作。

函数 `getpeername` 的安全漏洞

2002 年，从事 FreeBSD 开源操作系统项目的程序员意识到，他们对 `getpeername` 函数的实现存在安全漏洞。代码的简化版本如下：

```
1  /*
2   * Illustration of code vulnerability similar to that found in
3   * FreeBSD's implementation of getpeername()
4   */
5
6  /* Declaration of library function memcpy */
7  void *memcpy(void *dest, void *src, size_t n);
8
9  /* Kernel memory region holding user-accessible data */
10 #define KSIZE 1024
11 char kbuf[KSIZE];
12
13 /* Copy at most maxlen bytes from kernel region to user buffer */
14 int copy_from_kernel(void *user_dest, int maxlen) {
15     /* Byte count len is minimum of buffer size and maxlen */
16     int len = KSIZE < maxlen ? KSIZE : maxlen;
17     memcpy(user_dest, kbuf, len);
18     return len;
19 }
```

在这段代码里，第 7 行给出的是库函数 `memcpy` 的原型，这个函数是要将一段指定长度为 `n` 的字节从存储器的一个区域复制到另一个区域。

从第 14 行开始的函数 `copy_from_kernel` 是要将一些操作系统内核维护的数据复制到指定的用户可以访问的存储器区域。对用户来说，大多数内核维护的数据结构应该是不可读的，因为这些数据结构可能包含其他用户和系统上运行的其他作业的敏感信息，但是显示为 `kbuf` 的区域是用户可以读的。参数 `maxlen` 给出的是分配给用户的缓冲区的长度，这个缓冲区是用参数 `user_dest` 指示的。然后，第 16 行的计算确保复制的字节数据不会超出源或者目标缓冲区可



用的范围。

不过，假设有些怀有恶意的程序员在调用 `copy_from_kernel` 的代码中对 maxlen 使用了负数值，那么，第 16 行的最小值计算会把这个值赋给 len，然后 len 会作为参数 n 被传递给 `memcpy`。不过，请注意参数 n 是被声明为数据类型 `size_t` 的。这个数据类型是在库文件 `stdio.h` 中（通过 `typedef`）被声明的。典型地，在 32 位机器上被定义为 `unsigned int`。既然参数 n 是无符号的，那么 `memcpy` 会把它当作一个非常大的正整数，并且试图将这样多字节的数据从内核区域复制到用户的缓冲区。虽然复制这么多字节（至少 2^{31} 个）实际上不会完成，因为程序会遇到进程中非法地址的错误，但是程序还是能读到没有被授权的内核存储器区域。

我们可以看到，这个问题是由于数据类型的不匹配造成的：在一个地方，长度参数是有符号数；而另一个地方，它又是无符号数。正如这个例子表明的那样，这样的不匹配会成为缺陷的原因，甚至会导致安全漏洞。幸运的是，还没有案例报告有程序员在 FreeBSD 上利用了这个漏洞。他们发布了一个安全建议，“FreeBSD-SA-02:38.signed-error”，建议系统管理员如何应用补丁消除这个漏洞。要修正这个缺陷，只要将 `copy_from_kernel` 的参数 maxlen 声明为类型 `size_t`，也就是与 `memcpy` 的参数 n 一致。同时，我们也应该将本地变量 len 和返回值声明为 `size_t`。

我们已经看到了由于许多无符号运算的细微特性，尤其是有符号数到无符号数的隐式转换，会导致错误或者漏洞的方式。避免这类错误的一种方法就是绝不使用无符号数。实际上，除了 C 以外，很少有语言支持无符号整数。很明显，这些语言的设计者认为它们带来的麻烦要比益处多得多。例如，Java 只支持有符号整数，并且要求用补码运算来实现。正常的右移运算符 `>>` 被定义为执行算术右移。特殊的运算符 `>>>` 被指定为执行逻辑右移。

当我们想要把字仅仅看做是位的集合，并且没有任何数字意义时，无符号数值是非常有用的。例如，往一个字中放入描述各种布尔条件的标记（flag）时，就是这样。地址自然地就是无符号的，所以系统程序员发现无符号类型是很有帮助的。当实现模运算和多精度运算的数学包时，数字是由字的数组来表示的，无符号值也会非常有用。

2.3 整数运算

许多刚入门的程序员非常惊奇地发现，两个正数相加会得出一个负数，并且比较表达式 $x < y$ 和比较表达式 $x - y < 0$ 会产生不同的结果。这些属性是由于计算机运算的有限性造成的。理解计算机运算的细微之处能够帮助程序员编写更可靠的代码。

2.3.1 无符号加法

考虑两个非负整数 x 和 y ，满足 $0 \leq x, y \leq 2^w - 1$ 。每个数都能表示为 w 位无符号数字。然而，如果计算它们的和，我们就会有一个可能的范围 $0 \leq x + y \leq 2^{w+1} - 2$ 。表示这个和可能需要 $w + 1$ 位。例如，图 2-20 展示了当 x 和 y 有 4 位表示时，函数 $x + y$ 的坐标图。参数（显示在水平轴上）取值范围为 $0 \sim 15$ ，但是和的取值范围为 $0 \sim 30$ 。函数的形状是一个有坡度的平面（在两个维度上，函数都是线性的）。如果保持和为一个 $w+1$ 位的数字，并且用它加上另外一个数值，我们可能需要 $w+2$ 位，以此类推。这种持续的“字长膨胀”意味着，要想完整地表示算术运算的结果，我们不能对字长做任何限制。一些编程语言，例如 Lisp，实际上就支持无限精度的运算，允许任意的（当然，要在机器的存储器限制之内）整数运算。更常见的是，编程语言支持固定精度的运算，因此像“加法”和“乘法”这样的运算不同于它们在整数上的相应运算。

无符号运算可以被视为一种模运算形式。无符号加法等价于计算和模上 2^w 。可以通过简单的丢弃 $x+y$ 的 $w+1$ 位表示的最高位，来计算这个数值。比如，考虑一个 4 位数字表示， $x=9$ 和 $y=12$ 的位表示分别为 [1001] 和 [1100]。它们的和是 21，5 位的表示为 [10101]。但是如果丢弃最



高位，我们就得到 [0101]，也就是十进制值的 5。这就和值 $21 \bmod 16 = 5$ 一致。

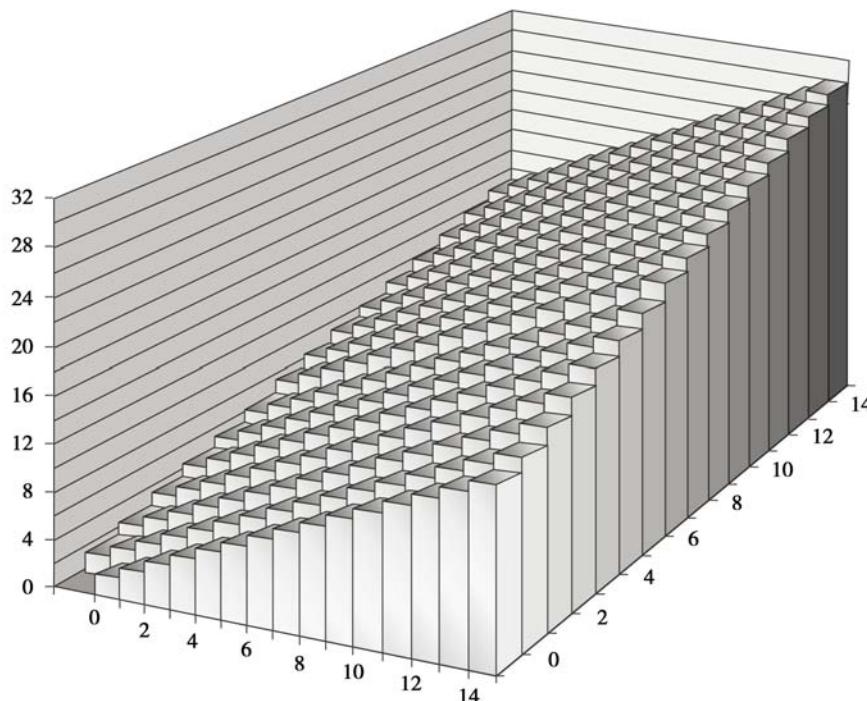


图 2-20 整数加法。对于一个 4 位的字长，其和可能需要 5 位

一般而言，我们可以看到，如果 $x + y < 2^w$ ，则和的 $w+1$ 位表示中的最高位会等于 0，因此丢弃它不会改变这个数值。另一方面，如果 $2^w - x - y < 2^{w+1}$ ，则和的 $w+1$ 位表示中的最高位会等于 1，因此丢弃它就相当于从和中减去了 2^w 。图 2-21 说明了这两种情况。这样会得到一个范围 $0 \leq x + y - 2^w < 2^{w+1} - 2^w = 2^w$ 中的值，刚好等于 x 与 y 的和模 2^w 的结果。现在定义参数 x 和 y 的运算 $+_w^u$ ，这里 $0 \leq x, y < 2^w$ ，如下：

$$x +_w^u y = \begin{cases} x + y, & x + y < 2^w \\ x + y - 2^w, & 2^w \leq x + y < 2^{w+1} \end{cases} \quad (2-11)$$

这正好是在 C 中执行两个 w 位无符号数值加法时得到的结果。

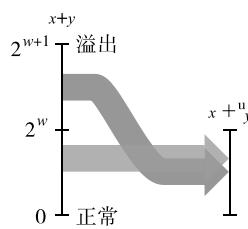


图 2-21 整数加法和无符号加法间的关系。当 $x+y$ 大于 $2^w - 1$ 时，其和溢出

一个算术运算溢出，是指完整的整数结果不能放到数据类型的字长限制中去。如等式 (2-11) 所示，当两个运算数的和为 2^w 或者更大时，就发生了溢出。图 2-22 展示了字长 $w = 4$ 的无符号加法函数的坐标图。这个和是按模 $2^4=16$ 计算的。当 $x+y < 16$ 时，没有溢出，并且 $x +_4^u y$ 就是 $x+y$ ，这对应于图中标记为“正常”的斜面。当 $x+y = 16$ 时，加法溢出，结果相当于从和中减去 16，这对应于图中标记为“溢出”的斜面。

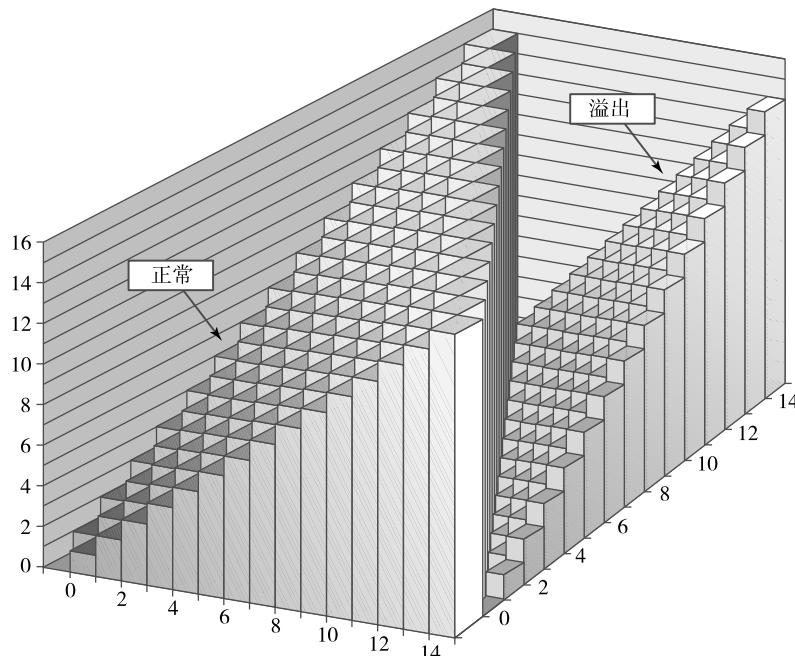


图 2-22 无符号加法（4位字长，加法是模 16 的）

当执行 C 程序时，不会将溢出作为错误而发信号。不过有的时候，我们可能希望判定是否发生了溢出。比如，假设计算 $s \doteq x +^u y$ ，并且我们想要判定 s 是否等于 $x + y$ 。我们声称当且仅当 $s < x$ （或者等价地 $s < y$ ）时，发生了溢出。要明白这一点，请注意 $x + y \geq x$ ，因此如果 s 没有溢出，我们能够肯定 $s = x$ 。另一方面，如果 s 确实溢出了，我们就有 $s = x + y - 2^w$ 。假设 $y < 2^w$ ，我们就有 $y - 2^w < 0$ ，因此 $s = x + (y - 2^w) < x$ 。在前面的示例中，我们看到 $9 +_4^u 12 = 5$ 。由于 $5 < 9$ ，我们可以看出发生了溢出。

练习题 2.27 写出一个具有如下原型的函数：

```
/* Determine whether arguments can be added without overflow */  
int uadd_ok(unsigned x, unsigned y);
```

如果参数 x 和 y 相加不会产生溢出，这个函数就返回 1。

模数加法形成了一种数学结构，称为阿贝尔群（Abelian group），这是以丹麦数学家 Niels Henrik Abel (1802 ~ 1829) 的名字命名。也就是说，它是可交换的（这就是为什么叫“abelian”的地方）和可结合的。它有一个单位元 0，并且每个元素有一个加法逆元。考虑 w 位的无符号数的集合，执行加法运算 $+_w^u$ 。对于每个值 x ，必然有某个值 $-_w^u x$ 满足 $-_w^u x +_w^u x = 0$ 。当 $x = 0$ 时，加法逆元显然是 0。对于 $x > 0$ ，考虑值 $2^w - x$ 。我们观察到这个数字在 $0 \leq 2^w - x < 2^w$ 范围之内，并且 $(x + 2^w - x) \bmod 2^w = 2^w \bmod 2^w = 0$ 。因此，它就是 x 在 $+_w^u$ 下的逆元。这两种情况就导出了对于 $0 \leq x < 2^w$ 的等式：

$$-_w^u x = \begin{cases} x, & x = 0 \\ 2^w - x, & x > 0 \end{cases} \quad (2-12)$$

练习题 2.28 我们能用一个十六进制数字来表示长度 $w=4$ 的位模式。对于这些数字的无符号解释，使用等式 (2-12) 填写下表，给出所示数字的无符号加法逆元的位表示（用十六进制形式）。



十六进制	x	十进制	$\frac{u}{4}x$	十六进制
0				
5				
8				
D				
F				

2.3.2 补码加法

对于补码加法，我们必须确定当结果太大（为正）或者太小（为负）时，应该做些什么。给定在范围 $-2^{w-1} \leq x, y \leq 2^{w-1}-1$ 之内的整数值 x 和 y ，它们的和就在范围 $-2^w \leq x+y \leq 2^w-2$ 之内，要想准确表示，可能需要 $w+1$ 位。就像以前一样，我们通过将表示截断到 w 位，来避免数据大小的不断扩张。然而，结果却不像模数加法那样在数学上感觉很熟悉。

两个数的 w 位补码之和与无符号之和有完全相同的位级表示。实际上，大多数计算机使用同样的机器指令来执行无符号或者有符号加法。因此，我们能够定义字长为 w 的、运算数 x 和 y 上的补码加法 (x 和 y 满足 $-2^{w-1} \leq x, y \leq 2^{w-1}$)，表示为 $+_w^t$ ：

$$x +_w^t y = U2T_w(T2U_w(x) +_w^u T2U_w(y)) \quad (2-13)$$

根据等式 (2-5)，我们可以把 $T2U_w(x)$ 写成 $x_{w-1}2^w + x$ ，把 $T2U_w(y)$ 写成 $y_{w-1}2^w + y$ 。使用属性，即 $+_w^u$ 是模 2^w 的加法，以及模数加法的属性，我们就能得到：

$$\begin{aligned} x +_w^t y &= U2T_w(T2U_w(x) +_w^u T2U_w(y)) \\ &= U2T_w[(x_{w-1}2^w + x) + (y_{w-1}2^w + y) \bmod 2^w] \\ &= U2T_w[(x + y) \bmod 2^w] \end{aligned}$$

消除了 $x_{w-1}2^w$ 和 $y_{w-1}2^w$ 这两项，因为它们模 2^w 等于 0。

为了更好地理解这个数量，定义 z 为整数和 $z' \triangleq x+y$ ， z' 为 $z' \triangleq z \bmod 2^w$ ，而 z'' 为 $z'' \triangleq U2T_w(z')$ 。数值 z'' 等于 $x +_w^t y$ 。我们分成 4 种情况分析，如图 2-23 所示。

1) $-2^w \leq z < -2^{w-1}$ 。然后，我们会有 $z' = z + 2^w$ 。这就得出 $0 \leq z' < -2^{w-1} + 2^w = 2^{w-1}$ 。检查等式 (2-8)，我们看到 z' 在满足 $z'' = z'$ 的范围之内。这种情况称为负溢出 (negative overflow)。我们将两个负数 x 和 y 相加（这是我们能得到 $z < -2^{w-1}$ 的唯一方式），得到一个非负的结果 $z'' = x + y + 2^w$ 。

2) $-2^{w-1} \leq z < 0$ 。那么，我们又将有 $z' = z + 2^w$ ，得到 $-2^{w-1} + 2^w = 2^{w-1} \leq z' < 2^w$ 。检查等式 (2-8)，我们看到 z' 在满足 $z'' = -2^w$ 的范围之内，因此 $z'' = z' - 2^w = z + 2^w - 2^w = z$ 。也就是说，我们的补码和 z'' 等于整数和 $x+y$ 。

3) $0 \leq z < 2^{w-1}$ 。那么，我们将有 $z' = z$ ，得到 $0 \leq z' < 2^{w-1}$ ，因此 $z'' = z' = z$ 。补码和 z'' 又等于整数和 $x+y$ 。

4) $2^{w-1} \leq z < 2^w$ 。我们又将有 $z' = z$ ，得到 $2^{w-1} \leq z' < 2^w$ 。但是在这个范围内，我们有 $z'' = z' - 2^w$ ，得到 $z'' = x + y - 2^w$ 。这种情况称为正溢出 (positive overflow)。我们将正数 x 和 y 相加（这

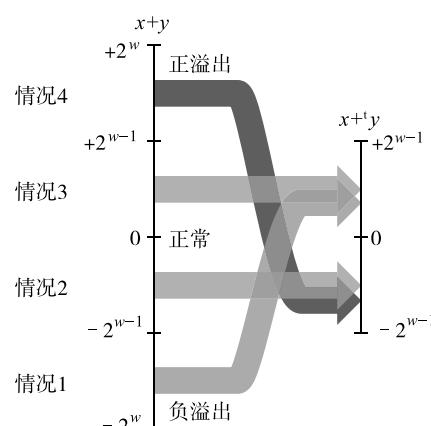


图 2-23 整数和补码加法之间的关系。当 $x+y$ 小于 -2^{w-1} 时，产生负溢出。当它大于 $2^{w-1}+1$ 时，产生正溢出



是我们能得到 $z = 2^{w-1}$ 的唯一方式), 得到一个负数结果 $z'' = x + y - 2^w$ 。

通过前面的分析可以证明, 范围在 $-2^{w-1} \leq x, y \leq 2^{w-1}-1$ 之内的 x 和 y 实施运算 $+_w^t$ 时, 我们有下面这样的式子:

$$x +_w^t y = \begin{cases} x + y - 2^w, & 2^{w-1} \leq x + y \quad \text{正溢出} \\ x + y, & -2^{w-1} \leq x + y < 2^{w-1} \quad \text{正常} \\ x + y + 2^w, & x + y < -2^{w-1} \quad \text{负溢出} \end{cases} \quad (2-14)$$

图 2-24 展示了一些 4 位补码加法的示例作为说明。每个示例的情况都被标号为对应于等式 (2-14) 的推导过程中的情况。注意 $2^4 = 16$, 因此负溢出得到的结果比整数和大 16, 而正溢出得到的结果比之小 16。我们包括了运算数和结果的位级表示。可以观察到, 能够通过对运算数执行二进制加法并将结果截断到 4 位, 从而得到结果。

x	y	$x + y$	$x +_4^t y$	情况
-8 [1000]	-5 [1011]	-13 [10011]	3 [0011]	1
-8 [1000]	-8 [1000]	-16 [10000]	0 [0000]	1
-8 [1000]	5 [0101]	-3 [11101]	-3 [1101]	2
2 [0010]	5 [0101]	7 [00111]	7 [0111]	3
5 [0101]	5 [0101]	10 [01010]	-6 [1010]	4

