UNIVERSITY OF CALIFORNIA
Department of Electrical Engineering
and Computer Sciences
Computer Science Division

These are the class notes of Brian Harvey—an instructor here at Berkeley who often teaches CS61C. Brian has very interesting views on many of the topics covered in the class, and in particular on the C programming language. His perspective will most certainly help you understand the material in the course, and is likely to spark lively discussions.

CS 61C: C: Introduction, Pointers, & Arrays

Brian Harvey
Edited by J. Wawrzynek
Edited by G. Gibeling
August 24, 2007

1 61C from a 61A perspective

61C is a course in which there are a lot of details (such as the particular instructions and representation formats of the MIPS architecture), and so it's easy to lose the forest in the trees. Here are two things to keep in mind to avoid that:

1.1 Abstraction

Just as in the case of software, computer hardware develops in power by using the idea of abstraction to enable greater complexity and generality.

Abstraction in hardware isn't limited to the idea of pushing some of the work into software. There's abstraction at work within hardware design itself. The most important example is the idea of the *digital domain*. We talk as though every wire in the computer is either on (high voltage) or off (low voltage). To talk more concretely, let's say that high voltage means five volts, and low voltage means zero volts. (These used to be realistic numbers.) But in fact, the voltage at the output of some circuit (an adder, for example, or an AND gate) depends on how many inputs of other circuits are attached to it. Each reader of a signal reduces the voltage a little; if one circuit is attached to an output, it might really be at 4.8 volts; with two circuits reading the value, it might be 4.6 volts. (I'm making up those numbers.) At some point, with enough inputs attached, the voltage will be low enough so that the circuits reading it might interpret it as a low voltage, so a 1 bit might appear as a 0 bit instead. (The number of circuits that can be allowed to read a given circuit's output before this happens is called the maximum "fanout" of the circuit.) In the digital abstraction, we pretend there are no intermediate voltages, and we think about fanout constraints separately.

1.1.1 An Example: Number Representations

As one example, consider the history of number representation. In a few weeks you'll learn about the now-universal convention that integers are represented in computer hardware using twos complement

加州大学电气工程系

和计算机科学计算机科学
部

这些是布莱恩·哈维（Brian Harvey）的课堂笔记，他是伯克利的一名讲师，经常教授CS61C。Brian对课程中涵盖的许多主题都有非常有趣的看法，尤其是 C 编程语言。他的观点肯定会帮助你理解课程中的材料，并可能引发热烈的讨论。

CS 61C： C： 简介， 指针， & 数组

布莱恩·哈维

编辑：J. Wawrzynek 编
辑：G. Gibeling 2007年
8月24日

1 从61A的角度来看61C

61C是一门有很多细节（例如MIPS架构的特定指令和表示格式）的课程，因此很容易失去树木中的森林。为了避免这种情况，请记住以下两件事：

1.1 抽象化

就像软件一样，计算机硬件通过使用抽象的思想来实现更大的复杂性和通用性。

硬件中的抽象并不局限于将一些工作推送到软件中的想法。硬件设计本身存在抽象性。最重要的例子是数字领域的概念。我们说话时好像计算机中的每根电线要么打开（高压），要么关闭（低电压）。更具体地说，假设高压意味着五伏，低电压意味着零伏。（这些曾经是现实的数字。但实际上，某些电路（例如加法器或AND门）输出端的电压取决于连接到其上的其他电路的输入数量。每个信号读取器都会稍微降低电压;如果一个电路连接到输出端，它可能真的是 4.8 伏;当两个电路读取该值时，它可能是 4.6 伏。（这些数字是我编造的。在某些时候，如果连接了足够的输入，电压将足够低，以便读取它的电路可能会将其解释为低电压，因此 1 位可能会显示为 0 位。（在此之前，可以允许读取给定电路输出的电路数称为电路的最大“扇出”。在数字抽象中，我们假设没有中间电压，并分别考虑扇出约束。

1.1.1 示例：数字表示

举个例子，考虑数字表示的历史。几周后，您将了解现在通用的约定，即整数在计算机硬件中使用二进制补码表示

binary (in which each bit represents a power of two). But some earlier computers used a different representation, called BCD (Binary Coded Decimal). In this system, each decimal digit of a human-readable numeral is directly encoded using four bits.