图 2-24 补码加法示例。通过执行运算数的二进制加法并将结果截断到 4 位, 可以获得 4 位补码和的位级表示

图 2-25 阐明了字长 $w = 4$ 的补码加法。运算数的范围为 $-8 \sim 7$ 。当 $x + y < -8$ 时, 补码加法就会负溢出, 导致和增加了 16。当 $-8 \leq x + y < 8$ 时, 加法就产生 $x + y$ 。当 $x + y = 8$, 加法就会正溢出, 使得和减少了 16。这三种情况中的每一种都形成了图中的一个斜面。

等式 (2-14) 也让我们认出了哪些情况下会发生溢出。当 x 和 y 都是负数, 但是 $x +_w^t y = 0$ 时, 我们就会得到负溢出。当 x 和 y 都是正数, 但是 $x +_w^t y < 0$ 时, 我们会得到正溢出。

练习题 2.29 按照图 2-24 的形式填写下表。分别列出 5 位参数的整数值、整数和的数值、补码和的数值、补码和的位级表示, 以及属于等式 (2-14) 推导中的哪种情况。

x	y	$x + y$	$x +_5^t y$	情况
[10100]	[10001]			
[11000]	[11000]			
[10111]	[01000]			
[00010]	[00101]			
[01100]	[00100]			

练习题 2.30 写出一个具有如下原型的函数：

```
/* Determine whether arguments can be added without overflow */  
int tadd_ok(int x, int y);
```



如果参数 x 和 y 相加不会产生溢出，这个函数就返回 1。

练习题 2.31 你的同事对你补码加法溢出条件的分析有些不耐烦了，他给出了一个函数 `tadd_ok` 的实现，如下所示：

```
/* Determine whether arguments can be added without overflow */  
/* WARNING: This code is buggy. */  
int tadd_ok(int x, int y) {  
    int sum = x+y;  
    return (sum-x == y) && (sum-y == x);  
}
```

你看了代码以后笑了。解释一下为什么。

练习题 2.32 你现在有个任务，编写函数 `tsub_ok` 的代码，函数的参数是 x 和 y ，如果计算 $x-y$ 不产生溢出，函数就返回 1。假设你写的练习题 2.30 的代码如下所示：

```
/* Determine whether arguments can be subtracted without overflow */  
/* WARNING: This code is buggy. */  
int tsub_ok(int x, int y) {  
    return tadd_ok(x, -y);  
}
```

x 和 y 取什么值时，这个函数会产生错误的结果？写一个该函数的正确版本（家庭作业 2.74）。

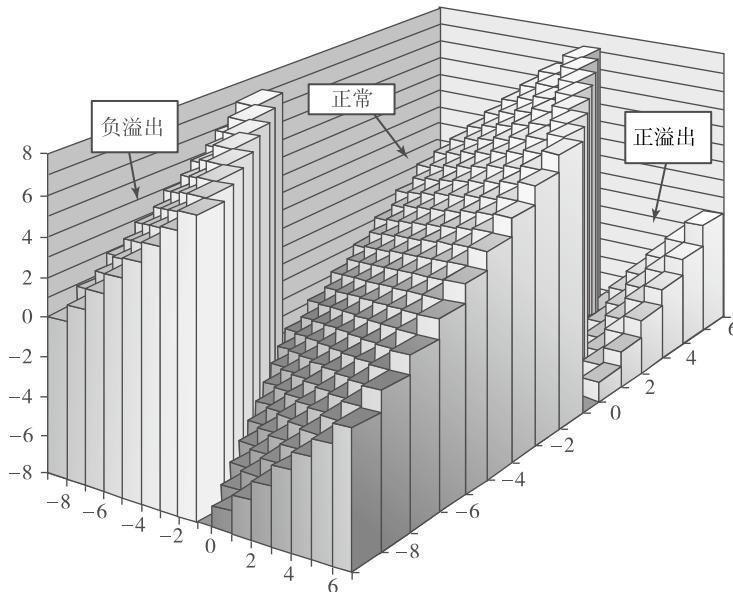


图 2-25 补码加法（字长为 4 位的情况下，当 $x+y < -8$ 时，产生负溢出； $x+y > 8$ 时，会产生正溢出）

2.3.3 补码的非

我们可以看到范围在 $-2^{w-1} \leq x < 2^{w-1}$ 中的每个数字 x 都有 $+_w^t$ 下的加法逆元。首先，对于 $x \neq -2^{w-1}$ ，我们可以看到它的加法逆元就是 $-x$ 。也就是，我们有 $-2^{w-1} < -x < 2^{w-1}$ 和 $-x+_w^t x = -x + x = 0$ 。另一方面，对于 $x = -2^{w-1} = TMin_w$ ， $-x = 2^{w-1}$ 不能表示为一个 w 位的数。我们声明，这个特殊值本身就是它在 $+_w^t$ 下的加法逆元。 $-2^{w-1}+_w^t-2^{w-1}$ 的值由等式 (2-14) 的第三种情况给出，因为 $-2^{w-1} + -2^{w-1} = -2^w$ 。最后得出 $-2^{w-1}+_w^t-2^{w-1} = -2^w + 2^w = 0$ 。从这个分析中，我们可以定义对于范围 $-2^{w-1} \leq x < 2^{w-1}$ 内的 x ，补码的非运算 (negation operation) $-_w^t$ 如下：

$$-_w^t x = \begin{cases} -2^{w-1}, & x = -2^{w-1} \\ -x, & x > -2^{w-1} \end{cases} \quad (2-15)$$



练习题 2.33 我们可以用一个十六进制数字来表示长度 $w=4$ 的位模式。根据这些数字的补码的解释，填写下表，确定所示数字的加法逆元。

x		${}_{-4}^t x$	
十六进制	十进制	十进制	十六进制
0			
5			
8			
D			
F			

对于补码和无符号（见练习题 2.28）非（negation）产生的位模式，你观察到什么？

网络旁注 DATA : TNNEG : 补码非的位级表示

计算一个位级表示的值的补码非有几种聪明的方法。这些技术很有用（例如当你在调试程序的时候遇到值 0xfffffffffa），同时它们也能够让你更了解补码表示的本质。

行位级补码非的第一种方法是对每一位求补，再对结果加 1。在 C 语言中，我们可以确定，对于任意整数值 x ，计算表达式 $-x$ 和 $\sim x + 1$ 得到的结果完全一样。

下面是一些字长为 4 的示例：

\vec{x}	$\sim \vec{x}$	$incr(\sim \vec{x})$
[0101]	5	[1010] -6
[0111]	7	[1000] -8
[1100]	-4	[0011] 3
[0000]	0	[1111] -1
[1000]	-8	[0111] 7

从前面的例子我们知道 0xf 的补是 0x0，而 0xa 的补是 0x5，因而 0xfffffffffa 是 -6 的补码表示。

计算一个数 x 的补码非的第二种方法是建立在将位向量分为两部分的基础之上的。假设 k 是最右边的 1 的位置，因而 x 的位级表示形如 $[x_{w-1}, x_{w-2}, \dots, x_{k+1}, 1, 0, \dots, 0]$ 。（只要 $x \neq 0$ 就能够找到这样的 k 。）这个值的非写成二进制格式就是 $[\sim x_{w-1}, \sim x_{w-2}, \dots, \sim x_{k+1}, 1, 0, \dots, 0]$ 。也就是说，我们对位位置 k 左边的所有位取反。

我们用一些 4 位数字来说明这个方法，这里用斜体来突出最右边的模式 1, 0, ..., 0 :

x	$\sim x$
[1100]	-4
[1000]	-8
[0101]	5
[0111]	7

2.3.4 无符号乘法

范围在 $0 \leq x, y \leq 2^w - 1$ 内的整数 x 和 y 可以表示为 w 位的无符号数，但是它们的乘积 $x \cdot y$ 的取值范围为 0 到 $(2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$ 之间。这可能需要 $2w$ 位来表示。不过，C 语言中的无符号乘法被定义为产生 w 位的值，就是 $2w$ 位的整数乘积的低 w 位表示的值。根据等式 (2-9)，这可以看作等价于计算乘积模 2^w 。因此， w 位无符号乘法运算 $*_w^u$ 的结果为：

$$x *_w^u y = (x \cdot y) \bmod 2^w \quad (2-16)$$

2.3.5 补码乘法

范围在 $-2^{w-1} \leq x, y \leq 2^{w-1} - 1$ 内的整数 x 和 y 可以表示为 w 位的补码数字，但是它们的乘积 $x \cdot y$ 的取值范围在 $-2^{w-1} \cdot (2^{w-1} - 1) = -2^{2w-2} + 2^{w-1}$ 和 $-2^{w-1} \cdot -2^{w-1} = -2^{2w-2}$ 之间。要用补码来表示这



个乘积，可能需要 $2w$ 位——大多数情况下只需要 $2w-1$ 位，但是特殊情况 2^{2w-2} 需要 $2w$ 位（包括一个符号位 0）。然而，C 语言中的有符号乘法是通过将 $2w$ 位的乘积截断为 w 位的方式实现的。根据等式 (2-10)， w 位的补码乘法运算 $*_w^t$ 的结果为：

$$x *_w^t y = U2T_w((x \cdot y) \bmod 2^w) \quad (2-17)$$

我们认为对于无符号和补码乘法来说，乘法运算的位级表示都是一样的。也就是，给定长度为 w 的位向量 \vec{x} 和 \vec{y} ，无符号乘积 $B2U_w(\vec{x}) *_*^u_w B2U_w(\vec{y})$ 的位级表示与补码乘积 $B2T_w(\vec{x}) *_*^t_w B2T_w(\vec{y})$ 的位级表示是相同的。这表明机器可以用一种乘法指令来进行有符号和无符号整数的乘法。

举例说明，图 2-26 给出了不同 3 位数字的乘法结果。对于每一对位级运算数，我们执行无符号和补码乘法，得到 6 位的乘积，然后再把这些乘积截断到 3 位。无符号的截断后的乘积总是等于 $x \cdot y \bmod 8$ 。虽然无符号和补码两种乘法乘积的 6 位表示不同，但是截断后的乘积的位级表示都相同。

模式	x		y		$x \cdot y$		截断的 $x \cdot y$	
无符号	5	[101]	3	[011]	15	[001111]	7	[111]
补码	-3	[101]	3	[011]	-9	[110111]	-1	[111]
无符号	4	[100]	7	[111]	28	[011100]	4	[100]
补码	-4	[100]	-1	[111]	4	[000100]	-4	[100]
无符号	3	[011]	3	[011]	9	[001001]	1	[001]
补码	3	[011]	3	[011]	9	[001001]	1	[001]

图 2-26 3 位无符号和补码乘法示例（虽然完整的乘积位级表示相同，但是截断后的乘积的位级表示都相同）

为了说明（无符号和补码）乘积的低位是相同的，设 $x = B2T_w(\vec{x})$ 和 $y = B2T_w(\vec{y})$ 是这些位模式表示的补码值，而 $x' = B2U_w(\vec{x})$ 和 $y' = B2U_w(\vec{y})$ 是这些位模式表示的无符号值。根据等式 (2-5)，我们有 $x' = x + x_{w-1}2^w$ 和 $y' = y + y_{w-1}2^w$ 。计算这些值的乘积模 2^w 得到以下结果：

$$\begin{aligned}(x' \cdot y') \bmod 2^w &= [(x + x_{w-1}2^w) \cdot (y + y_{w-1}2^w)] \bmod 2^w \\&= [x \cdot y + (x_{w-1}y + y_{w-1}x)2^w + x_{w-1}y_{w-1}2^{2w}] \bmod 2^w \\&= (x \cdot y) \bmod 2^w\end{aligned}\quad (2-18)$$

由于模运算符，所有带有权重 2^w 的项都丢掉了，因此我们看到 $x \cdot y$ 和 $x' \cdot y'$ 的低 w 位是相同的。

练习题 2.34 按照图 2-26 的格式填写下表，说明不同的 3 位数字乘法的结果。

模式	x	y	$x \cdot y$	截断的 $x \cdot y$
无符号	[100]	[101]		
补码	[100]	[101]		
无符号	[010]	[111]		
补码	[010]	[111]		
无符号	[110]	[110]		
补码	[110]	[110]		

我们可以看出， w 位数字上的无符号运算和补码运算是同构的——运算 $+_*^u_w$ 、 $-_*^u_w$ 、 $*_*^u_w$ 和 $+_*^t_w$ 、 $-_*^t_w$ 、 $*_*^t_w$ 在位级上有相同的结果。

练习题 2.35 给你一个任务，开发函数 `tmult_ok` 的代码，该函数会判断两个参数相乘是否会产生溢出。下面是你的解决方案：



```
/* Determine whether arguments can be multiplied without overflow */
int tmult_ok(int x, int y) {
    int p = x*y;
    /* Either x is zero, or dividing p by x gives y */
    return !x || p/x == y;
}
```

你用 x 和 y 的很多值来测试这段代码，似乎都工作正常。你的同事向你挑战，说：“如果我不能用减法来检验加法是否溢出（参见练习题 2.31），那么你怎么能用除法来检验乘法是否溢出呢？”

按照下面的思路，用数学推导来证明你的方法是对的。首先，证明 $x=0$ 的情况是正确的。另外，考虑 w 位数字 x ($x \neq 0$)、 y 、 p 和 q ，这里 p 是 x 和 y 补码乘法的结果，而 q 是 p 除以 x 的结果。

1. 说明 x 和 y 的整数乘积 $x \cdot y$ ，可以写成这样的形式： $x \cdot y = p + t2^w$ ，其中 $t \neq 0$ 当且仅当 p 的计算溢出。
2. 说明 p 可以写成这样的形式： $p = x \cdot q + r$ ，其中 $|r| < |x|$ 。
3. 说明 $q = y$ 当且仅当 $r = t = 0$ 。

练习题 2.36 对于数据类型 `int` 为 32 位的情况，设计一个版本的 `tmult_ok` 函数（见练习题 2.35），要使用 64 位精度的数据类型 `long long`，而不使用除法。

XDR 库中的安全漏洞

2002 年，人们发现 Sun Microsystems 公司提供的实现 XDR 库的代码有安全漏洞，XDR 库是一个广泛使用的程序间共享数据结构的工具，造成这个安全漏洞的原因是程序会在毫无察觉的情况下产生乘法溢出。

包含安全漏洞的代码与下面所示类似：

```
1  /*
2   * Illustration of code vulnerability similar to that found in
3   * Sun's XDR library.
4   */
5  void* copy_elements(void *ele_src[], int ele_cnt, size_t ele_size) {
6      /*
7       * Allocate buffer for ele_cnt objects, each of ele_size bytes
8       * and copy from locations designated by ele_src
9       */
10     void *result = malloc(ele_cnt * ele_size);
11     if (result == NULL)
12         /* malloc failed */
13         return NULL;
14     void *next = result;
15     int i;
16     for (i = 0; i < ele_cnt; i++) {
17         /* Copy object i to destination */
18         memcpy(next, ele_src[i], ele_size);
19         /* Move pointer to next memory region */
20         next += ele_size;
21     }
22     return result;
23 }
```

函数 `copy_elements` 的设计可以将 `ele_cnt` 个数据结构复制到第 10 行的函数分配的缓冲区中，每个数据结构包含 `ele_size` 个字节。需要的字节数是通过计算 `ele_cnt * ele_size` 得到的。

想象一下，一个怀有恶意的程序员用参数 `ele_cnt` 等于 $1\ 048\ 577$ ($2^{20} + 1$)、`ele_size` 等于 $4\ 096$ (2^{12}) 来调用这个函数。然后第 10 行上的乘法会溢出，导致只会分配 4096 个字节，而不是装下这些数据所需要的 $4\ 294\ 971\ 392$ 个字节。从第 16 行开始的循环会试图复制所有的字节，超越已分配的缓冲区的界限，因而破坏了其他的数据结构。这会导致程序崩溃或者行为异常。



几乎每个操作系统都使用了这段 Sun 的代码，像 Internet Explorer 和 Kerberos 验证系统这样使用广泛的程序都用到了它。计算机紧急响应组（Computer Emergency Response Team, CERT），卡内基 - 梅隆软件工程协会（Carnegie Mellon Software Engineering Institute）运行的一个追踪安全漏洞或失效的组织，发布了建议“CA-2002-25”，于是许多公司急忙对它们的代码打补丁。幸运的是，还没有由于这个漏洞引起的安全失效的报告。

库函数 `calloc` 的实现中存在着类似的漏洞。这些已经被修补过了。

练习题 2.37 现在你的任务是修补上述 XDR 代码中的漏洞。你决定将待分配字节数设置为数据类型 `long long unsigned`, 来消除乘法溢出的可能性（至少在 32 位机器上）。你把原来对 `malloc` 函数的调用（第 10 行）替换如下：

```
long long unsigned asize =  
    ele_cnt * (long long unsigned) ele_size;  
void *result = malloc(asize);
```

- A. 这段代码在原始代码基础上有了哪些改进？
- B. 假设数据类型 `size_t` 和 `unsigned int` 是一样的，并且都是 32 位长，你该如何修改代码来消除这个漏洞？

2.3.6 乘以常数

在大多数机器上，整数乘法指令相当慢，需要 10 个或者更多的时钟周期，然而其他整数运算（例如加法、减法、位级运算和移位）只需要 1 个时钟周期。因此，编译器使用了一项重要的优化，试着用移位和加法运算的组合来代替乘以常数因子的乘法。首先，我们会考虑乘以 2 的幂的情况，然后再概括成乘以任意常数。

设 x 为位模式 $[x_{w-1}, x_{w-2}, \dots, x_0]$ 表示的无符号整数。那么，对于任何 $k > 0$ ，我们都认为 $[x_{w-1}, x_{w-2}, \dots, x_0, 0, \dots, 0]$ 给出了 $x2^k$ 的位级表示，这里右边增加了 k 个 0。这个属性可以通过等式（2-1）推导出来：

$$\begin{aligned} B2U_{w+k}([x_{w-1}, x_{w-2}, \dots, x_0, 0, \dots, 0]) &= \sum_{i=0}^{w-1} x_i 2^{i+k} \\ &= \left[\sum_{i=0}^{w-1} x_i 2^i \right] \cdot 2^k \\ &= x 2^k \end{aligned}$$

对于 $k < w$ ，我们可以将移位后的位向量截断到长度 w ，得到 $[x_{w-k-1}, x_{w-k-2}, \dots, x_0, 0, \dots, 0]$ 。根据等式（2-9），这个位向量的数值为 $x2^k \bmod 2^w = x * 2^k$ 。因此，对于无符号变量 x ，C 表达式 $x \ll k$ 等价于 $x * \text{pwr2k}$ ，这里 pwr2k 等于 2^k 。特别地，我们可以用 $1U \ll k$ 来计算 pwr2k 。

通过类似的推理，我们可以得出，对于一个位模式为 $[x_{w-1}, x_{w-2}, \dots, x_0]$ 的补码数 x ，以及范围在 $0 \leq k < w$ 内任意的 k ，位模式 $[x_{w-k-1}, x_{w-k-2}, \dots, x_0, 0, \dots, 0]$ 就是 $x * 2^k$ 的补码表示。因此，对于有符号变量 x ，C 表达式 $x \ll k$ 等价于 $x * \text{pwr2k}$ ，这里 pwr2k 等于 2^k 。

注意，无论是无符号运算还是补码运算，乘以 2 的幂都可能会导致溢出。结果表明，即使溢出的时候，我们通过移位得到的结果也是一样的。

由于整数乘法比移位和加法的代价要大得多，许多 C 语言编译器试图以移位、加法和减法的组合来消除很多整数乘以常数的情况。例如，假设一个程序包含表达式 $x * 14$ 。利用等式 $14 = 2^3 + 2^2 + 2^1$ ，编译器会将乘法重写为 $(x \ll 3) + (x \ll 2) + (x \ll 1)$ ，实现了将一个乘法替换为三个移位和两个加法。无论 x 是无符号的还是补码，甚至当乘法会导致溢出时，两个计算都会得到



一样的结果。(根据整数运算的属性可以证明这一点。)更好的方法是,编译器还可以利用属性 $14 = 2^4 - 2^1$,将乘法重写为 $(x << 4) - (x << 1)$,这时只需要两个移位和一个减法。

练习题 2.38 就像我们将在第 3 章中看到的那样,LEA 指令能够执行形如 $(a << k) + b$ 的计算,这里 k 等于 0、1、2 或 3,而 b 等于 0 或者某个程序值。编译器常常用这条指令来执行常数因子乘法。例如,我们可以用 $(a << 1) + a$ 来计算 $3 * a$ 。

考虑 b 等于 0 或者等于 a 、 k 为任意可能的值的情况,用一条 LEA 指令可以计算 a 的哪些倍数?