[Digression: Why four bits? With N bits we can represent 2^N distinct values. Three bits can therefore represent $23 = 8$ values. That would be enough, for example, if we wanted to encode a day of the week (Monday is 000, Tuesday is 001, etc.); but it's not enough for the ten decimal digits.]

So, for example, the integer 2479 would be represented as four groups of four bits each:

```
0010 0100 0111 1001
```

Each group of four is a small unsigned binary integer.

What was the point of this representation? As you can imagine, the circuitry to perform arithmetic operations on BCD numbers is much more complicated than the circuitry used for binary arithmetic. BCD computers existed at the same time as other computers that did use binary, so it's not that the designers didn't know of any better alternative.

The reason for BCD was that in those days, manufacturers sold two different kinds of computer to two different markets. There were *scientific* computers, which used binary, and *business* computers, which used BCD. Scientific computers, like all modern computers, had binary integers and a floating-point notation (which we'll study later) for non-integer values. Business computers used a direct encoding of the format in which business people represent numbers. Since four bits can encode 16 values, and there instructions. So I really should have said

```
0010 0100 0111 1001 1111 0001 0000
```

supposing that 1111 represents the decimal point. Some machines of this time had a FORMAT instruction, implemented entirely in hardware, that would convert such a BCD number into a formatted ASCII text string including programmer-specified extra characters, so this number might appear as `$2,479.10` or, to fill up the available space when writing a check, as `$**********2,479.10`.

The IBM System/360, introduced with great fanfare in the mid-1960s as a universal computer, suitable for both business and scientific computing, was "universal" by virtue of including hardware support for both binary and BCD arithmetic. (The "360" in its name was meant to suggest that it covered a complete circle of computing applications.)

Today, of course, both business and scientific applications are supported on computers that represent numbers in binary form. Business users don't have to read binary (not that scientists do either!); the numbers are converted to and from human-readable decimal format in software. In this example, the general principle of abstraction takes the specific form of moving a task from hardware (a low level of abstraction) to software (a higher level). We'll see that the RISC (Reduced Instruction Set Computing) approach to computer design depends on many instances of this same idea.

I'm afraid I've told this story in a way that makes the early computer designers look like idiots. There's more complexity to the story. For example, it turns out that binary floating-point representation, used universally today for non-integer values, can't exactly represent the fraction 1/10, and therefore can't exactly represent the value 2479.10. This is one reason why business users didn't like the idea of binary computers; they were afraid of roundoff errors. If an error of less than a dollar seems trivial, consider that a few years ago a programmer who worked for a bank went to prison for writing the program that computes the interest on savings accounts so that it always gave the customer the correct interest, rounded *down* to an integer number of cents; it then added up the leftover fractions of a cent from all the accounts and deposited that sum into the programmer's account. The programmer reasoned that the bank was paying out exactly what it should have, so it wasn't being cheated, and each customer

几周后，您将了解现在通用的约定，即整数在计算机硬件中使用二进制补码二进制表示（其中每个位表示 2 的幂）。但一些早期的计算机使用不同的表示形式，称为 BCD（二进制编码十进制）。在这个系统中，人类可读数字的每个十进制数字都直接使用四位进行编码。

[题外话：为什么是四位？使用 N 位，我们可以表示 2 个不同的值。因此，三位可以表示 23 = 8 个值。例如，如果我们想对一周中的某一天进行编码（星期一是 000，星期二是 001，等等），这就足够了;但对于十进制数字来说，这还不够。

因此，例如，整数 2479 将表示为四组，每组四位：

0010 0100 0111 1001

每组 4 个是一个无符号二进制小整数。

这种表示的意义何在？可以想象，对 BCD 数字执行算术运算的电路比用于二进制算术的电路复杂得多。BCD计算机与其他使用二进制的计算机同时存在，因此设计人员并不是不知道任何更好的替代方案。

BCD的原因是，在那些日子里，制造商向两个不同的市场出售两种不同类型的计算机。有使用二进制的科学计算机和使用 BCD 的商业计算机。与所有现代计算机一样，科学计算机具有二进制整数和非整数值的浮点表示法（我们将在后面研究）。商用计算机使用商务人士表示数字的格式的直接编码。由于四位可以编码 16 个值，并且有指令。所以我真的应该说