归纳一下我们的例子,考虑一个任务,对于某个常数 K 的表达式 $x * K$ 生成代码。编译器会将 K 的二进制表示表达为一组 0 和 1 交替的序列:

$$[(0 \cdots 0)(1 \cdots 1)(0 \cdots 0) \cdots (1 \cdots 1)].$$

例如,14 可以写成 $[(0 \cdots 0)(111)(0)]$ 。考虑一组从位位置 n 到位位置 m 的连续的 1 ($n < m$)。(对于 14 来说,我们有 $n = 3$ 和 $m = 1$ 。)我们可以用下面两种不同形式中的一种来计算这些位对乘积的影响:

形式 A: $(x << n) + (x << n-1) + \cdots + (x << m)$

形式 B: $(x << n+1) - (x << m)$

把每个这样连续的 1 的结果加起来,不用做乘法,我们就能计算出 $x * K$ 。当然,选择使用移位、加法和减法的组合,还是使用一条乘法指令,取决于这些指令的相对速度,而这些是与机器高度相关的。大多数编译器只在需要少量移位、加法和减法就足够的时候才使用这种优化。

练习题 2.39 对于位位置 n 为最高有效位的情况,我们要怎样修改形式 B 的表达式?

练习题 2.40 对于下面每个 K 的值,找出只用指定数量的运算表达 $x * K$ 的方法,这里我们认为加法和减法的开销相当。除了我们已经考虑过的简单的形式 A 和 B 原则,你可能会需要使用一些技巧。

K	移位	加法 / 减法	表达式
6	2	1	
31	1	1	
-6	2	1	
55	2	2	

练习题 2.41 对于一组从位位置 n 开始到位位置 m 结束的连续的 1 ($n < m$),我们看到可以产生两种形式的代码,A 和 B。编译器该如何决定使用哪一种呢?

2.3.7 除以 2 的幂

在大多数机器上,整数除法要比整数乘法更慢——需要 30 个或者更多的时钟周期。除以 2 的幂也可以用移位运算来实现,只不过我们用的是右移,而不是左移。无符号和补码数分别使用逻辑移位和算术移位来达到目的。

整数除法总是舍入到零。对于 $x > 0$ 和 $y > 0$,结果是 x/y ,这里对于任何实数 a , a 定义为唯一的整数 d ,使得 $d \leq a < d+1$ 。例如, $3.14_1 = 3$, $-3.14_1 = -4$,而 $3_1 = 3$ 。

考虑对一个无符号数执行逻辑右移 k 位的效果。我们认为这和除以 2^k 有一样的效果。例如,图 2-27 给出了在 12340 的 16 位表示上执行逻辑右移的结果,以及对它执行除以 1、2、16 和 256 的结果。从左端移入的 0 以斜体表示。我们还给出了如果用真正的运算去做除法得到的结果。这些示例说明,移位总是舍入到零,这一结果与整数除法的规则一样。

为了证明逻辑右移和除以 2 的幂之间的关系,设 x 为位模式 $[x_{w-1}, x_{w-2}, \dots, x_0]$ 表示的无符号整数,而 k 的取值范围为 $0 \leq k < w$ 。设 x' 为 $w-k$ 位的位表示 $[x_{w-1}, x_{w-2}, \dots, x_k]$ 的无符号数,而 x'' 为 k 位的位表示 $[x_{k-1}, \dots, x_0]$ 的无符号数。我们有 $x' = x/2^k$ 。证明如下:



根据等式(2-1), 我们有 $x = \sum_{i=0}^{w-1} x_i 2^i$, $x' = \sum_{i=k}^{w-1} x_i 2^{i-k}$ 和 $x'' = \sum_{i=0}^{k-1} x_i 2^i$ 。因此, 我们可以把 x 写为 $x = 2^k x' + x''$ 。可以观察到 $0 \leq x'' \leq \sum_{i=0}^{k-1} 2^i = 2^k - 1$, 因此 $0 \leq x' < 2^k$, 这意味着 $x''/2^k = 0$ 。因此, $\lfloor x/2^k \rfloor = \lfloor x'+x''/2^k \rfloor = x'+\lfloor x''/2^k \rfloor = x'$ 。

对位向量 $[x_{w-1}, x_{w-2}, \dots, x_0]$ 逻辑右移 k 位会得到位向量

$$[0, \dots, 0, x_{w-1}, x_{w-2}, \dots, x_k]$$

这个位向量有数值 x' 。因此, 对于无符号变量 x , C 表达式 $x>>k$ 等价于 $x/\text{pwr2k}$, 这里 pwr2k 等价于 2^k 。

k	>> k (二进制)	十进制	$12340/2^k$
0	0011000000110100	12340	12340.0
1	0001100000011010	6170	6170.0
4	0000001100000011	771	771.25
8	0000000001100000	48	48.203125

图 2-27 无符号数除以 2 的幂

现在考虑对一个补码数进行算术右移的结果。对于一个正整数, 最高有效位为 0, 所以效果与逻辑右移是一样的。因此, 对于非负数来说, 算术右移 k 位与除以 2^k 是一样的。作为一个负数的例子, 图 2-28 给出了对 -12340 的 16 位表示进行算术右移不同位数的结果。正如我们能看到的, 结果与除以 2 的幂几乎完全一样。对于不需要舍入的情况 ($k=1$), 结果是正确的。但是当需要进行舍入时, 移位导致结果向下舍入, 而不是像规则需要的那样向零舍入。例如, 表达式 $-7/2$ 应该得到 -3 , 而不是 -4 。

k	>> k (二进制)	十进制	$-12340/2^k$
0	1100111111001100	-12340	-12340.0
1	/110011111100110	-6170	-6170.0
4	///110011111100	-772	-771.25
8	////////11001111	-49	-48.203125

图 2-28 进行算术右移

让我们更好地理解算术右移的效果, 以及如果利用它来执行除以 2 的幂。设 x 为位模式 $[x_{w-1}, x_{w-2}, \dots, x_0]$ 表示的补码整数, 而 k 的取值范围为 $0 \leq k < w$ 。设 x' 为 $w-k$ 位 $[x_{w-1}, x_{w-2}, \dots, x_k]$ 表示的补码数, 而 x'' 为低 k 位 $[x_{k-1}, \dots, x_0]$ 表示的无符号数。与分析无符号情况类似, 我们有 $x = 2^k x' + x''$, 而 $0 \leq x'' < 2^k$, 得到 $x' = \lfloor x/2^k \rfloor$ 。进一步, 可以观察到, 算术右移位向量 $[x_{w-1}, x_{w-2}, \dots, x_0]$ k 位, 得到位向量

$$[x_{w-1}, \dots, x_{w-1}, x_{w-1}, x_{w-2}, \dots, x_k]$$

它刚好就是将 $[x_{w-1}, x_{w-2}, \dots, x_k]$ 从 $w-k$ 位符号扩展到 w 位。因此, 这个移位后的位向量就是 $x/2^k$ 的补码表示。这个分析证实了我们从图 2-28 的示例中发现的结论。

对于 $x < 0$, 或是不需要舍入的时候 ($x'' = 0$), 我们的分析表明这个移位的结果就是所期望的值。不过, 对于 $x < 0$ 和 $y > 0$, 整数除法的结果应该是 $\lceil x/y \rceil$, 这里, 对于任何实数 a , $\lceil a \rceil$ 被定义为使得 $a'-1 < a \leq a'$ 的唯一整数 a' 。也就是说, 整数除法应该将为负的结果向上朝零舍入。因此, 当有舍入发生时, 将一个负数右移 k 位不等价于把它除以 2^k 。这个分析也证实了我们从图 2-28 的示例中发现的结论。

我们可以在移位之前“偏置”(biasing)这个值, 通过这种方法修正这种不合适的舍入。这种技术利用的属性是: 对于整数 x 和任意 $y > 0$ 的 y , 有 $\lceil x/y \rceil = \lfloor (x+y-1)/y \rfloor$ 。例如,



当 $x = -30$ 且 $y = 4$, 我们有 $x + y - 1 = -27$, 而 $\lceil -30/4 \rceil = -7 = \lfloor 27/4 \rfloor$ 。当 $x = -32$ 且 $y = 4$ 时, 我们有 $x + y - 1 = -29$, 而 $\lceil -32/4 \rceil = -8 = \lfloor 29/4 \rfloor$ 。通用的方式表达这个关系, 假设 $x = ky + r$, 这里 $0 \leq r < y$, 得到 $(x + y - 1)/y = k + (r + y - 1)/y$, 因此 $(x + y - 1)/y = k + (r + y - 1)/y$ 。当 $r = 0$ 时, 后面一项等于 0, 而当 $r > 0$ 时, 等于 1。也就是说, 通过给 x 增加一个偏量 $y - 1$, 然后再将除法向下舍入, 当 y 整除 x 时, 我们得到 k , 否则, 就得到 $k + 1$ 。因此, 对于 $x < 0$, 如果在右移之前, 先将 x 加上 $2^k - 1$, 那么我们就会得到正确舍入的结果了。

这个分析表明对于使用算术右移的补码机器, C 表达式

```
(x<0 ? (x + (1<<k) - 1) : x) >> k
```

等价于 $x/pwr2k$, 这里 $pwr2k$ 等于 2^k 。

图 2-29 说明在执行算术右移之前加上一个适当的偏量是如何导致结果正确舍入的。在第 3 列, 我们给出了 -12340 加上偏量值之后的结果, 低 k 位 (那些会向右移出的位) 以斜体表示。我们可以看到, 低 k 位左边的位可能会加 1, 也可能不会加 1。对于不需要舍入的情况 ($k = 1$), 加上偏量只影响那些被移掉的位。对于需要舍入的情况, 加上偏量导致较高的位加 1, 所以结果会向零舍入。

k	偏量	-12340 + 偏量	>> k (二进制)	十进制	-12340/2 ^k
0	0	1100111111001100	1100111111001100	-12340	-12340.0
1	1	1100111111001101	<i>1</i> 10011111100110	-6170	-6170.0
4	15	1100111111011011	<i>111110011111101</i>	-771	-771.25
8	255	1101000011001011	<i>1111111110100000</i>	-48	-48.203125

图 2-29 补码除以 2 的幂 (右移之前加上一个偏量, 结果向零舍入)

练习题 2.42 写一个函数 `div16`, 对于整数参数 x 返回 $x/16$ 的值。你的函数不能使用除法、模运算、乘法、任何条件语句 (`if` 或者 `:=`)、任何比较运算符 (例如 `<`、`>` 或 `==`) 或任何循环。你可以假设数据类型 `int` 是 32 位长, 使用补码表示, 而右移是算术右移。

现在我们看到, 除以 2 的幂可以通过逻辑或者算术右移来实现。这也正是为什么大多数机器上提供这两种类型的右移。不幸的是, 这种方法不能推广到除以任意常数。同乘法不同, 我们不能用除以 2 的幂的除法来表示除以任意常数 K 的除法。

练习题 2.43 在下面的代码中, 我们省略了常数 M 和 N 的定义:

```
#define M /* Mystery number 1 */
#define N /* Mystery number 2 */
int arith(int x, int y) {
    int result = 0;
    result = x*M + y/N; /* M and N are mystery numbers. */
    return result;
}
```

我们以某个 M 和 N 的值编译这段代码。编译器用我们讨论过的方法优化乘法和除法。下面是将产生的机器代码翻译回 C 语言的结果:

```
/* Translation of assembly code for arith */
int optarith(int x, int y) {
    int t = x;
    x <= 5;
    x -= t;
    if (y < 0) y += 7;
    y >>= 3; /* Arithmetic shift */
    return x+y;
}
```

M 和 N 的值为多少?



2.3.8 关于整数运算的最后思考

正如我们看到的，计算机执行的“整数”运算实际上是一种模运算形式。表示数字的有限字长限制了可能的值的取值范围，结果运算可能溢出。我们还看到，补码表示提供了一种既能表示负数也能表示正数的灵活方法，同时使用了与执行无符号算术相同的位级实现，这些运算包括加法、减法、乘法，甚至除法，无论运算数是以无符号形式还是以补码形式表示的，都有完全一样或者非常类似的位级行为。

我们看到了C语言中的某些规定可能会产生令人意想不到的结果，而这些可能是难以察觉和理解的缺陷的源头。我们特别看到了`unsigned`数据类型，虽然它概念上很简单，但可能导致即使是资深程序员都意想不到的行为。我们还看到这种数据类型会以出乎意料的方式出现，比如，当书写整数常数和当调用库函数时。

练习题 2.44 假设我们在对有符号值使用补码运算的32位机器上运行代码。对于有符号值使用的是算术右移，而对于无符号值使用的是逻辑右移。变量的声明和初始化如下：

```
int x = foo(); /* Arbitrary value */  
int y = bar(); /* Arbitrary value */  
  
unsigned ux = x;  
unsigned uy = y;
```

对于下面每个C表达式，1) 证明对于所有的x和y值，它都为真（等于1）；或者2) 给出使得它为假（等于0）的x和y的值：

- A. $(x > 0) \mid\mid (x - 1 < 0)$
- B. $(x \& 7) != 7 \mid\mid (x << 29 < 0)$
- C. $(x * x) \geq 0$
- D. $x < 0 \mid\mid -x \leq 0$
- E. $x > 0 \mid\mid -x \geq 0$
- F. $x + y == uy + ux$
- G. $x * \sim y + uy * \sim ux == -x$

2.4 浮点数

浮点表示对形如 $V = x \times 2^y$ 的有理数进行编码。它对执行涉及非常大的数字($|V| >> 0$)、非常接近于0($|V| < 1$)的数字，以及更普遍地作为实数运算的近似值的计算，是很有用的。

直到20世纪80年代，每个计算机制造商都设计了自己的表示浮点数的规则，以及对浮点数执行运算的细节。另外，它们常常不会太多地关注运算的精确性，而把实现的速度和简便性看得比数字精确性更重要。

大约在1985年，这些情况随着IEEE标准754的推出而改变了，这是一个仔细制订的表示浮点数及其运算的标准。这项工作是从1976年开始由Intel赞助的，在8087设计的同时，8087是一种为8086处理器提供浮点支持的芯片。他们请William Kahan（加州大学伯克利分校的一位教授）作为顾问，帮助设计未来处理器浮点标准。他们支持Kahan加入一个IEEE资助的制订工业标准的委员会。这个委员会最终采纳的标准非常接近于Kahan为Intel设计的标准。目前，实际上所有的计算机都支持这个后来被称为IEEE浮点的标准。这大大提高了科学应用程序在不同机器上的可移植性。



IEEE

电气和电子工程师协会（IEEE——读做“Eye-Triple-Eee”）是一个包括所有电子和计算机技术的专业团体。它出版刊物，举办会议，并且建立委员会来定义标准，内容涉及范围从电力传输到软件工程。

在本节，我们将看到在 IEEE 浮点格式中是如何表示数字的。我们还将探讨舍入（rounding）的问题，当一个数字不能被准确地表示为这种格式时，就必须向上调整或者向下调整，此时就会出现舍入。然后，我们将探讨加法、乘法和关系运算符的数学属性。许多程序员认为浮点数没意思，往坏了说，深奥难懂。我们将看到，因为 IEEE 格式是定义在一组小而一致的原则上的，所以它实际上是相当优雅和容易理解的。

2.4.1 二进制小数

理解浮点数的第一步是考虑含有小数值的二进制数字。首先，让我们来看看更熟悉的十进制表示法。十进制表示法使用的表示形式为： $d_m d_{m-1} \cdots d_1 d_0 . d_{-1} d_{-2} \cdots d_{-n}$ ，其中每个十进制数 d_i 的取值范围是 0 ~ 9。这个表示方法描述的数值 d 定义如下：

$$d = \sum_{i=-n}^m 10^i \times d_i$$

数字权的定义与十进制小数点符号（‘.’）相关，这意味着小数点左边的数字的权是 10 的正幂，得到整数值，而小数点右边的数字的权是 10 的负幂，得到小数值。例如， 12.34_{10} 表示数字 $1 \times 10^1 + 2 \times 10^0 + 3 \times 10^{-1} + 4 \times 10^{-2} = 12\frac{34}{100}$ 。

类似地，考虑一个形如 $b_m b_{m-1} \cdots b_1 b_0 . b_{-1} b_{-2} \cdots b_{-n-1} b_{-n}$ 的表示法，其中每个二进制数字，或者称为位， b_i 的取值范围是 0 和 1，如图 2-30 所示。这种表示方法表示的数 b 定义如下：

$$b = \sum_{i=-n}^m 2^i \times b_i; \quad (2-19)$$

符号 ‘.’ 现在变为了二进制的点，点左边的位的权是 2 的正幂，点右边的位的权是 2 的负幂。例如， 101.11_2 表示数字 $1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} = 4 + 0 + 1 + \frac{1}{2} + \frac{1}{4} = 5\frac{3}{4}$ 。

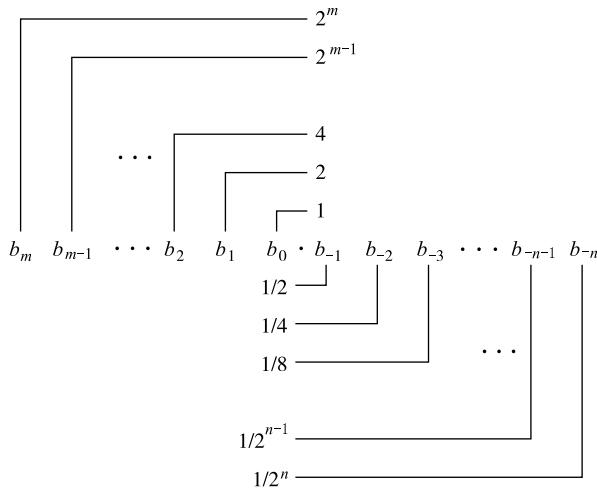


图 2-30 小数的二进制表示。二进制点左边的数字的权形如 2^i ，而右边的数字的权形如 $1/2^i$

从等式 (2-19) 中可以很容易地看出，二进制小数点向左移动一位相当于这个数被 2 除。例如， 101.11_2 表示数 $5\frac{3}{4}$ ，而 10.111_2 表示数 $2 + 0 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} = 2\frac{7}{8}$ 。类似地，二进制小数点向右移动一位相当于将该数乘 2。例如 1011.1_2 表示数 $8 + 0 + 2 + 1 + \frac{1}{2} = 11\frac{1}{2}$ 。



注意，形如 $0.11\dots 1_2$ 的数表示刚好小于 1 的数。例如， 0.111111_2 表示 $\frac{63}{64}$ ，我们将用简单的表达法 1.0- 来表示这样的数值。

假定我们仅考虑有限长度的编码，那么十进制表示法不能准确地表达像 $\frac{1}{3}$ 和 $\frac{5}{7}$ 这样的数。类似地，小数的二进制表示法只能表示那些能够被写成 $x \times 2^y$ 的数。其他的值只能被近似地表示。例如，数字 $\frac{1}{5}$ 可以用十进制小数 0.20 精确表示。不过，我们并不能把它准确地表示为一个二进制小数，我们只能近似地表示它，增加二进制表示的长度可以提高表示的精度：

表示	值	十进制
0.0_2	$\frac{0}{2}$	0.0_{10}
0.01_2	$\frac{1}{4}$	0.25_{10}
0.010_2	$\frac{2}{8}$	0.25_{10}
0.0011_2	$\frac{3}{16}$	0.1875_{10}
0.00110_2	$\frac{6}{32}$	0.1875_{10}
0.001101_2	$\frac{13}{64}$	0.203125_{10}
0.0011010_2	$\frac{26}{128}$	0.203125_{10}
0.00110011_2	$\frac{51}{256}$	0.19921875_{10}

练习题 2.45 填写下表中的缺失的信息：

小数值	二进制表示	十进制表示
$\frac{1}{8}$	0.001	0.125
$\frac{3}{4}$		
$\frac{25}{16}$		
	10.1011	
	1.001	
		5.875
		3.1875

练习题 2.46 浮点运算的不精确性能够产生灾难性的后果。

爱国者导弹系统中有一个内置的时钟，其实现类似一个计数器，每 0.1 秒就加 1。为了以秒为单位来确定时间，程序将用一个 24 位的近似于 $1/10$ 的二进制小数值来乘以这个计数器的值。特别地， $1/10$ 的二进制表达式是一个无穷序列 $0.000110011[0011]\dots_2$ ，其中，方括号里的部分是无限循环的。程序用值 x 近似地表示 0.1， x 只考虑这个序列的二进制小数点右边的前 23 位： $x = 0.00011001100110011001100$ 。（参考练习题 2.51，里面有关于如何更精确地近似表示 0.1 的讨论。）

- A. $0.1 - x$ 的二进制表示是什么？
- B. $0.1 - x$ 的近似的十进制值是多少？
- C. 当系统初始启动时，时钟从 0 开始，并且一直保持计数。在这个例子中，系统已经运行了大约 100 个小时。程序计算出的时间和实际的时间之差为多少？
- D. 系统根据一枚来袭导弹的速率和它最后被雷达侦测到的时间，来预测它将在哪里出现。假定飞毛腿导弹的速率大约是 2000 米每秒，对它的预测偏差了多少？

通过一次读取时钟得到的绝对时间中的一个轻微错误，通常不会影响跟踪的计算。相反，它应该依赖于两次连续的读取之间的相对时间。问题是爱国者导弹的软件已经升级，可以使用更精确的函数来读取时间，但不是所有的函数调用都用新的代码替换。结果就是，跟踪软件一次读取用的是精确的时间，而另一次读取用的是不精确的时间 [100]。



2.4.2 IEEE 浮点表示

前一节中谈到的定点表示法不能很有效地表示非常大的数字。例如，表达式 5×2^{100} 是用 101 后面跟随 100 个零组成的位模式来表示。相反地，我们希望通过给定 x 和 y 的值，来表示形如 $x \times 2^y$ 的数。

IEEE 浮点标准用 $V = (-1)^s \times M \times 2^E$ 的形式来表示一个数：

- 符号 (sign) s 决定这个数是负数 ($s=1$) 还是正数 ($s=0$)，而对于数值 0 的符号位解释作为特殊情况处理。
- 尾数 (significand) M 是一个二进制小数，它的范围是 $1 \sim 2 - \varepsilon$ ，或者是 $0 \sim 1 - \varepsilon$ 。
- 阶码 (exponent) E 的作用是对浮点数加权，这个权重是 2 的 E 次幂（可能是负数）。

将浮点数的位表示划分为三个字段，分别对这些值进行编码：

- 一个单独的符号位 s 直接编码符号 s 。
- k 位的阶码字段 $\text{exp} = e_{k-1}\cdots e_1 e_0$ 编码阶码 E 。
- n 位小数字段 $\text{frac} = f_{n-1}\cdots f_1 f_0$ 编码尾数 M ，但是编码出来的值也依赖于阶码字段的值是否等于 0。

图 2-31 给出了将这三个字段装进字中两种最常见的格式。在单精度浮点格式（C 语言中的 float）中， s 、 exp 和 frac 字段分别为 1 位、 $k = 8$ 位和 $n = 23$ 位，得到一个 32 位的表示。在双精度浮点格式（C 语言中的 double）中， s 、 exp 和 frac 字段分别为 1 位、 $k = 11$ 位和 $n = 52$ 位，得到一个 64 位的表示。

单精度



双精度

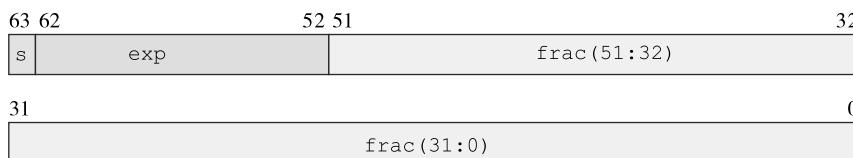


图 2-31 标准浮点格式（浮点由 3 个字段表示，两种最常见的格式是它们被封装到 32 位（单精度）和 64 位（双精度）的字中）

给定了位表示，根据 exp 的值，被编码的值可以分成三种不同的情况（最后一种情况有两个变种）。图 2-32 说明了对单精度格式的情况。

1. 规格化的



2. 非规格化的



3a. 无穷大



3b. NaN



图 2-32 单精度浮点数值的分类（阶码的值决定了这个数是规格化的、非规格化的、或特殊值）

情况 1：规格化的值

这是最普遍的情况。当 exp 的位模式既不全为 0（数值 0），也不全为 1（单精度数值为



255，双精度数值为 2047) 时，都属于这类情况。在这种情况下，阶码字段被解释为以偏置 (biased) 形式表示的有符号整数。也就是说，阶码的值是 $E = e - Bias$ ，其中 e 是无符号数，其位表示为 $e_{k-1} \cdots e_1 e_0$ ，而 $Bias$ 是一个等于 $2^{k-1} - 1$ (单精度是 127，双精度是 1023) 的偏置值。由此产生指数的取值范围，对于单精度是 $-126 \sim +127$ ，而对于双精度是 $-1022 \sim +1023$ 。

对小数字段 frac 的解释为描述小数值 f ，其中 $0 \leq f < 1$ ，其二进制表示为 $0.f_{n-1} \cdots f_1 f_0$ ，也就是二进制小数点在最高有效位的左边。尾数定义为 $M = 1 + f$ 。有时，这种方式也叫做隐含的以 1 开头的 (implied leading 1) 表示，因为我们可以把 M 看成一个二进制表达式为 $1.f_{n-1} f_{n-2} \cdots f_0$ 的数字。既然我们总是能够调整阶码 E ，使得尾数 M 在范围 $1 \leq M < 2$ 之中 (假设没有溢出)，那么这种表示方法是一种轻松获得一个额外精度位的技巧。由于第一位总是等于 1，因此我们就不需要显式地表示它。

情况 2：非规格化的值

当阶码域为全 0 时，所表示的数就是非规格化形式。在这种情况下，阶码值是 $E = 1 - Bias$ ，而尾数的值是 $M = f$ ，也就是小数字段的值，不包含隐含的开头的 1。

对于非规格化值为什么要这样设置偏置值

使阶码值为 $1 - Bias$ 而不是简单的 $-Bias$ 似乎是违反直觉的。我们将很快看到，这种方式提供了一种从非规格化值平滑转换到规格化值的方法。

非规格化数有两个用途。首先，它们提供了一种表示数值 0 的方法，因为使用规格化数，我们必须总是使 $M = 1$ ，因此我们就不能表示 0。实际上， $+0.0$ 的浮点表示的位模式为全 0：符号位是 0，阶码字段全为 0 (表明是一个非规格化值)，而小数域也全为 0，这就得到 $M = f = 0$ 。令人奇怪的是，当符号位为 1，而其他域全为 0 时，我们得到值 -0.0 。根据 IEEE 的浮点格式，认为值 $+0.0$ 和 -0.0 在某些方面是不同的，而在其他方面是相同的。

非规格化数的另外一个功能是表示那些非常接近于 0.0 的数。它们提供了一种属性，称为逐渐溢出 (gradual underflow)，其中，可能的数值分布均匀地接近于 0.0。

情况 3：特殊值

最后一类数值是当指阶码全为 1 的时候出现的。当小数域全为 0 时，得到的值表示无穷，当 $s = 0$ 时是 $+\infty$ ，或者当 $s = 1$ 时是 $-\infty$ 。当我们把两个非常大的数相乘，或者除以零时，无穷能够表示溢出的结果。当小数域为非零时，结果值被称为 “*Nan*”，就是“不是一个数” (Not a Number) 的缩写。一些运算的结果不能是实数或无穷，就会返回这样的 *Nan* 值，比如当计算 $\sqrt{-1}$ 或 $-\infty - \infty$ 时。在某些应用中，表示未初始化的数据时，它们也很有用处。

2.4.3 数字示例

图 2-33 展示了一组数值，它们可以用假定的 6 位格式来表示，有 $k = 3$ 的阶码位和 $n = 2$ 的尾数位。偏置量是 $2^{3-1} - 1 = 3$ 。图中的 A 部分显示了所有可表示的值 (除了 *Nan*)。两个无穷值

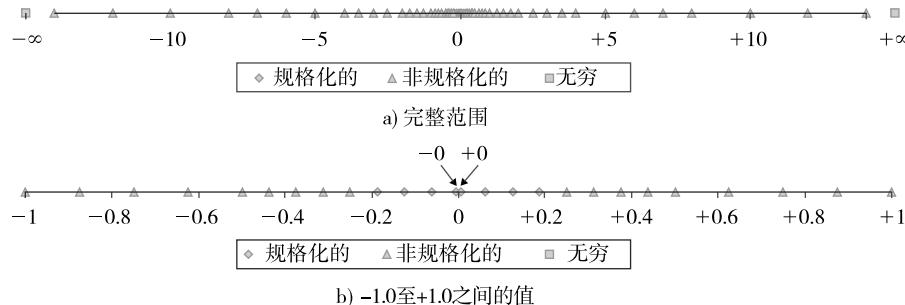


图 2-33 6 位浮点格式可表示的值 ($k = 3$ 的阶码位, $n = 2$ 的尾数位, 偏置量为 3)



在两个末端。最大数量值的规格化数是 ± 14 。非规格化数聚集在 0 的附近。图的 B 部分中，我们只展示了介于 -1.0 和 +1.0 之间的数值，这样就能够看得更加清楚了。两个零是特殊的非规格化数。可以观察到，那些可表示的数并不是均匀分布的——越靠近原点处它们越稠密。

图 2-34 展示了假定的 8 位浮点格式的示例，其中有 $k = 4$ 的阶码位和 $n = 3$ 的小数位。偏置量是 $2^{4-1} - 1 = 7$ 。图被分成了三个区域，用来描述三类数字。不同的列给出了阶码字段是如何编码阶码 E 的，小数字段是如何编码尾数 M 的，以及它们一起是如何形成要表示的值 $V = 2^E \times M$ 的。从 0 自身开始，最靠近 0 的是非规格化数。这种格式的非规格化数的 $E = 1 - 7 = -6$ ，得到权 $2^E = \frac{1}{64}$ 。小数 f 的值的范围是 $0, \frac{1}{8}, \dots, \frac{7}{8}$ ，从而得到数 V 的范围是 $0 \sim \frac{1}{64} \times \frac{7}{8} = \frac{7}{512}$ 。

描述	位表示	指数			小数		值		
		e	E	2^E	f	M	$2^E \times M$	V	十进制
0 最小的非规格化数	0 0000 000	0	-6	$\frac{1}{64}$	$\frac{0}{8}$	$\frac{0}{8}$	$\frac{0}{512}$	0	0.0
	0 0000 001	0	-6	$\frac{1}{64}$	$\frac{1}{8}$	$\frac{1}{8}$	$\frac{1}{512}$	$\frac{1}{512}$	0.001953
	0 0000 010	0	-6	$\frac{1}{64}$	$\frac{2}{8}$	$\frac{2}{8}$	$\frac{2}{512}$	$\frac{1}{256}$	0.003906
	0 0000 011	0	-6	$\frac{1}{64}$	$\frac{3}{8}$	$\frac{3}{8}$	$\frac{3}{512}$	$\frac{3}{512}$	0.005859
	⋮								
最大的非规格化数	0 0000 111	0	-6	$\frac{1}{64}$	$\frac{7}{8}$	$\frac{7}{8}$	$\frac{7}{512}$	$\frac{7}{512}$	0.013672
1 最小的规格化数	0 0001 000	1	-6	$\frac{1}{64}$	$\frac{0}{8}$	$\frac{8}{8}$	$\frac{8}{512}$	$\frac{1}{64}$	0.015625
	0 0001 001	1	-6	$\frac{1}{64}$	$\frac{1}{8}$	$\frac{9}{8}$	$\frac{9}{512}$	$\frac{9}{512}$	0.017578
	⋮								
	0 0110 110	6	-1	$\frac{1}{2}$	$\frac{6}{8}$	$\frac{14}{8}$	$\frac{14}{16}$	$\frac{7}{8}$	0.875
	0 0110 111	6	-1	$\frac{1}{2}$	$\frac{7}{8}$	$\frac{15}{8}$	$\frac{15}{16}$	$\frac{15}{16}$	0.9375
	0 0111 000	7	0	1	$\frac{0}{8}$	$\frac{8}{8}$	$\frac{8}{8}$	1	1.0
	0 0111 001	7	0	1	$\frac{1}{8}$	$\frac{9}{8}$	$\frac{9}{8}$	$\frac{9}{8}$	1.125
	0 0111 010	7	0	1	$\frac{2}{8}$	$\frac{10}{8}$	$\frac{10}{8}$	$\frac{5}{4}$	1.25
	⋮								
	0 1110 110	14	7	128	$\frac{6}{8}$	$\frac{14}{8}$	$\frac{1792}{8}$	224	224.0
最大的规格化数	0 1110 111	14	7	128	$\frac{7}{8}$	$\frac{15}{8}$	$\frac{1920}{8}$	240	240.0
无穷大	0 1111 000	—	—	—	—	—	—	∞	—

图 2-34 8 位浮点格式的非负值示例

这种形式的最小规格化数同样有 $E = 1 - 7 = -6$ ，并且小数取值范围也为 $0, \frac{1}{8}, \dots, \frac{7}{8}$ 。然而，尾数的范围在 $1 + 0 = 1$ 和 $1 + \frac{7}{8} = \frac{15}{8}$ 之间，得出数 V 的范围在 $\frac{8}{512} = \frac{1}{64}$ 和 $\frac{15}{512}$ 之间。

可以观察到最大非规格化数 $\frac{7}{512}$ 和最小规格化数 $\frac{8}{512}$ 之间的平滑转变。这种平滑性归功于我们对非规格化数的 E 的定义。通过将 E 定义为 $1 - Bias$ ，而不是 $-Bias$ ，我们可以补偿非规格化数的尾数没有隐含的开头的 1 这一事实。

当增大阶码时，我们成功地得到更大的规格化值，通过 1.0 后得到最大的规格化数。这个数具有阶码 $E = 7$ ，得到一个权 $2^E = 128$ 。小数等于 $\frac{7}{8}$ 得到尾数 $M = \frac{15}{8}$ 。因此，数值是 $V = 240$ 。超出这个值就会溢出到 $+\infty$ 。

这种表示具有一个有趣的属性，假如我们将图 2-34 中的值的位表达式解释为无符号整数，它们就是按升序排列的，就像它们表示的浮点数一样。这不是偶然的——IEEE 如此设计格式就



是为了浮点数能够使用整数排序函数来进行排序。当处理负数时，有一个小的难点，因为它们有开头的1，并且它们是按照降序出现的，但是不需要浮点运算来进行比较也能解决这个问题（参见家庭作业2.83）。

练习题2.47 假设一个基于IEEE浮点格式的5位浮点表示，有1个符号位、2个阶码位($k=2$)和两个小数位($n=2$)。阶码偏置量是 $2^{2-1}-1=1$ 。

下表中列举了这个5位浮点表示的全部非负取值范围。使用下面的条件，填写表格中的空白项：

e ：假定阶码字段是一个无符号整数表示的值。

E ：偏置之后的阶码值。

2^E ：阶码的权重数。

f ：小数值。

M ：尾数的值。

$2^E \times M$ ：该数(未归约的)小数值。

V ：该数归约后的小数值。

十进制：该数的十进制表示。

写出 2^E 、 f 、 M 、 $2^E \times M$ 和 V 的值，要么是整数(如果可能的话)，要么是形如 $\frac{x}{y}$ 的小数，这里 y 是2的幂。标注“-”的条目不用填。

位	e	E	2^E	f	M	$2^E \times M$	V	十进制
0 00 00								
0 00 01								
0 00 10								
0 00 11								
0 01 00								
0 01 01	1	0	1	$\frac{1}{4}$	$\frac{5}{4}$	$\frac{5}{4}$	$\frac{5}{4}$	1.25
0 01 10								
0 01 11								
0 10 00								
0 10 01								
0 10 10								
0 10 11								
0 11 00	—	—	—	—	—	—	—	—
0 11 01	—	—	—	—	—	—	—	—
0 11 10	—	—	—	—	—	—	—	—
0 11 11	—	—	—	—	—	—	—	—

图2-35展示了重要的单精度和双精度浮点数的表示和数字值。根据图2-34中展示的8位格式，我们能够看出有 k 位阶码和 n 位小数的浮点表示的一般属性。

- 值+0.0总有一个全为0的位表示。
- 最小的正非规格化值的位表示，是由最低有效位为1而其他所有位为0构成的。它具有小数(和尾数)值 $M=f=2^{-n}$ 和阶码值 $E=-2^{k-1}+2$ 。因此它的数字值是 $V=2^{-n-2^{k-1}+2}$ 。
- 最大的非规格化值的位模式是由全为0的阶码字段和全为1的小数字段组成的。它有小数(和尾数)值 $M=f=1-2^{-n}$ (我们写成 $1-$)和阶码值 $E=-2^{k-1}+2$ 。因此，数值 $V=(1-2^{-n})\times 2^{-n-2^{k-1}+2}$ ，这仅比最小的规格化值小一点。
- 最小的正规格化值的位模式的阶码字段的最低有效位为1，其他位全为0。它的尾数值 $M=1$ ，而阶码值 $E=-2^{k-1}+2$ 。因此，数值 $V=2^{-2^{k-1}+2}$ 。



- 值 1.0 的位表示的阶码字段除了最高有效位等于 1 以外，其他位都等于 0。它的尾数值是 $M = 1$ ，而它的阶码值是 $E = 0$ 。
- 最大的规格化值的位表示的符号位为 0，阶码的最低有效位等于 0，其他位等于 1。它的小数值 $f = 1 - 2^{-n}$ ，尾数 $M = 2 - 2^{-n}$ （我们写作 $2 - \epsilon$ ）。它的阶码值 $E = 2^{k-1} - 1$ ，得到数值 $V = (2 - 2^{-n}) \times 2^{2^{k-1}-1} = (1 - 2^{-n-1}) \times 2^{2^{k-1}}$ 。

描述	exp	frac	单精度		双精度	
			值	十进制	值	十进制
0	00...00	0...00	0	0.0	0	0.0
最小非规格化数	00...00	0...01	$2^{-23} \times 2^{-126}$	1.4×10^{-45}	$2^{-52} \times 2^{-1022}$	4.9×10^{-324}
最大非规格化数	00...00	1...11	$(1 - \epsilon) \times 2^{-126}$	1.2×10^{-38}	$(1 - \epsilon) \times 2^{-1022}$	2.2×10^{-308}
最小规格化数	00...01	0...00	1×2^{-126}	1.2×10^{-38}	1×2^{-1022}	2.2×10^{-308}
1	01...11	0...00	1×2^0	1.0	1×2^0	1.0
最大规格化数	11...10	1...11	$(2 - \epsilon) \times 2^{127}$	3.4×10^{38}	$(2 - \epsilon) \times 2^{1023}$	1.8×10^{308}

图 2-35 非负浮点数的示例

练习把一些整数值转换成浮点形式对理解浮点表示很有用。例如，在图 2-14 中我们看到 12 345 具有二进制表示 [11000000111001]。通过将二进制小数点左移 13 位，我们创建这个数的一个规格化表示，得到 $12345 = 1.1000000111001_2 \times 2^{13}$ 。为了用 IEEE 单精度形式来编码，我们丢弃开头的 1，并且在末尾增加 10 个 0，来构造小数字段，得到二进制表示 [10000001110010000000000]。为了构造阶码字段，我们用 13 加上偏置量 127，得到 140，其二进制表示为 [10001100]。加上符号位 0，我们就得到二进制的浮点表示 [0100011001000000111001000000000000]。回想 2.1.4 节，我们观察到整数值 12345 (0x3039) 和单精度浮点值 12345.0 (0x4640E400) 在位级表示上有下列关系：

0	0	0	0	3	0	3	9
00000000000000000000000011000000111001							

4	6	4	0	E	4	0	0
0100011001000000111001000000000000							

现在我们可以看到，相关的区域对应于整数的低位，刚好在等于 1 的最高有效位之前停止（这个位就是隐含的开头的位 1），和浮点表示的小数部分的高位是相匹配的。

练习题 2.48 正如在练习题 2.6 中提到的，整数 3 510 593 的十六进制表示为 0x00359141，而单精度浮点数 3510593.0 的十六进制表示为 0x4A564504。推导出这个浮点表示，并解释整数和浮点数表示的位之间的关系。

练习题 2.49

- 对于一种具有 n 位小数的浮点格式，给出不能准确描述的最小正整数的公式（因为要想准确表示它需要 $n+1$ 位小数）。假设阶码字段长度 k 足够大，可以表示的阶码范围不会限制这个问题。
- 对于单精度格式 ($n = 23$)，这个整数的数字值是多少？

2.4.4 舍入

因为表示方法限制了浮点数的范围和精度，浮点运算只能近似地表示实数运算。因此，对于值 x ，我们一般想用一种系统的方法，能够找到“最接近的”匹配值 x' ，它可以用期望的浮点形式表示出来。这就是舍入 (rounding) 运算的任务。一个关键问题是在两个可能值的中间确定舍入方向。例如，如果我有 1.50 美元，想把它舍入到最接近的美元数，应该是 1 美元还是 2 美元



呢？一种可选择的方法是维持实际数字的下界和上界。例如，我们可以确定可表示的值 x^- 和 x^+ ，使得 x 的值位于它们之间： $x^- \leq x \leq x^+$ 。IEEE 浮点格式定义了四种不同的舍入方式。默认的方法是找到最接近的匹配，而其他三种可用于计算上界和下界。