0010 0100 0111 1001 1111 0001 0000

假设 1111 代表小数点。当时的一些机器有一个完全在硬件中实现的 FORMAT 指令，该指令会将这样的 BCD 编号转换为格式化的 ASCII 文本字符串，包括程序员指定的额外字符，因此这个数字可能显示为 2,479.10 美元，或者，为了在写支票时填充可用空间，显示为 **********2,479.10 美元。

IBM System/360 在 1960 年代中期大张旗鼓地推出，作为适用于商业和科学计算的通用计算机，由于包括对二进制和 BCD 算术的硬件支持，它是“通用的”。（其名称中的“360”意味着它涵盖了一整套计算应用。

当然，今天，以二进制形式表示数字的计算机支持商业和科学应用程序。业务用户不必阅读二进制文件（科学家也不必阅读！这些数字在软件中转换为人类可读的十进制格式。在此示例中，抽象的一般原则采用将任务从硬件（低级抽象）移动到软件（高级）的特定形式。我们将看到，RISC（精简指令集计算）计算机设计方法依赖于同一想法的许多实例。

恐怕我讲这个故事的方式让早期的计算机设计师看起来像白痴。

故事更加复杂。例如，事实证明，今天普遍用于非整数值的二进制浮点表示不能精确表示分数 1/10，因此不能准确表示值 2479.10。这是企业用户不喜欢二进制计算机的原因之一;他们害怕四舍五入错误。如果一个不到一美元的错误看起来微不足道，想想几年前，一位在银行工作的程序员因编写计算储蓄账户利息的程序而入狱，以便它总是给客户正确的利息，四舍五入到整数美分;然后，它将所有账户中剩余的一美分相加，并将这笔钱存入程序员的账户。程序员推断，银行支付的正是它应该支付的，所以它没有被欺骗，每个客户

was getting the correct interest to within a penny, so what's the harm?

Of course the problem of fractions of a dollar can be solved by representing money amounts as a whole number of cents, rather than a fractional number of dollars. And an honest interest-calculation program pays interest rounded to the *nearest* penny, rather than rounding down.

1.2 The Stored Program Computer

One of the exciting big ideas in 61A is the metacircular evaluator, viewed as a *universal* program: the shift from having a separate program crafted for every function, such as a program to compute

$$x \mapsto 3x + 5,$$

to a single program, the metacircular evaluator (or, by extension, any programming language interpreter or compiler), which takes as part of its input *data* an encoding of the particular computation desired — in this example, the encoding is the character string

```
(lambda (x) (+ (* 3 x) 5))
```

An exactly analogous huge idea is at the center of computer hardware design. There were computers, of a sort, at least a few hundred years ago: machines that were built to perform one specific computation. The earliest such machines were closely analogous to the "function machine" model of software, in that you literally turned a crank to operate the machines. Later versions were electronic, but still had to be built (perhaps using a patchboard to connect up prebuilt components) to solve each individual problem. (The paradigmatic problem for which these machines were used was the numeric-approximation solution to a given differential equation, but military encryption machines such as the now-famous German "Enigma" machine are also in this category, since the operator physically replaced parts of the machine to change the encoding key.)

What made the modern computer possible was the idea of the "stored program." This means that instead of representing the problem to be solved in the circuitry of the computer, you represent the problem *as data* in the computer's memory, and what you put in the circuitry is the algorithm by which that representation (called a machine languge program) is interpreted. The central processor of a modern computer is, therefore, exactly analogous to the metacircular evaluator.

2 C as a Low-Level Language

Why don't people settle on the one best programming language and abandon all the others? One reason is that some languages are more "high-level" while others are more "low-level." [Note: In some contexts, people use the name "high-level" for everything other than a direct representation of the hardware's native machine language. But in the present context, high-level and low-level both refer to compiled or interpreted languages.]