图 2-36 举例说明了应用四种舍入方式，将一个金额数舍入到最接近的整数美元数。向偶数舍入（round-to-even），也称为向最接近的值舍入（round-to-nearest），是默认的方式，试图找到一个最接近的匹配值。因此，它将 1.40 美元舍入成 1 美元，而将 1.60 美元舍入成 2 美元，因为它们是最接近的整数美元值。唯一的设计决策是确定两个可能结果中间数值的舍入效果。向偶数舍入方式采取的方法是：将数字向上或者向下舍入，使得结果的最低有效数字是偶数。因此，这种方法将 1.5 美元和 2.5 美元都舍入成 2 美元。

方 式	1.40	1.60	1.50	2.50	-1.50
向偶数舍入	1	2	2	2	-2
向零舍入	1	1	1	2	-1
向下舍入	1	1	1	2	-2
向上舍入	2	2	2	3	-1

图 2-36 以美元为例说明舍入方式（单位为美元）

其他三种方式产生实际值的确界（guaranteed bound）。这些方法在一些数字应用中是很有用的。向零舍入方式把正数向下舍入，把负数向上舍入，得到值 \hat{x} ，使得 $|\hat{x}| \leq |x|$ 。向下舍入方式把正数和负数都向下舍入，得到值 x^- ，使得 $x^- \leq x$ 。向上舍入方式把正数和负数都向上舍入，得到值 x^+ ，满足 $x \leq x^+$ 。

向偶数舍入最初看上去好像是个相当随意的目标——有什么理由偏向取偶数呢？为什么不始终把位于两个可表示的值中间的值都向上舍入呢？使用这种方法的一个问题就是很容易假想到这样的情景：这种方法舍入一组数值，会在计算这些值的平均数中引入统计偏差。我们采用这种方式舍入得到的一组数的平均值将比这些数本身的平均值略高一些。相反，如果我们总是把两个可表示值中间的数字向下舍入，那么舍入后的一组数的平均值将比这些数本身的平均值略低一些。向偶数舍入在大多数现实情况中避免了这种统计偏差。在 50% 的时间里，它将向上舍入，而在 50% 的时间里，它将向下舍入。

在我们不想舍入到整数时，也可以使用向偶数舍入。我们只是简单地考虑最低有效数字是奇数还是偶数。例如，假设我们想将十进制数舍入到最接近的百分位。不管用那种舍入方式，我们都将把 1.2349999 舍入到 1.23，而将 1.2350001 舍入到 1.24，因为它们不是在 1.23 和 1.24 的正中间。另一方面我们将把两个数 1.2350000 和 1.2450000 都舍入到 1.24，因为 4 是偶数。

相似地，向偶数舍入法能够运用于二进制小数。我们将最低有效位的值 0 认为是偶数，值 1 认为是奇数。一般来说，只有对形如 $XX\cdots X.YY\cdots Y100\cdots$ 的二进制位模式的数，这种舍入方式才有效，其中 X 和 Y 表示任意位值，最右边的 Y 是要舍入的位置。只有这种位模式表示在两个可能的结果正中间的值。例如，考虑舍入值到最近的四分之一的问题（也就是二进制小数点右边 2 位）。我们将 $10.00011_2(2\frac{3}{32})$ 向下舍入到 $10.00_2(2)$ ， $10.00110_2(2\frac{3}{16})$ 向上舍入到 $10.01_2(2\frac{1}{4})$ ，因为这些值不是两个可能值的正中间值。我们将 $10.11100_2(2\frac{7}{8})$ 向上舍入成 $11.00_2(3)$ ，而 $10.10100_2(2\frac{5}{8})$ 向下舍入成 $10.10_2(2\frac{1}{2})$ ，因为这些值是两个可能值的中间值，并且我们倾向于使最低有效位为零。

练习题 2.50 根据舍入到偶数规则，说明如何将下列二进制小数值舍入到最接近的二分之一（二进制小数点右边 1 位）。对每种情况，给出舍入前后的数字值。

- A. 10.010_2 B. 10.011_2 C. 10.110_2 D. 11.001_2



练习题 2.51 在练习题 2.46 中我们看到，爱国者导弹软件将 0.1 近似表示为 $x = 0.00011001100110011001100_2$ 。

假设使用 IEEE 舍入到偶数方式确定 0.1 的二进制小数点右边 23 位的近似表示 x' 。

- A. x' 的二进制表示是什么？
- B. $x' - 0.1$ 的十进制表示的近似值是什么？
- C. 运行 100 小时后，计算时钟值会有多少偏差？
- D. 该程序对飞毛腿导弹位置的预测会有多少偏差？

练习题 2.52 考虑下列基于 IEEE 浮点格式的 7 位浮点表示。两个格式都没有符号位——它们只能表示非负的数字。

1. 格式 A

- 有 $k=3$ 个阶码位。阶码的偏置值是 3。
- 有 $n=4$ 个小数位。

2. 格式 B

- 有 $k=4$ 个阶码位。阶码的偏置值是 7。
- 有 $n=3$ 个小数位。

下面给出了一些格式 A 表示的位模式，你的任务是将它们转换成格式 B 中最接近的值。如果有必要，请使用舍入到偶数的原则。另外，给出由格式 A 和格式 B 表示的位模式对应的数字的值。给出整数（例如 17）或者小数（例如 17/64）。

格式A		格式B	
位	值	位	值
011 0000	1	0111 000	1
101 1110			
010 1001			
110 1111			
000 0001			

2.4.5 浮点运算

IEEE 标准指定了一个简单的规则，用来确定诸如加法和乘法这样的算术运算的结果。把浮点值 x 和 y 看成实数，而某个运算 \odot 定义在实数上，计算将产生 $\text{Round}(x \odot y)$ ，这是对实际运算的精确结果进行舍入后的结果。在实际中，浮点单元的设计者使用一些聪明的小技巧来避免执行这种精确的计算，因为计算只要精确到能够保证得到一个正确的舍入结果就可以了。当参数中有一个是特殊值（如 -0 、 $-\infty$ 或 NaN ）时，IEEE 标准定义了一些使之更合理的规则。例如，定义 $1/-0$ 将产生 $-\infty$ ，而定义 $1/+0$ 会产生 $+\infty$ 。

IEEE 标准中指定浮点运算行为方法的一个优势在于，它可以独立于任何具体的硬件或者软件实现。因此，我们可以检查它的抽象数学属性，而不必考虑实际上它是如何实现的。

前面我们看到了整数（包括无符号和补码）加法形成了阿贝尔群。实数上的加法也形成了阿贝尔群，但是我们必须考虑舍入对这些属性的影响。我们将 $x +^f y$ 定义为 $\text{Round}(x+y)$ 。这个运算的定义针对 x 和 y 的所有取值，但是虽然 x 和 y 都是实数，由于溢出，该运算可能得到无穷值。对于所有 x 和 y 的值，这个运算是可交换的，也就是说 $x +^f y = y +^f x$ 。另一方面，这个运算是不可结合的。例如，使用单精度浮点，表达式 $(3.14+1e10)-1e10$ 求值得到 0.0——因为舍入，值 3.14 会丢失。另一方面，表达式 $3.14+(1e10-1e10)$ 得到值 3.14。作为阿贝尔群，大多数数值的浮点加法都有逆元，也就是说 $x +^f -x = 0$ 。无穷（因为 $+\infty - \infty = \text{NaN}$ ）和 NaN 是例外情况，因为对于任何 x ，都有 $\text{NaN} +^f x = \text{NaN}$ 。

浮点加法不具有结合性，这是缺少的最重要的群属性。对于科学计算程序员和编译器编写者来说，这具有重要的含义。例如，假设一个编译器给定了如下代码片段：



```
x = a + b + c;  
y = b + c + d;
```

编译器可能试图产生下列代码来省去一个浮点加法：

```
t = b + c;  
x = a + t;  
y = t + d;
```

然而，对于 x 来说，这个计算可能会产生与原始值不同的值，因为它使用了加法运算的不同结合方式。在大多数应用中，这种差异小得无关紧要。不幸的是，编译器无法知道在效率和忠实于原始程序的确切行为之间，使用者愿意做出什么样的选择。结果是，编译器倾向于保守，避免任何对功能产生影响的优化，即使是很轻微的影响。

另一方面，浮点加法满足了单调性属性：如果 $a < b$ ，那么对于任何 a 、 b 以及 x 的值，除了 NaN ，都有 $x + a < x + b$ 。无符号或补码加法不具有这个实数（和整数）加法的属性。

浮点乘法也遵循通常乘法所具有的许多属性。我们定义 $x^f y$ 为 *Round* ($x \times y$)。这个运算在乘法中是封闭的（虽然可能产生无穷大或 NaN ），它是可交换的，并且它的乘法单位元为 1.0。另一方面，由于可能发生溢出，或者由于舍入而失去精度，它不具有可结合性。例如，在单精度浮点情况下，表达式 $(1e20 * 1e20) * 1e-20$ 的值为 $+\infty$ ，而 $1e20 * (1e20 * 1e-20)$ 将得出 $1e20$ 。另外，浮点乘法在加法上不具备分配性。例如，在单精度浮点情况下，表达式 $1e20 * (1e20 - 1e20)$ 的值为 0.0，而 $1e20 * 1e20 - 1e20 * 1e20$ 会得出 NaN 。

另一方面，对于任何 a 、 b 和 c ，并且 a 、 b 和 c 都不等于 NaN ，浮点乘法满足下列单调性：

$$\begin{array}{llll} a & b \text{ 且 } c & 0 & a^f c & b^f c \\ a & b \text{ 且 } c & 0 & a^f c & b^f c \end{array}$$

此外，我们还可以保证，只要 $a \neq NaN$ ，就有 $a^f a = 0$ 。像我们先前所看到的，无符号或补码的乘法没有这些单调性属性。

对于科学计算程序员和编译器编写者来说，缺乏结合性和分配性是很严重的问题。即使为了在三维空间中确定两条线是否交叉而写代码这样看上去很简单的任务，也可能成为一个很大的挑战。

2.4.6 C 语言中的浮点数

所有的 C 语言版本提供了两种不同的浮点数据类型：`float` 和 `double`。在支持 IEEE 浮点格式的机器上，这些数据类型就对应于单精度和双精度浮点。另外，这类机器使用向偶数舍入的方式。不幸的是，因为 C 语言标准不要求机器使用 IEEE 浮点，所以没有标准的方法来改变舍入方式或者得到诸如 -0 、 $+\infty$ 、 $-\infty$ 或者 NaN 之类的特殊值。大多数系统提供 `include('h')` 文件和读取这些特征的过程库，但是细节因为系统不同而不同。例如，当程序文件中出现下列句子时，GNU 编译器 GCC 会定义程序常数 `INFINITY`（表示 $+\infty$ ）和 `NAN`（表示 NaN ）。

```
# define _GNU_SOURCE 1  
# include <math.h>
```

较新版本的 C 语言，包括 ISO C99，包含第三种浮点数据类型 `long double`。对于许多机器和编译器来说，这种数据类型等价于 `double` 数据类型。不过对于 Intel 兼容机来说，GCC 用 80 位“扩展精度”格式来实现这种数据类型，提供了比标准 64 位格式大得多的取值范围和精度。家庭作业 2.85 研究了这种格式的属性。

练习题 2.53 完成下列宏定义，生成双精度值 $+\infty$ 、 $-\infty$ 和 0。

```
#define POS_INFINITY
```



```
#define NEG_INFINITY  
#define NEG_ZERO
```

不能使用任何 `include` 文件（例如 `math.h`），但你能利用的是：双精度能够表示的最大的有限数，大约是 1.8×10^{308} 。

当在 `int`、`float` 和 `double` 格式之间进行强制类型转换时，程序改变数值和位模式的原则如下（假设 `int` 是 32 位的）：

- 从 `int` 转换成 `float`，数字不会溢出，但是可能被舍入。
- 从 `int` 或 `float` 转换成 `double`，因为 `double` 有更大的范围（也就是可表示值的范围），也有更高的精度（也就是有效位数），所以能够保留精确的数值。
- 从 `double` 转换成 `float`，因为范围要小一些，所以值可能溢出成为 $+\infty$ 或 $-\infty$ 。另外，由于精确度较小，它还可能被舍入。
- 从 `float` 或者 `double` 转换成 `int`，值将会向零舍入。例如，`1.999` 将被转换成 `1`，而 `-1.999` 将被转换成 `-1`。进一步来说，值可能会溢出。C 语言标准没有对这种情况指定固定的结果。与 Intel 兼容的微处理器指定位模式 [10...00]（字长为 w 时的 $TMin_w$ ）为整数不确定（integer indefinite）值。一个从浮点数到整数的转换，如果不能为该浮点数找到一个合理的整数近似值，就会产生这样一个值。因此，表达式 `(int)+1e10` 会得到 `-21483648`，即从一个正值变成了一个负值。

网络旁注 DATA : IA32-FP : Intel IA32 的浮点运算

在下一章，我们将深入研究 Intel IA32 处理器，这种处理器大量地应用于今天的个人计算机中。这里，我们重点突出这种机器的一个特性，即用 GCC 编译的时候，它能够严重影响程序对浮点数运算的行为。

像大多数其他处理器一样，IA32 处理器有特别的存储器元素，称为寄存器，当计算或者使用浮点数时，用来保存浮点值。IA32 非同一般的属性是，浮点寄存器使用一种特殊的 80 位的扩展精度格式。与存储器中保存值所使用的普通 32 位单精度和 64 位双精度格式相比，它提供了更大的表示范围和更高的精度。（参见家庭作业 2.85。）所有的单精度和双精度数在从存储器加载到浮点寄存器中时，都会转换成这种格式。运算总是以扩展精度格式进行的。当数字存储在存储器中时，它们就从扩展精度转换成单精度或者双精度格式。

对于程序员而言，把所有寄存器数据扩展成 80 位，并把所有存储器数据收缩成更小的格式的做法，会产生一些不太好的结果。这意味着从寄存器存储一个值到存储器中，然后再把它取回到寄存器中，由于舍入、下溢或者上溢，可能会改变它的值。对于 C 语言程序员来说，这种存入和取出并不总是可见的，因而会导致一些非常异常的结果。

较新版本的 Intel 处理器，包括 IA32 和较新的 64 位机器，对单精度和双精度浮点运算提供了直接的硬件支持。随着新硬件以及基于较新的浮点指令产生代码的新编译器的使用，以前 IA32 做法导致的这些奇怪特性会逐渐消失。

Ariane 5——浮点溢出的高昂代价

将大的浮点数转换成整数是一种常见的程序错误来源。1996 年 6 月 4 日，Ariane 5 火箭初次航行，一个错误便产生了灾难性的后果。发射后仅仅 37 秒，火箭偏离了它的飞行路径，解体并且爆炸。火箭上载有价值 5 亿美元的通信卫星。

后来的调查 [69, 39] 显示，控制惯性导航系统的计算机向控制引擎喷嘴的计算机发送了一个无效数据。它没有发送飞行控制信息，而是送出了一个诊断位模式，表明将一个 64 位浮点数转换成 16 位有符号整数时，产生了溢出。



溢出的值测量的是火箭的水平速率，这比早先的 Ariane 4 火箭所能达到的速度高出了 5 倍。设计 Ariane 4 火箭软件的时候，他们小心地分析了这些数字值，并且确定水平速率决不会超出一个 16 位数的表示范围。不幸的是，他们在 Ariane 5 火箭的系统中简单地重用了这一部分，而没有检查它所基于的假设。



练习题 2.54 假定变量 x 、 f 和 d 的类型分别是 `int`、`float` 和 `double`。除了 f 和 d 都不能等于 $+\infty$ 、 $-\infty$ 或者 NaN 之外，它们的值是任意的。下面每个 C 表达式，证明它总是为真（也就是求值为 1），或者给出一个使表达式不为真的值（也就是求值为 0）。

- A. $x == (int)(double)x$
- B. $x == (int)(float)x$
- C. $d == (double)(float)d$
- D. $f == (float)(double)f$
- E. $f == -(-f)$
- F. $1.0/2 == 1/2.0$
- G. $d * d >= 0.0$
- H. $(f+d) - f == d$

2.5 小结

计算机将信息按位编码，通常组织成字节序列。用不同的编码方式表示整数、实数和字符串。不同的计算机模型在编码数字和多字节数据中的字节排序时使用不同的约定。

C 语言的设计可以包容多种不同字长和数字编码的实现。虽然高端机器逐渐开始使用 64 位字长，但是目前大多数机器仍使用 32 位字长。大多数机器对整数使用补码编码，而对浮点数使用 IEEE 浮点编码。在位级上理解这些编码，并且理解算术运算的数学特性，对于想使编写的程序能在全部数值范围内正确运算的程序员来说，是很重要的。

在相同长度的无符号和有符号整数之间进行强制类型转换时，大多数 C 语言实现遵循的原则是底层的位模式不变。在补码机器上，对于一个 w 位的值，这种行为是由函数 $T2U_w$ 和 $U2T_w$



来描述的。C 语言隐式的强制类型转换会出现许多程序员无法预计的结果，常常导致程序错误。

由于编码的长度有限，与传统整数和实数运算相比，计算机运算具有完全不同的属性。当超出表示范围时，有限长度能够引起数值溢出。当浮点数非常接近于 0.0，从而转换成零时，也会下溢。

和大多数其他程序语言一样，C 语言实现的有限整数运算和真实的整数运算相比，有一些特殊的属性。例如，由于溢出，表达式 $x * x$ 能够得出负数。但是，无符号数和补码的运算都满足整数运算的许多其他属性，包括结合律、交换律和分配律。这就允许编译器做很多的优化。例如，用 $(x < 3) - x$ 取代表达式 $7 * x$ 时，我们就利用了结合律、交换律和分配律的属性，还利用了移位和乘以 2 的幂之间的关系。

我们已经看到了几种使用位级运算和算术运算组合的聪明方法。例如，使用补码运算， $\sim x + 1$ 等价于 $-x$ 。另外一个例子，假设我们想要一个形如 $[0, \dots, 0, 1, \dots, 1]$ 的位模式，由 $w - k$ 个 0 后面紧跟着 k 个 1 组成。这些位模式有助于掩码运算。这种模式能够通过 C 表达式 $(1 \ll k) - 1$ 生成，利用的是这样一个属性，即我们想要的位模式的数值为 $2^k - 1$ 。例如，表达式 $(1 \ll 8) - 1$ 将产生位模式 0xFF。

浮点表示通过将数字编码为 $x \times 2^y$ 的形式来近似地表示实数。最常见的浮点表示方式是由 IEEE 标准 754 定义的。它提供了几种不同的精度，最常见的是单精度（32 位）和双精度（64 位）。IEEE 浮点也能够表示特殊值 $+\infty$ 、 $-\infty$ 和 NaN 。

必须非常小心地使用浮点运算，因为浮点运算只有有限的范围和精度，而且不遵守普遍的算术属性，比如结合性。

参考文献说明

关于 C 语言的参考书 [48, 58] 讨论了不同的数据类型和运算的属性。（这两本书中，只有 Steele 和 Harbison 的书 [48] 涵盖了 ISO C99 的新特性。）对于精确的字长或者数字编码 C 语言标准没有详细的定义。这些细节是故意省去的，这样可以在更大范围的不同机器上实现 C 语言。已经有几本书 [59, 70] 给了 C 语言程序员一些建议，警告他们关于溢出、隐式强制类型转换到无符号数，以及其他一些已经在这一章中谈及的陷阱。这些书还提供了对变量命名、编码风格和代码测试的有益建议。Seacord 的书 [94] 是关于 C 和 C++ 程序中的安全问题的，本书结合了 C 程序的有关信息，如何编译和执行程序，以及漏洞是如何造成的。关于 Java 的书（我们推荐 Java 语言的创始人 James Gosling 参与编写的一本书 [4]）描述了 Java 支持的数据格式和算术运算。

关于逻辑设计的书 [56, 115] 都有关于编码和算术运算的章节，描述了实现算术电路的不同方式。Overton 的关于 IEEE 浮点数的书 [78]，从数字应用程序员的角度，详细描述了格式和属性。

家庭作业

- * 2.55 在你能够访问的不同机器上，使用 `show_bytes`（文件 `show-bytes.c`）编译并运行示例代码。确定这些机器使用的字节顺序。
- * 2.56 试着用不同的示例值来运行 `show_bytes` 的代码。
- * 2.57 编写程序 `show_short`、`show_long` 和 `show_double`，它们分别打印类型为 `short int`、`long int` 和 `double` 的 C 语言对象的字节表示。请试着在几种机器上运行。
- ** 2.58 编写过程 `is_little_endian`，当在小端法机器上编译和运行时返回 1，在大端法机器上编译运行时则返回 0。这个程序应该可以运行在任何机器上，无论机器的字长是多少。
- ** 2.59 编写一个 C 表达式，使它生成一个字，由 `x` 的最低有效字节和 `y` 中剩下的字节组成。对于运算数



$x=0x89ABCDEF$ 和 $y=0x76543210$, 就得到 $0x765432EF$ 。

- **2.60** 假设我们将一个 w 位的字中的字节从 0(最低位) 到 $w/8-1$ (最高位) 编号。写出下面 C 函数的代码, 它会返回一个无符号值, 其中参数 x 的字节 i 被替换成字节 b :

```
unsigned put_byte (unsigned x, unsigned char b int i);
```

以下的一些示例, 说明了这个函数该如何工作:

```
replace_byte(0x12345678, 0xAB, 2) --> 0x12AB5678  
replace_byte(0x12345678, 0xAB, 0) --> 0x123456AB
```

位级整数编码规则

在接下来的作业中, 我们特意限制了你能使用的编程结构, 来帮你更好地理解 C 语言的位级、逻辑和算术运算。在回答这些问题时, 你的代码必须遵守下面这些规则:

- 假设

- 整数用补码形式表示。
- 有符号数的右移是算术右移。
- 数据类型 `int` 是 w 位长的。对于某些题目, 会给定 w 的值, 但是在其他情况下, 只要 w 是 8 的整数倍, 你的代码就应该能工作。你可以用表达式 `sizeof(int)<<3` 来计算 w 。

- 禁止使用

- 条件语句 (`if` 或者 `?:`)、循环、分支语句、函数调用和宏调用。
- 除法、模运算和乘法。
- 相对比较运算符 (`<`、`>`、`<=` 和 `>=`)。
- 强制类型转换, 无论是显式的还是隐式的。

- 允许的运算

- 所有的位级和逻辑运算。
- 左移和右移, 但是位移的数量只能在 0 和 $w - 1$ 之间。
- 加法和减法。
- 相等 (`==`) 和不相等 (`!=`) 测试。(在有些题目中, 也不允许这些运算。)
- 整型常数 `INT_MIN` 和 `INT_MAX`。

即使有这些条件的限制, 你仍然可以选择描述性的变量名, 并且使用注释来描述你的解决方案的逻辑, 尽量提高代码的可读性。例如, 下面这段代码从整数参数 x 中抽取出最高有效字节:

```
/* Get most significant byte from x */  
int get_msb(int x) {  
    /* Shift by w-8 */  
    int shift_val = (sizeof(int)-1)<<3;  
    /* Arithmetic shift */  
    int xright = x >> shift_val;  
    /* Zero all but LSB */  
    return xright & 0xFF;  
}
```

- **2.61** 写一个 C 表达式, 在下列描述的条件下产生 1, 而在其他情况下得到 0。假设 x 是 `int` 类型。

- A. x 的任何位都等于 1。
- B. x 的任何位都等于 0。
- C. x 的最高有效字节中的位都等于 1。
- D. x 的最低有效字节中的位都等于 0。

代码应该遵循位级整数编码规则, 另外还有一个限制, 你不能使用相等 (`==`) 和不相等 (`!=`) 测试。

- **2.62** 编写一个函数 `int_shifts_are_logical()`, 在对 `int` 类型的数使用算术右移的机器上运行时, 这个函数生成 1, 而其他情况下生成 0。你的代码应该可以运行在任何字长的机器上。在几种机器上



测试你的代码。

- ** 2.63** 将下面的 C 函数代码补充完整。函数 `srl` 用算术右移（由值 `xsra` 给出）来完成逻辑右移，后面的操作不包括右移或者除法。函数 `sra` 用逻辑右移（由值 `xsrl` 给出）来完成算术右移，后面的操作不包括右移或者除法。可以通过计算 `8*sizeof(int)` 来确定数据类型 `int` 中的位数 `w`。位移量 `k` 的取值范围为 `0 ~ w - 1`。

```
int sra(int x, int k) {
    /* Perform shift logically */
    int xsrl = (unsigned) x >> k;
    :
}
unsigned srl(unsigned x, int k) {
    /* Perform shift arithmetically */
    unsigned xsra = (int) x >> k;
    :
}
```

- * 2.64** 写出代码实现如下函数：

```
/* Return 1 when any even bit of x equals 1; 0 otherwise.
   Assume w=32 */
int any_even_one(unsigned x);
```

函数应该遵循位级整数编码规则，不过你可以假设数据类型 `int` 有 `w = 32` 位。

- ** 2.65** 写出代码实现如下函数：

```
/* Return 1 when x contains an even number of 1s; 0 otherwise.
   Assume w=32 */
int even_ones(unsigned x);
```

函数应该遵循位级整数编码规则，不过你可以假设数据类型 `int` 有 `w = 32` 位。

你的代码最多只能包含 12 个算术运算、位运算和逻辑运算。

- ** 2.66** 写出代码实现如下函数：

```
/*
 * Generate mask indicating leftmost 1 in x. Assume w=32.
 * For example 0xFF00 -> 0x8000, and 0x6600 --> 0x4000.
 * If x = 0, then return 0.
 */
int leftmost_one(unsigned x);
```

函数应该遵循位级整数编码规则，不过你可以假设数据类型 `int` 有 `w = 32` 位。

你的代码最多只能包含 15 个算术运算、位运算和逻辑运算。

提示：先将 `x` 转换成形如 [0...011...1] 的位向量。

- ** 2.67** 给你一个任务，编写一个过程 `int_size_is_32()`，当在一个 `int` 是 32 位的机器上运行时，该程序产生 1，而其他情况则产生 0。不允许使用 `sizeof` 运算符。下面是开始时的尝试：

```
1  /* The following code does not run properly on some machines */
2  int bad_int_size_is_32() {
3      /* Set most significant bit (msb) of 32-bit machine */
4      int set_msbit = 1 << 31;
5      /* Shift past msb of 32-bit word */
6      int beyond_msbit = 1 << 32;
7
8      /* set_msbit is nonzero when word size >= 32
       beyond_msbit is zero when word size <= 32 */
```



```
10     return set_msb && !beyond_msb;
11 }
```

当在 SUN SPARC 这样的 32 位机器上编译并运行时，这个过程返回的却是 0。下面的编译器信息给了我们一个问题的指示：

```
warning: left shift count >= width of type
A. 我们的代码在哪个方面没有遵守 C 语言标准？
B. 修改代码，使得它在 int 至少为 32 位的任何机器上都能正确地运行。
C. 修改代码，使得它在 int 至少为 16 位的任何机器上都能正确地运行。
```

****2.68** 写出具有如下原型的函数的代码：

```
/*
 * Clear all but least significant n bits of x
 * Examples: x = 0x78ABCDEF, n = 8 --> 0xEF, n = 16 --> 0xCDEF
 * Assume 1 <= n <= w
 */
int lower_bits(int x, int n);
```

函数应该遵循位级整数编码规则。要注意 $n=w$ 的情况。

**** 2.69** 写出具有如下原型的函数的代码：

```
/*
 * Do rotating right shift. Assume 0 <= n < w
 * Examples when x = 0x12345678 and w = 32:
 *   n=4 -> 0x81234567, n=20 -> 0x45678123
 */
unsigned rotate_right(unsigned x, int n);
```

函数应该遵循位级整数编码规则。要注意 $n=0$ 的情况。

**** 2.70** 写出具有如下原型的函数的代码：

```
/*
 * Return 1 when x can be represented as an n-bit, 2's complement
 * number; 0 otherwise
 * Assume 1 <= n <= w
 */
int fits_bits(int x, int n);
```

函数应该遵循位级整数编码规则。

*** 2.71** 你刚刚开始在一家公司工作，他们要实现一组过程来操作一个数据结构，要将 4 个有符号字节封装成一个 32 位 unsigned。一个字节从 0 (最低有效字节) 编号到 3 (最高有效字节)。分配给你的任务是：为一个使用补码运算和算术右移的机器编写一个具有如下原型的函数：

```
/* Declaration of data type where 4 bytes are packed
   into an unsigned */
typedef unsigned packed_t;

/* Extract byte from word. Return as signed integer */
int xbyte(packed_t word, int bytenum);
```

也就是说，函数会抽取出指定的字节，再把它符号扩展为一个 32 位 int。

你的前任（因为水平不够高而被解雇了）编写了下面的代码：



```
/* Failed attempt at xbyte */
int xbyte(packed_t word, int bytenum)
{
    return (word >> (bytenum << 3)) & 0xFF;
}
```

- A. 这段代码错在哪里？
- B. 给出函数的正确实现，只能使用左右移位和一个减法。

**** 2.72** 给你一个任务，写一个函数，将整数 val 复制到缓冲区 buf 中，但是只有当缓冲区中有足够可用的空间时，才执行复制。

你写的代码如下：

```
/* Copy integer into buffer if space is available */
/* WARNING: The following code is buggy */
void copy_int(int val, void *buf, int maxbytes) {
    if (maxbytes-sizeof(val) >= 0)
        memcpy(buf, (void *) &val, sizeof(val));
}
```

这段代码使用了库函数 memcpy。虽然在这里用这个函数有点刻意，因为我们只是想复制一个 int，但是它说明了一种复制较大数据结构的常见方法。

你仔细地测试了这段代码后发现，哪怕 maxbytes 很小的时候，它也能把值复制到缓冲区中。

- A. 解释为什么代码中的条件测试总是成功。**提示：**sizeof 运算符返回类型为 size_t 的值。
- B. 你该如何重写这个条件测试，使之工作正确。

**** 2.73** 写出具有如下原型的函数的代码：

```
/* Addition that saturates to TMin or TMax */
int saturating_add(int x, int y);
```

同正常的补码加法溢出的方式不同，当正溢出时，saturating_add 返回 TMax，负溢出时，返回 TMin。这种运算常常用在执行数字信号处理的程序中。

你的函数应该遵循位级整数编码规则。

**** 2.74** 写出具有如下原型的函数的代码：

```
/* Determine whether subtracting arguments will cause overflow */
int tsub_ovf(int x, int y);
```

如果计算 $x - y$ 导致溢出，这个函数就返回 1。

**** 2.75** 假设我们想要计算 $x \cdot y$ 的完整的 $2w$ 位表示，其中， x 和 y 都是无符号数，并且运行在数据类型 unsigned 是 w 位的机器上。乘积的低 w 位能够用表达式 $x \cdot y$ 计算，所以，我们只需要一个具有下列原型的函数：

```
int signed_high_prod(int x, int y);
```

这个函数计算无符号变量 $x \cdot y$ 的高 w 位。

我们使用一个具有下面原型的库函数：

```
unsigned unsigned_high_prod(unsigned x, unsigned y);
```

它计算在 x 和 y 采用补码形式的情况下， $x \cdot y$ 的高 w 位。编写代码调用这个过程，以实现用无符号数为参数的函数。验证你的解答的正确性。

提示：看看等式 (2-18) 的推导中，有符号乘积 $x \cdot y$ 和无符号乘积 $x' \cdot y'$ 之间的关系。

**** 2.76** 假设我们有一个任务：生成一段代码，将整数变量 x 乘以不同的常数因子 K 。为了提高效率，我们想只使用 +、- 和 << 运算。对于下列 K 的值，写出执行乘法运算的 C 表达式，每个表达式中最多



使用 3 个运算。

- A. $K = 5$:
- B. $K = 9$:
- C. $K = 30$:
- D. $K = -56$:

****2.77** 写出具有如下原型的函数的代码：

```
/* Divide by power of two. Assume 0 <= k < w-1 */  
int divide_power2(int x, int k);
```

该函数要用正确的舍入方式计算 $x/2^k$ ，并且应该遵循位级整数编码规则。

****2.78** 写出函数 mul5div8 的代码，对于整数参数 x ，计算 $5*x/8$ ，但是要遵循位级整数编码规则。你的代码计算 $5*x$ 也会产生溢出。

****2.79** 写出函数 five_eighths 的代码，对于整数参数 x ，计算 $5/8x$ 的值，向零舍入。它不会溢出。函数应该遵循位级整数编码规则。

****2.80** 编写 C 表达式产生如下位模式，其中 a^n 表示符号 a 重复 n 次。假设一个 w 位的数据类型。你的代码可以包含对参数 m 和 n 的引用，它们分别表示 m 和 n 的值，但是不能使用表示 w 的参数。

- A. $1^{w-n}0^n$ 。
- B. $0^{w-n-m}1^n0^m$ 。

***2.81** 我们在一个 int 类型值为 32 位的机器上运行程序。这些值以补码形式表示，而且它们都是算术右移的。unsigned 类型的值也是 32 位的。

我们产生随机数 x 和 y ，并且把它们转换成无符号数，显示如下：

```
/* Create some arbitrary values */  
int x = random();  
int y = random();  
/* Convert to unsigned */  
unsigned ux = (unsigned) x;  
unsigned uy = (unsigned) y;
```

对于下列每个 C 表达式，你要指出表达式是否总是为 1。如果它总是为 1，那么请描述其中的数学原理。否则，列举一个使它为 0 的参数示例。

- A. $(x>y) == (-x<-y)$
- B. $((x+y)<<5) + x-y == 31*y+33*x$
- C. $\sim x + \sim y == \sim(x+y)$
- D. $(int)(ux-uy) == -(y-x)$
- E. $((x >> 1) << 1) <= x$

****2.82** 一些数字的二进制表示是由形如 $0.y\ y\ y\ y\ y\dots$ 的无穷串组成的，其中 y 是一个 k 位的序列。例如， $\frac{1}{3}$ 的二进制表示是 $0.01010101\dots$ ($y = 01$)，而 $\frac{1}{5}$ 的二进制表示是 $0.001100110011\dots$ ($y = 0011$)。

A. 设 $Y = B2U_k(y)$ ，也就是说，这个数具有二进制表示 y 。给出一个由 Y 和 k 组成的公式表示这个无穷串的值。**提示**：请考虑将二进制小数点右移 k 位的结果。

B. 对于下列 y 的值，串的数值是多少？

- (a) 001
- (b) 1001
- (c) 000111

***2.83** 填写下列程序的返回值，这个程序是测试它的第一个参数是否大于或者等于第二个参数。假定函数



`f2u` 返回一个无符号 32 位数字，其位表示与它的浮点参数相同。你可以假设两个参数都不是 `Nan`。两种 0, +0 和 -0 被认为是相等的。

```
int float_ge(float x, float y) {
    unsigned ux = f2u(x);
    unsigned uy = f2u(y);
    /* Get the sign bits */
    unsigned sx = ux >> 31;
    unsigned sy = uy >> 31;

    /* Give an expression using only ux, uy, sx, and sy */
    return _____;
}
```

- *2.84 给定一个浮点格式，有 k 位指数和 n 位小数，对于下列数，写出阶码 E 、尾数 M 、小数 f 和值 V 的公式。另外，请描述其位表示。

- A. 数 5.0。
- B. 能够被准确描述的最大奇整数。
- C. 最小的规格化数的倒数。

- *2.85 与 Intel 兼容的处理器也支持“扩展精度”浮点形式，这种格式具有 80 位字长，被分成 1 个符号位、 $k = 15$ 个阶码位、1 个单独的整数位和 $n = 63$ 个小数位。整数位是 IEEE 浮点表示中隐含位的显式副本。也就是说，对于规格化的值它等于 1，对于非规格化的值它等于 0。填写下表，给出用这种格式表示的一些“有趣的”数字的近似值。

描述	扩展精度	
	值	十进制
最小的正非规格化数		
最小的正规格化数		
最大的规格化数		

- *2.86 考虑一个基于 IEEE 浮点格式的 16 位浮点表示，它具有 1 个符号位、7 个阶码位 ($k = 7$) 和 8 个小数位 ($n = 8$)。阶码偏置量是 $2^{7-1} - 1 = 63$ 。

对于每个给定的数，填写下表，其中，每一列具有如下指示说明：

Hex：描述编码形式的 4 个十六进制数字。

M ：尾数的值。这应该是一个形如 x 或 $\frac{x}{y}$ 的数，其中 x 是一个整数，而 y 是 2 的整数幂。例如：0, $\frac{67}{64}$ 和 $\frac{1}{256}$ 。
 E ：阶码的整数值。

V ：所表示的数字值。使用 x 或者 $x \times 2^z$ 表示，其中 x 和 z 都是整数。

举一个例子，为了表示数 $\frac{7}{2}$ ，我们有 $s = 0$, $M = \frac{7}{4}$ 和 $E = 1$ 。因此这个数的阶码字段为 0x40 (十进制值 $63+1=64$)，尾数字段为 0xC0 (二进制 11000000_2)，得到一个十六进制的表示 40C0。

标记为“—”的条目不用填写。

描述	Hex	M	E	V
-0				—
最小的值 > 1				—
256				—
最大的非规格化数				—
-	—	—	—	—
十六进制表示为 3AA0 的数	—			

- **2.87 考虑下面两个基于 IEEE 浮点格式的 9 位浮点表示。



1. 格式 A

- 有一个符号位。
- 有 $k=5$ 个阶码位。阶码偏置量是 15。
- 有 $n=3$ 个小数位。

2. 格式 B

- 有一个符号位。
- 有 $k=4$ 个阶码位。阶码偏置量是 7。
- 有 $n=4$ 个小数位。

下面给出了一些格式 A 表示的位模式，你的任务是把它们转换成最接近的格式 B 表示的值。如果需要舍入，你要向 $+\infty$ 舍入。另外，给出用格式 A 和格式 B 表示的位模式对应的值。要么是整数（例如，17），要么是小数（例如， $17/64$ 或 $17/2^6$ ）。

格式A		格式B	
位	值	位	值
1 01110 001	$\frac{-9}{16}$	1 0110 0010	$\frac{-9}{16}$
0 10110 101			
1 00111 110			
0 00000 101			
1 11011 000			
0 11000 100			

*2.88 我们在一个 int 类型为 32 位补码表示的机器上运行程序。float 类型的值使用 32 位 IEEE 格式，而 double 类型的值使用 64 位 IEEE 格式。

我们产生随机整数 x、y 和 z，并且把它们转换成 double 类型的值：

```
/* Create some arbitrary values */
int x = random();
int y = random();
int z = random();
/* Convert to double */
double dx = (double) x;
double dy = (double) y;
double dz = (double) z;
```

对于下列的每个 C 表达式，你要指出表达式是否总是为 1。如果它总是为 1，描述其中的数学原理。否则，列举出使它为 0 的参数的例子。请注意，不能使用 IA32 机器运行 GCC 来测试你的答案，因为对于 float 和 double，它使用的都是 80 位的扩展精度表示。

- A. $(\text{double})(\text{float}) \ x == dx$
- B. $dx + dy == (\text{double})(x+y)$
- C. $dx + dy + dz == dz + dy + dx$
- D. $dx * dy * dz == dz * dy * dx$
- E. $dx / dx == dy / dy$

*2.89 分配给你一个任务，编写一个 C 函数来计算 2^x 的浮点表示。你意识到完成这个任务的最好方法是直接创建结果的 IEEE 单精度表示。当 x 太小时，你的程序将返回 0.0。当 x 太大时，它会返回 $+\infty$ 。填写下列代码的空白部分，以计算出正确的结果。假设函数 u2f 返回的浮点值与它的无符号参数有相同的位表示。



```
float fpwr2(int x)
{
    /* Result exponent and fraction */
    unsigned exp, frac;
    unsigned u;

    if (x < _____) {
        /* Too small. Return 0.0 */
        exp = _____;
        frac = _____;
    } else if (x < _____) {
        /* Denormalized result */
        exp = _____;
        frac = _____;
    } else if (x < _____) {
        /* Normalized result. */
        exp = _____;
        frac = _____;
    } else {
        /* Too big. Return +oo */
        exp = _____;
        frac = _____;
    }

    /* Pack exp and frac into 32 bits */
    u = exp << 23 | frac;
    /* Return as float */
    return u2f(u);
}
```

- * 2.90 大约在公元前 250 年，希腊数学家阿基米德证明了 $\frac{223}{71} < \pi < \frac{22}{7}$ 。如果当时有一台计算机和标准库 <math.h>，他就能够确定 π 的单精度浮点近似值的十六进制表示为 0x40490FDB。当然，所有的这些都只是近似值，因为 π 不是有理数。

- A. 这个浮点值表示的二进制小数是多少？
- B. $\frac{22}{7}$ 的二进制小数表示是什么？**提示：**参见家庭作业 2.82。
- C. 这两个 $\frac{22}{7}$ 的近似值从哪一位（相对于二进制小数点）开始不同的？

位级浮点编码规则

在接下来的题目中，你要写的代码要实现浮点函数在浮点数的位级表示上直接运算。你的代码应该完全遵循 IEEE 浮点运算的规则，包括当需要舍入时，要使用向偶数舍入的方式。

为此，我们定义数据类型 `float-bits` 等价于 `unsigned`：

```
/* Access bit-level representation floating-point number */
typedef unsigned float_bits;
```

你的代码中不使用数据类型 `float`，而要使用 `float_bits`。你可以使用数据类型 `int` 和 `unsigned`，包括无符号和整数常数和运算。你不可以使用任何联合、结构和数组。更重要的是，你不能使用任何浮点数据类型、运算或者常数。取而代之的是，你的代码应该执行实现这些指定的浮点运算的位操作。

下面的函数说明了对这些规则的使用。对于参数 f ，如果 f 是非规格化的，该函数返回 ± 0 （保持 f 的符号），否则，返回 f 。

```
/* If f is denorm, return 0. Otherwise, return f */
float_bits float_denorm_zero(float_bits f) {
```



```
/* Decompose bit representation into parts */
unsigned sign = f>>31;
unsigned exp = f>>23 & 0xFF;
unsigned frac = f      & 0x7FFFFFF;
if (exp == 0) {
    /* Denormalized. Set fraction to 0 */
    frac = 0;
}
/* Reassemble bits */
return (sign << 31) | (exp << 23) | frac;
}
```

*2.91 遵循位级浮点编码规则，实现具有如下原型的函数：

```
/* Compute |f|. If f is NaN, then return f. */
float_bits float_absval(float_bits f);
```

对于浮点数 f ，这个函数计算 $|f|$ 。如果 f 是 NaN ，你的函数应该简单地返回 f 。

测试你的函数，对参数 f 可以取的所有 2^{32} 个值求值，将结果与你使用机器的浮点运算得到的结果相比较。

**2.92 遵循位级浮点编码规则，实现具有如下原型的函数：

```
/* Compute -f. If f is NaN, then return f. */
float_bits float_negate(float_bits f);
```

对于浮点数 f ，这个函数计算 $|f|$ 。如果 f 是 NaN ，你的函数应该简单地返回 f 。

测试你的函数，对参数 f 可以取的所有 2^{32} 个值求值，将结果与你使用机器的浮点运算得到的结果相比较。

**2.93 遵循位级浮点编码规则，实现具有如下原型的函数：

```
/* Compute 0.5*f. If f is NaN, then return f. */
float_bits float_half(float_bits f);
```

对于浮点数 f ，这个函数计算 $0.5 \cdot f$ 。如果 f 是 NaN ，你的函数应该简单地返回 f 。

测试你的函数，对参数 f 可以取的所有 2^{32} 个值求值，将结果与你使用机器的浮点运算得到的结果相比较。

**2.94 遵循位级浮点编码规则，实现具有如下原型的函数：

```
/* Compute 2*f. If f is NaN, then return f. */
float_bits float_twice(float_bits f);
```

对于浮点数 f ，这个函数计算 $2.0 \cdot f$ 。如果 f 是 NaN ，你的函数应该简单地返回 f 。

测试你的函数，对参数 f 可以取的所有 2^{32} 个值求值，将结果与你使用机器的浮点运算得到的结果相比较。

**2.95 遵循位级浮点编码规则，实现具有如下原型的函数：

```
/* Compute (float) i */
float_bits float_i2f(int i);
```

对于函数 i ，这个函数计算 $(\text{float}) i$ 的位级表示。

测试你的函数，对参数 f 可以取的所有 2^{32} 个值求值，将结果与你使用机器的浮点运算得到的结果相比较。



**2.96 遵循位级浮点编码规则，实现具有如下原型的函数：

```
/*
 * Compute (int) f.
 * If conversion causes overflow or f is NaN, return 0x80000000
 */
int float_f2i(float_bits f);
```

对于浮点数 f ，这个函数计算 $(int) f$ 。如果 f 是 Nan ，你的函数应该向零舍入。如果 f 不能用整数表示（例如，超出表示范围，或者它是一个 Nan ），那么函数应该返回 $0x80000000$ 。

测试你的函数，对参数 f 可以取的所有 2^{32} 个值求值，将结果与你使用机器的浮点运算得到的结果相比较。

练习题答案

练习题 2.1 在我们开始查看机器级程序的时候，理解十六进制和二进制格式之间的关系将是很重要的。虽然本书中介绍了完成这些转换的方法，但是做点练习能够让你的转换更加熟练。

A. 将 $0x39A7F8$ 转换成二进制：

十六进制	3	9	A	7	F	8
二进制	0011	1001	1010	0111	1111	1000

B. 将二进制 1100100101111011 转换成十六进制：

二进制	1100	1001	0111	1011
十六进制	C	9	7	B

C. 将 $0xD5E4C$ 转换成二进制：

十六进制	D	5	E	4	C
二进制	1101	0101	1110	0100	1100

D. 将二进制 1001101110011110110101 转换成十六进制：

二进制	10	0110	1110	0111	1011	0101
十六进制	2	6	E	7	B	5

练习题 2.2 这个问题给你一个机会思考 2 的幂和它们的十六进制表示。

n	2^n (十进制)	2^n (十六进制)
9	512	0x200
19	524 288	0x80000
14	16 384	0x4000
16	65 536	0x10000
17	131 072	0x20000
5	32	0x20
7	128	0x80

练习题 2.3 这个问题给你一个机会试着对一些小的数在十六进制和十进制表示之间进行转换。对于较大的数，使用计算器或者转换程序会更加方便和可靠一些。

十进制	二进制	十六进制
0	0000 0000	0x00
$167 = 10 \cdot 16 + 7$	1010 0111	0xA7
$62 = 3 \cdot 16 + 14$	0011 1110	0x3E
$188 = 11 \cdot 16 + 12$	1011 1100	0xBC



(续)

十进制	二进制	十六进制
$3 \cdot 16 + 7 = 55$	0011 0111	0x37
$8 \cdot 16 + 8 = 136$	1000 1000	0x88
$15 \cdot 16 + 3 = 243$	1111 0011	0xF3
$5 \cdot 16 + 2 = 82$	0101 0010	0x52
$10 \cdot 16 + 12 = 172$	1010 1100	0xAC
$14 \cdot 16 + 7 = 231$	1110 0111	0xE7

练习题 2.4 当开始调试机器级程序时，你将发现在许多情况下，一些简单的十六进制运算是很有用的。可以总是把数转换成十进制，完成运算，再把它们转换回来，但是能够直接用十六进制工作更加有效，而且能够提供更多的信息。

- A. $0x503c + 0x8 = 0x5044$ 。8 加上十六进制 c 得到 4 并且进位 1。
- B. $0x503c - 0x40 = 0x4ffc$ 。在第二个数位，3 减去 4 要从第三位借 1。因为第三位是 0，所以我们必须从第四位借位。
- C. $0x503c + 64 = 0x507c$ 。十进制 64 (2^6) 等于十六进制 0x40。
- D. $0x50ea - 0x503c = 0xae$ 。十六进制数 a (十进制 10) 减去十六进制数 c (十进制 12)，我们从第二位借 16，得到十六进制数 e (十进制数 14)。在第二个数位，我们现在用十六进制 d (十进制 13) 减去 3，得到十六进制 a (十进制 10)。

练习题 2.5 这个练习测试你对数据的字节表示和两种不同字节顺序的理解。

小端法：21

大端法：87

小端法：21 43

大端法：87 65

小端法：21 43 65

大端法：87 65 43

回想一下，`show_bytes` 列举了一系列字节，从低位地址的字节开始，然后逐一列出高位地址的字节。在小端法机器上，它将按照从最低有效字节到最高有效字节的顺序列出字节。在大端法机器上，它将按照从最高有效字节到最低有效字节的顺序列出字节。

练习题 2.6 这又是一个练习从十六进制到二进制转换的机会。同时也让你思考整数和浮点表示。我们将在本章后面更加详细地研究这些表示。

- A. 利用书中示例的符号，我们将两个串写成：

0	0	3	5	9	1	4	1
00000000001101011001000101000001							

4	A	5	6	4	5	0	4
01001010010101100100010100000100							

- B. 将第二个字相对于第一个字向右移动 2 位，我们发现一个有 21 个匹配位的序列。

C. 我们发现除了最高有效位 1，整数的所有位都嵌在浮点数中。这正好也是书中示例的情况。另外，浮点数有一些非零的高位不与整数中的高位相匹配。

练习题 2.7 它打印 61 62 63 64 65 66。回想一下，库函数 `strlen` 不计算终止的空字符，所以 `show_bytes` 只打印到字符 ‘f’。

练习题 2.8 这是一个帮助你更加熟悉布尔运算的练习。

运算	结果	运算	结果
a	[01101001]	$a \& b$	[01000001]
b	[01010101]	$a b$	[01111101]
$\sim a$	[10010110]	$a ^ b$	[00111100]
$\sim b$	[10101010]		



练习题 2.9 这个问题说明了怎样用布尔代数描述和解释现实世界的系统。我们能够看到这个颜色代数和长度为 3 的位向量上的布尔代数是一样的。

A. 颜色的取补是通过对 R 、 G 和 B 的值取补得到的。由此我们可以看出，白色是黑色的补，黄色是蓝色的补，红紫色是绿色的补，蓝绿色是红色的补。

B. 我们基于颜色的位向量表示来进行布尔运算。据此，我们得到以下结果：

$$\text{蓝色 (001)} \quad | \quad \text{绿色 (010)} = \text{蓝绿色 (011)}$$

$$\text{黄色 (110)} \quad \& \quad \text{蓝绿色 (011)} = \text{绿色 (010)}$$

$$\text{红色 (100)} \quad ^\wedge \quad \text{红紫色 (101)} = \text{蓝色 (001)}$$

练习题 2.10 这个程序依赖两个事实，EXCLUSIVE-OR 是可交换的和可结合的，以及对于任意的 a ，有 $a^\wedge a=0$ 。

步骤	$*x$	$*y$
初始	a	b
步骤 1	a	$a^\wedge b$
步骤 2	$a^\wedge (a^\wedge b) = (a^\wedge a)^\wedge b = b$	$a^\wedge b$
步骤 3	b	$b^\wedge (a^\wedge b) = (b^\wedge b)^\wedge a = a$

某种情况下这个函数会失败，参见练习题 2.11。

练习题 2.11 这个题目说明了我们的原地交换规程微妙而有趣的特性。

A. `first` 和 `last` 的值都为 `k`，所以我们试图交换正中间的元素和它自己。

B. 在这种情况下，`inplace_swap` 的参数 `x` 和 `y` 都指向同一个位置。当计算 $*x^\wedge *y$ 的时候，我们得到 0。然后将 0 作为数组正中间的元素，且后面的步骤一直都把这个元素设置为 0。我们可以看到，练习题 2.10 的推理隐含地假设 `x` 和 `y` 代表不同的位置。

C. 将 `reverse_array` 的第 4 行的测试简单地替换成 `first < last`，因为没有必要交换正中间的元素和它自己。

练习题 2.12 这些表达式如下：

A. $x \& ?0xFF$

B. $x \wedge \sim 0xFF$

C. $x \mid 0xFF$

这些表达式是在执行低级位运算中经常发现的典型类型。表达式 $\sim 0xFF$ 创建一个掩码，该掩码 8 个最低位等于 0，而其余的位为 1。可以观察到，这些掩码的产生和字长无关。而相比之下，表达式 `0xFFFFFFFF00` 只能在 32 位的机器上工作。

练习题 2.13 这个问题有助于你思考布尔运算和程序员应用掩码运算的典型方式之间的关系。代码如下：

```
/* Declarations of functions implementing operations bis and bic */
int bis(int x, int m);
int bic(int x, int m);

/* Compute x|y using only calls to functions bis and bic */
int bool_or(int x, int y) {
    int result = bis(x,y);
    return result;
}

/* Compute x^y using only calls to functions bis and bic */
int bool_xor(int x, int y) {
    int result = bis(bic(x,y), bic(y,x));
    return result;
}
```



`bis` 运算等价于布尔 OR——如果 `x` 或者 `m` 中的这一位置位了，那么 `z` 中的这一位就置位。另一方面，`bic(x, m)` 等价于 `x&~m`；我们想实现只有当 `x` 对应的位为 1 且 `m` 对应的位为 0 时，该位等于 1。

由此，可以通过对 `bis` 的一次调用来实现 `|`。为了实现 `^`，我们利用以下属性：

$$x \wedge y = (x \& \neg y) \mid (\neg x \& y)$$

练习题 2.14 这个问题突出了位级布尔运算和 C 语言中的逻辑运算之间的关系。常见的编程错误是在想用逻辑运算的时候用了位级运算，或者反过来。

表达式	值	表达式	值
<code>x & y</code>	0x20	<code>x && y</code>	0x01
<code>x y</code>	0x7F	<code>x y</code>	0x01
<code>\~x \~y</code>	0xDF	<code>\!x \!y</code>	0x00
<code>x & \!y</code>	0x00	<code>x && \~y</code>	0x01

练习题 2.15 这个表达式是 `!(x^y)`。

也就是，当且仅当 `x` 的每一位和 `y` 相应的每一位匹配时，`x^y` 等于零。然后，我们利用 `!` 来判定一个字是否包含任何非零位。

没有任何实际的理由要去使用这个表达式，因为可以简单地写成 `x==y`，但是它说明了位级运算和逻辑运算之间的一些细微差别。

练习题 2.16 这个练习可以帮助你理解不同移位运算。

x	x << 3		(逻辑) x >> 2		(算术) x >> 2		
	十六进制	二进制	二进制	十六进制	二进制	十六进制	
0xC3	[11000011]	[00011000]	0x18	[00110000]	0x30	[11110000]	0xF0
0x75	[01110101]	[10101000]	0xA8	[00011101]	0x1D	[00011101]	0x1D
0x87	[10000111]	[00111000]	0x38	[00100001]	0x21	[11100001]	0xE1
0x66	[01100110]	[00110000]	0x30	[00011001]	0x19	[00011001]	0x19

练习题 2.17 一般而言，研究字长非常小的例子是理解计算机运算的非常好的方法。

无符号值对应于图 2-2 中的值。对于补码值，十六进制数字 0 ~ 7 的最高有效位为 0，得到非负值，然而十六进制数字 8 ~ F 的最高有效位为 1，得到一个为负的值。

\vec{x}	$B2U_4(\vec{x})$		$B2T_4(\vec{x})$
	十六进制	二进制	
0xE	[1110]	$2^3 + 2^2 + 2^1 = 14$	$-2^3 + 2^2 + 2^1 = -2$
0x0	[0000]	0	0
0x5	[0101]	$2^2 + 2^0 = 5$	$2^2 + 2^0 = 5$
0x8	[1000]	$2^3 = 8$	$-2^3 = -8$
0xD	[1101]	$2^3 + 2^2 + 2^0 = 13$	$-2^3 + 2^2 + 2^0 = -3$
0xF	[1111]	$2^3 + 2^2 + 2^1 + 2^0 = 15$	$-2^3 + 2^2 + 2^1 + 2^0 = -1$

练习题 2.18 对于 32 位的机器，由 8 个十六进制数字组成，而且开始的那个数字在 `f` 之间的任何值，都是一个负数。数字以 `f` 串开头是很普遍的事情，因为负数的起始位全为 1。不过，你必须看仔细了。例如，数 `0x8048337` 仅有 7 个数字。把起始位填入 0，从而得到 `0x08048337`，这是一个正数。



8048337:	81 ec b8 01 00 00	sub	\$0x1b8,%esp	A.	440
804833d:	8b 55 08	mov	0x8(%ebp),%edx		
8048340:	83 c2 14	add	\$0x14,%edx	B.	20
8048343:	8b 85 58 fe ff ff	mov	0xfffffe58(%ebp),%eax	C.	-424
8048349:	03 02	add	(%edx),%eax		
804834b:	89 85 74 fe ff ff	mov	%eax,0xfffffe74(%ebp)	D.	-396
8048351:	8b 55 08	mov	0x8(%ebp),%edx		
8048354:	83 c2 44	add	\$0x44,%edx	E.	68
8048357:	8b 85 c8 fe ff ff	mov	0xfffffec8(%ebp),%eax	F.	-312
804835d:	89 02	mov	%eax,(%edx)		
804835f:	8b 45 10	mov	0x10(%ebp),%eax	G.	16
8048362:	03 45 0c	add	0xc(%ebp),%eax	H.	12
8048365:	89 85 ec fe ff ff	mov	%eax,0xfffffec(%ebp)	I.	-276
804836b:	8b 45 08	mov	0x8(%ebp),%eax		
804836e:	83 c0 20	add	\$0x20,%eax	J.	32
8048371:	8b 00	mov	(%eax),%eax		

练习题 2.19 从数学的视角来看，函数 $T2U$ 和 $U2T$ 是非常奇特的。理解它们的行为非常重要。

我们根据补码的值解答这个问题，重新排列练习题 2.17 答案中的行，然后列出无符号值作为函数应用的结果。我们展示十六进制值，以使这个进程更加具体。

\vec{x} (十六进制)	x	$T2U_4(x)$
0x8	-8	8
0xD	-3	13
0xE	-2	14
0xF	-1	15
0x0	0	0
0x5	5	5

练习题 2.20 这个练习题测试你对等式 (2-6) 的理解。

对于前 4 个条目， x 的值是负的，并且 $T2U_4(x) = x + 2^4$ 。对于剩下的两个条目， x 的值是非负的，并且 $T2U_4(x) = x$ 。

练习题 2.21 这个问题加强你对补码和无符号表示之间关系的理解，以及对 C 语言升级规则 (promotion rule) 的影响的理解。回想一下， $TMin_{32}$ 是 $-2^{147\,483\,648}$ ，并且将它强制类型转换为无符号数后，变成了 $2^{147\,483\,648}$ 。另外，如果有任何一个运算数是无符号的，那么在比较之前，另一个运算数会被强制类型转换为无符号数。

表达式	类型	求值
$-2147483647 - 1 == 2147483648U$	无符号数	1
$-2147483647 - 1 < 2147483647$	有符号数	1
$-2147483647 - 1U < 2147483647$	无符号数	0
$-2147483647 - 1 < -2147483647$	有符号数	1
$-2147483647 - 1U < -2147483647$	无符号数	1

练习题 2.22 这个练习很具体地说明了符号扩展如何保持一个补码表示的数值。

- A. [1011]: $-2^3 + 2^1 + 2^0 = -8 + 2 + 1 = -5$
B. [11011]: $-2^4 + 2^3 + 2^1 + 2^0 = -16 + 8 + 2 + 1 = -5$
C. [111011]: $-2^5 + 2^4 + 2^3 + 2^1 + 2^0 = -32 + 16 + 8 + 2 + 1 = -5$

练习题 2.23 这些函数的表达式是常见的程序“习惯用语”，可以从多个位域打包成的一个字中提取值。



它们利用不同移位运算的零填充和符号扩展的属性。请注意强制类型转换和移位运算的顺序。在 fun1 中，移位是在无符号 word 上进行的，因此是逻辑移位。在 fun2 中，移位是在把 word 强制类型转换为 int 之后进行的，因此是算术移位。

w	fun1(w)	fun2(w)
0x00000076	0x00000076	0x00000076
0x87654321	0x00000021	0x00000021
0x000000C9	0x000000C9	0xFFFFFFF9
0xEDCBA987	0x00000087	0xFFFFFFF7

B. 函数 fun1 从参数的低 8 位中提取一个值，得到范围 0 ~ 255 之间的一个整数。函数 fun2 也从这个参数的低 8 位中提取一个值，但是它还要执行符号扩展。结果将是介于 -128 ~ 127 之间的一个数。

练习题 2.24 对于无符号数来说，截断的影响是相当直观的，但是对于补码数却不是。这个练习让你使用非常小的字长来研究它的属性。

十六进制		无符号		补码	
原始数	截断后的数	原始数	截断后的数	原始数	截断后的数
0	0	0	0	0	0
2	2	2	2	2	2
9	1	9	1	-7	1
B	3	11	3	-5	3
F	7	15	7	-1	-1

正如等式 (2-9) 所描述的，这种截断无符号数值的结果就是发现它们模 8 余数。截断有符号数的结果要更复杂一些。根据等式 (2-10)，我们首先计算这个参数模 8 后的余数。对于参数 0 ~ 7，将得出值 0 ~ 7，对于参数 -8 ~ -1 也是一样。然后我们对这些余数应用函数 U2T_i，得出两个 0 ~ 3 和 -4 ~ 1 序列的反复。

练习题 2.25 设计这个问题是要说明从有符号数到无符号数的隐式强制类型转换很容易引起错误。将参数 length 作为一个无符号数来传递看上去是件相当自然的事情，因为没有人会想到使用一个长度为负数的值。停止条件 i <= length - 1 看上去也很自然。但是把这两点组合到一起，将产生意想不到的结果！

因为参数 length 是无符号的，计算 0 - 1 将进行无符号运算，这等价于模数加法。结果得到 UMax。

比较进行同样使用无符号数比较，而因为任何数都是小于或者等于 UMax 的，所以这个比较总是为真！因此，代码将试图访问数组 a 的非法元素。

有两种方法可以改正这段代码，其一是将 length 声明为 int 类型，其二是将 for 循环的测试条件改为 i < length。

练习题 2.26 这个例子说明了无符号运算的一个细微的特性，同时也是我们执行无符号运算时不会意识到的属性。这会导致一些非常棘手的错误。

A. 在什么情况下，这个函数会产生不正确的结果？当 s 比 t 短的时候，该函数会不正确地返回 1。

B. 解释为什么会出现这样不正确的结果。由于 strlen 被定义为产生一个无符号的结果，差和比较都采用无符号运算来计算。当 s 比 t 短的时候，差 strlen(s) - strlen(t) 会为负，但是变成了一个很大的无符号数，且大于 0。

C. 说明如何修改这段代码好让它能可靠地工作。将测试语句改成：

```
return strlen(s) > strlen(t);
```

练习题 2.27 这个函数是对确定无符号加法是否溢出的规则的直接实现。



```
/* Determine whether arguments can be added without overflow */
int uadd_ok(unsigned x, unsigned y) {
    unsigned sum = x+y;
    return sum >= x;
}
```

练习题 2.28 本题是对算术模 16 的简单示范。最容易的解决方法是将十六进制模式转换成它的无符号十进制值。对于非零的 x 值，一定有 $(-\frac{u}{4}x) + x = 16$ 。然后，我们就可以将取补后的值转换回十六进制。

x		$-\frac{u}{4}x$	
十六进制	十进制	十进制	十六进制
0	0	0	0
5	5	11	B
8	8	8	8
D	13	3	3
F	15	1	1

练习题 2.29 本题的目的是确定你理解了补码加法。

x	y	$x + y$	$x +_5 y$	情况
-12 [10100]	-15 [10001]	-27 [100101]	5 [00101]	1
-8 [11000]	-8 [11000]	-16 [110000]	-16 [10000]	2
-9 [10111]	8 [01000]	-1 [111111]	-1 [11111]	2
2 [00010]	5 [00101]	7 [000111]	7 [00111]	3
12 [01100]	4 [00100]	16 [010000]	-16 [10000]	4

练习题 2.30 这个函数是对确定补码加法是否溢出的规则的直接实现。

```
/* Determine whether arguments can be added without overflow */
int tadd_ok(int x, int y) {
    int sum = x+y;
    int neg_over = x < 0 && y < 0 && sum >= 0;
    int pos_over = x >= 0 && y >= 0 && sum < 0;
    return !neg_over && !pos_over;
}
```

练习题 2.31 通过对 2.3.2 节的学习，你的同事可能已经学会补码加上形成一个阿贝尔群，以及表达式 $(x+y)-x$ 求值得到 y ，无论加法是否溢出，而 $(x+y)-y$ 总是会求值得到 x 。

练习题 2.32 这个函数会给出正确的值，除了当 y 等于 $TMin$ 时。在这个情况下，我们有 $-y$ 也等于 $TMin$ ，因此函数 `tadd_ok` 会认为只要 x 是负数时，就会负溢出。实际上，此时 $x-y$ 根本就没有溢出。

这个练习说明，在函数的任何测试过程中， $TMin$ 都应该作为一种测试情况。

练习题 2.33 本题用非常小的字长帮助你理解补码的非。

对于 $w=4$ ，我们有 $TMin_4=-8$ 。因此 -8 是它自己的加法逆元，而其他数值是通过整数非来取非的。



x		$\frac{-1}{4}x$	
十六进制	十进制	十进制	十六进制
0	0	0	0
5	5	-5	B
8	-8	-8	8
D	-3	3	3
F	-1	1	1

对于无符号数的非，位的模式是相同的。

练习题 2.34 本题的目的是确定你理解了补码乘法。

模式	x	y	$x \cdot y$	截断了的 $x \cdot y$
无符号数 补码	4 [100]	5 [101]	20 [010100]	4 [100]
	-4 [100]	-3 [101]	12 [001100]	-4 [100]
无符号数 补码	2 [010]	7 [111]	14 [001110]	6 [110]
	2 [010]	-1 [111]	-2 [111110]	-2 [110]
无符号数 补码	6 [110]	6 [110]	36 [100100]	4 [100]
	-2 [110]	-2 [110]	4 [000100]	-4 [100]

练习题 2.35 用所有可能的 x 和 y 测试一遍这个函数显然是不现实的。当数据类型 int 为 32 位时，即使你每秒运行一百亿个测试，也需要 58 年才能完成所有的组合。另一方面，把函数中的数据类型改成 short 或者 char，然后再穷尽测试，倒是测试代码的一种可行的方法。

我们提出以下论据，这是一个更理论的方法：

1. 我们知道 $x \cdot y$ 可以写成一个 $2w$ 位的补码数字。用 u 表示低 w 位的无符号数， v 表示高 w 位的补码数字。那么，根据等式 (2-3)，我们可以得到 $x \cdot y = v2^w + u$ 。

我们还知道 $u = T2U_w(p)$ ，因为它们是从同一个位模式得出来的无符号和补码数字，因此根据等式 (2-5)，我们有 $u = p + p_{w-1}2^w$ ，这里 p_{w-1} 是 p 的最高有效位。设 $t = v + p_{w-1}$ ，我们得到 $x \cdot y = p + t2^w$ 。

当 $t = 0$ 时，有 $x \cdot y = p$ ；乘法不会溢出。当 $t \neq 0$ 时，有 $x \cdot y \neq p$ ；乘法溢出。

2. 根据整数除法的定义，用非零数 x 除以 p 会得到商 q 和余数 r ，即 $p = x \cdot q + r$ ，且 $|r| < |x|$ 。（这里用的是绝对值，因为 x 和 r 的符号可能不一致。例如，-7 除以 2 得到商 -3 和余数 -1。）

3. 假设 $q = y$ ，那么有 $x \cdot y = x \cdot y + r + t2^w$ 。在此，我们可以得到 $r + t2^w = 0$ 。但是 $|r| < |x| - 2^w$ ，所以只有当 $t = 0$ 时，这个等式才会成立，此时 $r = 0$ 。

假设 $r = t = 0$ ，那么我们有 $x \cdot y = x \cdot q$ ，隐含有 $y = q$ 。

当 $x = 0$ 时，乘法不溢出，所以我们的代码提供了一种可靠的方法来测试补码乘法是否会导致溢出。

练习题 2.36 如果用 64 位表示，乘法就不会有溢出。然后我们来验证将乘积强制类型转换为 32 位是否会改变它的值：

```
1 /* Determine whether arguments can be multiplied without overflow */
2 int tmult_ok(int x, int y) {
3     /* Compute product without overflow */
4     long long pll = (long long) x*y;
5     /* See if casting to int preserves value */
6     return pll == (int) pll;
7 }
```

注意，第 4 行右边的强制类型转换至关重要。如果我们将这一行写成以下形式：



```
long long pll = x*y;
```

就会用 32 位值来计算乘积（可能会溢出），然后再符号扩展到 64 位。

练习题 2.37

A. 这个改动完全没有帮助。虽然 `asize` 的计算会更准确，但是调用 `malloc` 会导致这个值被转换成一个 32 位无符号数字，因而还是会出现同样的溢出条件。

B. `malloc` 使用一个 32 位无符号数作为参数，它不可能分配一个大于 2^{32} 个字节的块，因此，没有必要试图去分配或者复印这样大的一块存储器。取而代之的，函数应该放弃，返回 `NULL`，用下面的代码取代对 `malloc` 原始的调用（第 10 行）：

```
long long unsigned required_size =
    ele_cnt * (long long unsigned) ele_size;
size_t request_size = (size_t) required_size;
if (required_size != request_size)
    /* Overflow must have occurred. Abort operation */
    return NULL;
void *result = malloc(request_size);
if (result == NULL)
    /* malloc failed */
    return NULL;
```

练习题 2.38 在第 3 章，我们将看到很多实际的 LEA 指令的例子。用这个指令来支持指针运算，但是 C 语言编译器经常用它来执行小常数乘法。

对于每个 k 的值，可以计算出 2^k （当 b 为 0 时）和 $2^k + 1$ （当 b 为 a 时）。因此我们能够计算出倍数为 1, 2, 3, 4, 5, 8 和 9 的值。

练习题 2.39 这个表达式简单地变成了 $-(x \ll m)$ 。要看清一点，设字长为 w , $n = w - 1$ 。形式 B 说我们要计算 $(x \ll w) - (x \ll m)$ ，但是将 x 向左移动 w 位会得到值 0。

练习题 2.40 这个题目要求你使用讲过的优化技术，同时也需要自己的一点儿创造力。

K	移位	加法/减法	表达式
6	2	1	$(x \ll 2) + (x \ll 1)$
31	1	1	$(x \ll 5) - x$
-6	2	1	$(x \ll 1) - (x \ll 3)$
55	2	2	$(x \ll 6) - (x \ll 3) - x$

可以观察到，第四种情况使用了形式 B 的改进版本。我们可以将位模式 [110111] 看作 6 个连续的 1 中间有一个 0，因而我们对形式 B 应用这个原则，但是需要在后来把中间 0 位对应的项减掉。

练习题 2.41 假设加法和减法有同样的性能，那么原则就是当 $n=m$ 时，选择形式 A，当 $n=m+1$ 时，随便选哪种，而当 $n>m+1$ 时，选择形式 B。

这个原则的证明如下。首先假设 $m>1$ 。当 $n=m$ 时，形式 A 只需要 1 个移位，而形式 B 需要 2 个移位和 1 个减法。当 $n=m+1$ 时，这两种形式都需要 2 个移位和 1 个加法或者 1 个减法。当 $n>m+1$ 时，形式 B 只需要 2 个移位和 1 个减法，而形式 A 需要 $n-m+1>2$ 个移位和 $n-m>1$ 个加法。对于 $m=1$ 的情况，对于形式 A 和 B 都要少 1 个移位，所以在两者中选择时，还是适用同样的原则。

练习题 2.42 这里唯一的挑战是不用任何测试或条件运算计算偏置量。我们利用了一个诀窍，表达式 $x >> 31$ 产生一个字，如果 x 是负数，这个字为全 1，否则为全 0。通过掩码屏蔽适当的位，我们就得到期望的偏置值。



```
int div16(int x) {
    /* Compute bias to be either 0 (x >= 0) or 15 (x < 0) */
    int bias = (x >> 31) & 0xF;
    return (x + bias) >> 4;
}
```

练习题 2.43 我们发现当人们直接与汇编代码打交道时是有困难的。但当把它放入 optarith 所示的形式中时，问题会变得更加清晰明了。

我们可以看到 M 是 31；是用 $(x \ll 5) - x$ 来计算 $x \times M$ 。

我们可以看到 N 是 8；当 y 是负数时，加上偏置量 7，并且右移 3 位。

练习题 2.44 这些“C 的谜题”清楚地告诉程序员必须理解计算机运算的属性。

A. $(x > 0) \mid\mid ((x - 1) < 0)$

假。设 x 等于 $-2^{147}483\,648$ ($TMin_{32}$)。那么，有 $x - 1$ 等于 $2^{147}483\,647$ ($TMax_{32}$)。

B. $(x \& 7) != 7 \mid\mid (x \ll 29 < 0)$

真。如果 $(x \& 7) != 7$ 这个表达式的值为 0，那么必须有位 x_2 等于 1。当左移 29 位时，这个位将变成符号位。

C. $(x * x) >= 0$

假。当 x 为 $65\,535$ ($0xFFFF$) 时， $x * x$ 为 $-131\,071$ ($0xFFE0001$)。

D. $x < 0 \mid\mid -x \leq 0$

真。如果 x 是非负数，则 $-x$ 是非正的。

E. $x > 0 \mid\mid -x > 0$

假。设 x 为 $-2^{147}483\,648$ ($TMin_{32}$)。那么 x 和 $-x$ 都为负数。

F. $x + y == uy + ux$

真。补码和无符号乘法有相同的位级行为，而且它们是可交换的。

G. $x^{\sim}y + uy^{\sim}ux == -x$

真。 $\sim y$ 等于 $-y - 1$ 。 $uy^{\sim}ux$ 等于 $x^{\sim}y$ 。因此，等式左边等价于 $x^{\sim} - y - x + x^{\sim}y$ 。

练习题 2.45 理解二进制小数表示是理解浮点编码的一个重要步骤。这个练习让你试验一些简单的例子。

小数值	二进制表示	十进制表示
$\frac{1}{8}$	0.001	0.125
$\frac{3}{4}$	0.11	0.75
$\frac{25}{16}$	1.1001	1.5625
$\frac{43}{16}$	10.1011	2.6875
$\frac{9}{8}$	1.001	1.125
$\frac{47}{8}$	101.111	5.875
$\frac{51}{16}$	11.0011	3.1875

考虑二进制小数表示的一个简单方法是将一个数表示为形如 $\frac{x}{2^k}$ 的小数。我们将这个形式表示为二进制的过程是：使用 x 的二进制表示，并把二进制小数点插入从右边算起的第 k 个位置。举一个例子，对于 $\frac{25}{16}$ ，有 $25_{10} = 11001_2$ 。然后把二进制小数点放在从右算起的第 4 位，得到 1.1001_2 。

练习题 2.46 在大多数情况下，浮点数的有限精度不是主要的问题，因为计算的相对误差仍然是相当低的。然而在这个例子中，系统对于绝对误差是很敏感的。

A. 我们可以看到 $0.1 - x$ 的二进制表示为：

$0.000000000000000000000000000000001100[1100] \dots_2$

把这个表示与 $\frac{1}{10}$ 的二进制表示进行比较，我们可以看到这就是 $2^{-20} \times \frac{1}{10}$ ，也就是大约 9.54×10^{-8} 。



B. $9.54 \times 10^{-8} \times 100 \times 60 \times 60 \times 10 \approx 0.343$ 秒。

C. $0.343 \times 2000 \approx 687$ 米。

练习题 2.47 研究字长非常小的浮点表示能够帮助澄清 IEEE 浮点是怎样工作的。要特别注意非规格化数和规格化数之间的过渡。

位	e	E	2^E	f	M	$2^E \times M$	V	十进制
0 0 0 0 0	0	0	1	$\frac{0}{4}$	$\frac{0}{4}$	$\frac{0}{4}$	0	0.0
0 0 0 0 1	0	0	1	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	0.25
0 0 0 1 0	0	0	1	$\frac{2}{4}$	$\frac{2}{4}$	$\frac{2}{4}$	$\frac{1}{2}$	0.5
0 0 0 1 1	0	0	1	$\frac{3}{4}$	$\frac{3}{4}$	$\frac{3}{4}$	$\frac{3}{4}$	0.75
0 0 1 0 0	1	0	1	$\frac{0}{4}$	$\frac{4}{4}$	$\frac{4}{4}$	1	1.0
0 0 1 0 1	1	0	1	$\frac{1}{4}$	$\frac{5}{4}$	$\frac{5}{4}$	$\frac{5}{4}$	1.25
0 0 1 1 0	1	0	1	$\frac{2}{4}$	$\frac{6}{4}$	$\frac{6}{4}$	$\frac{3}{2}$	1.5
0 0 1 1 1	1	0	1	$\frac{3}{4}$	$\frac{7}{4}$	$\frac{7}{4}$	$\frac{7}{4}$	1.75
0 1 0 0 0	2	1	2	$\frac{0}{4}$	$\frac{4}{4}$	$\frac{8}{4}$	2	2.0
0 1 0 0 1	2	1	2	$\frac{1}{4}$	$\frac{5}{4}$	$\frac{10}{4}$	$\frac{5}{2}$	2.5
0 1 0 1 0	2	1	2	$\frac{2}{4}$	$\frac{6}{4}$	$\frac{12}{4}$	3	3.0
0 1 0 1 1	2	1	2	$\frac{3}{4}$	$\frac{7}{4}$	$\frac{14}{4}$	$\frac{7}{2}$	3.5
0 1 1 0 0	—	—	—	—	—	—	∞	—
0 1 1 0 1	—	—	—	—	—	—	NaN	—
0 1 1 1 0	—	—	—	—	—	—	NaN	—
0 1 1 1 1	—	—	—	—	—	—	NaN	—

练习题 2.48 十六进制 0x359141 等价于二进制 [1101011001000101000001]。将之右移 21 位得到 1.101011001000101000001₂ × 2¹。除去起始位的 1 并增加 2 个 0 形成小数域，从而得到 [10101100100010100000100]。阶码是通过 21 加上偏置量 127 形成的，得到 148 (二进制 [10010100])。我们把它和符号字段 0 联合起来，得到二进制表示

[01001010010101100100010100000100]

我们看到两种表示中匹配的位对应于整数的低位到最高有效位等于 1，匹配小数的高 21 位：

0	0	3	5	9	1	4	1
00000000001101011001000101000001							

4	A	5	6	4	5	0	4
01001010010101100100010100000100							

练习题 2.49 这个练习帮助你思考什么数不能用浮点准确表示。

A. 这个数的二进制表示是：1 后面跟着 n 个 0，其后再跟 1，得到的值是 $2^{n+1} + 1$ 。

B. 当 n = 23 时，值是 $2^{24} + 1 = 16\ 777\ 217$ 。

练习题 2.50 人工舍入帮助你强化二进制数舍入到偶数的概念。

原始值	舍入后的值
10.010 ₂	$2\frac{1}{4}$
10.011 ₂	$2\frac{3}{8}$
10.110 ₂	$2\frac{3}{4}$
11.001 ₂	$3\frac{1}{8}$



练习题 2.51

A. 从 $1/10$ 的无穷序列中我们可以看到，舍入位置右边 2 位都是 1，所以 $1/10$ 更好一点儿的近似值应该是对 x 加 1，得到 $x' = 0.0001100110011001101_2$ ，它比 0.1 大一点儿。

B. 我们可以看到 $x' - 0.1$ 的二进制表示为：

$0.0000000000000000[1100]$

将这个值与 $1/10$ 的二进制表示比较，我们可以看到它等于 $2^{-22} \times 1/10$ ，大约等于 2.38×10^{-8} 。

C. $2.38 \times 10^{-8} \times 100 \times 60 \times 60 \times 10 \approx 0.086$ 秒，爱国者导弹系统中的误差是它的 4 倍。

D. $0.086 \times 2000 \approx 171$ 米。

练习题 2.52 这个题目考查了很多有关浮点表示的概念，包括规格化和非规格化的值的编码，以及舍入。

格式A		格式B		注
位	值	位	值	
011 0000	1	0111 000	1	
101 1110	$\frac{15}{2}$	1001 111	$\frac{15}{2}$	
010 1001	$\frac{25}{32}$	0110 100	$\frac{3}{4}$	向上舍入
110 1111	$\frac{31}{2}$	1011 000	16	向下舍入
000 0001	$\frac{1}{64}$	0001 000	$\frac{1}{64}$	Denorm \rightarrow norm

练习题 2.53 一般来说，使用库宏（library macro）会比你自己写的代码更好一些。然而这段代码似乎可以在多种机器上工作。

假设值 $1e400$ 溢出为无穷。

```
#define POS_INFINITY 1e400  
#define NEG_INFINITY (-POS_INFINITY)  
#define NEG_ZERO (-1.0/POS_INFINITY)
```

练习题 2.54 这个练习可以帮助你从程序员的角度来提高研究浮点运算的能力。确信自己理解下面每一个答案。

A. $x == (\text{int})(\text{double})x$

真，因为 double 类型比 int 类型具有更大的精度和范围。

B. $x == (\text{int})(\text{float})x$

假，例如当 x 为 $TMax$ 时。

C. $d == (\text{double})(\text{float})d$

假，例如当 d 为 $1e40$ 时，右边得到 $+\infty$ 。

D. $f == (\text{float})(\text{double})f$

真，因为 double 类型比 float 类型具有更大的精度和范围。

E. $f == -(-f)$

真，因为浮点数取非就是简单地对它的符号位取反。

F. $1.0/2 == 1/2.0$

真，在执行除法之前，分子和分母都会被转换成浮点表示。

G. $d * d >= 0.0$

真，虽然它可能会溢出到 $+\infty$ 。

H. $(f + d) - f == d$

假，例如当 f 是 $1.0e20$ 而 d 是 1.0 时，表达式 $f + d$ 会舍入到 $1.0e20$ ，因此左边的表达式求值得到 0.0，而右边是 1.0。