"High-level" is not a compliment, and "low-level" is not an insult. As we'll see, each of these is appropriate in different situations.

level	language	sizeof(int)	best for
high	Scheme	"infinite"	applications, fast development
medium	Java	32 bits	applications, more optimization possible by "tuning" to hardware
low	C	depends on hardware	operating systems, compilers

每个客户都在一分钱之内获得正确的利息，那么有什么害处呢?
当然，一美元的分数问题可以通过将货币金额表示为整数美分而不是小数美元来解决。一个诚实的利息计算程序支付的利息四舍五入到最接近的一分钱，而不是四舍五入。

1.2 存储程序计算机

61A 中令人兴奋的大创意之一是元循环评估器，它被视为一个通用程序：从为每个函数设计一个单独的程序（例如计算程序）的转变

$$x\ 7 \rightarrow 3x + 5,$$

对于单个程序，元循环赋值器（或扩展为任何编程语言解释器或编译器），它将所需特定计算的编码作为其输入数据的一部分 - 在本例中，编码是字符串

```
(λ (x) (+ (* 3 x) 5))
```

一个完全相似的巨大想法是计算机硬件设计的中心。至少在几百年前，就有了某种计算机：为执行特定计算而建造的机器。最早的这种机器与软件的“功能机器”模型非常相似，因为你实际上是转动曲柄来操作机器。后来的版本是电子的，但仍然必须构建（可能使用接线板连接预构建的组件）来解决每个单独的问题。（使用这些机器的典型问题是给定微分方程的数值近似解，但军用加密机器，如现在著名的德国“Enigma”机器也属于这一类，因为操作员物理更换了机器的部件来更改编码密钥。
使现代计算机成为可能的是“存储程序”的概念。这意味着，您不是在计算机的电路中表示要解决的问题，而是将问题表示为计算机内存中的数据，并且您在电路中放入的是解释该表示的算法（称为机器语言程序）。因此，现代计算机的中央处理器与元循环评估器完全相似。

2 C 作为低级语言

为什么人们不选择一种最好的编程语言而放弃所有其他语言呢？原因之一是有些语言更“高级”，而另一些语言则更“低级”。[注意：在某些情况下，人们使用“high-level”这个名称来表示硬件的本机语言以外的所有内容。但在目前的上下文中，高级和低级都是指编译或解释的语言。
“高”不是恭维，“低”不是侮辱。正如我们将看到的，这些中的每一个都适用于不同的情况。

级语言 sizeof（int） 最适合高 Scheme “无限”应用程序，快速开发中型
Java 32 位
应用程序，通过“调整”硬件可以进行更多优化
low C 依赖于硬件操作系统、编译器

In a low-level language, the programmer is most aware of how the particular computer being used works; this places more burden on the programmer, but also allows more control over the precise way in which the computer carries out the computation. In a high-level language, the programmer works at a higher level of abstraction (this is where the names come from), with less need to know how this computer, or computers in general, perform tasks, and more able to focus on the problem being solved. This is why high-level languages are best for writing application programs (a word processor, a web browser, etc.), while low-level languages are best for those tasks in which knowledge of the hardware is part of the problem itself: operating systems and compilers.

This is why 61C, which is about low-level programming, is most important as preparation for 162 (Operating Systems), 152 (Architecture) and 164 (Compilers), while 61A and 61B are more important as preparation for application-oriented courses such as 184 (Graphics) and 188 (Artificial Intelligence).

The column headed `sizeof(int)` in the table above illustrates one specific example of how some languages work at higher or lower levels. Last week there was a lot of concern expressed in some of the questions you asked about the whole idea of *overflow*: the fact that some arithmetic computations produce answers that are not representable in the particular encoding that the computer uses for numeric values.

[Digression: Why is this surprising? In pure mathematics, we talk easily about infinitely many integers, all equally valid. We can prove theorems about these infinitely many values, such as the fact that there are infinitely many prime numbers. (Proof by contradiction: Suppose there were only finitely many primes, p_1 through p_n for some finite n. Then consider the integer

$$(p_1 \cdot p_2 \cdot \ \cdots \ p_n) + 1$$

This number is clearly larger than any of the p_i, but it isn't divisible by any of them; the part in parentheses is divisible by each p_i, and so the entire thing gives the remainder 1 when divided by p_i. So this number is either prime itself, or divisible by some prime not included in our supposedly complete list.) But once you want to *represent* numbers, in *any* medium of expression, only finitely many of them can be represented. Consider that there are only finitely many atoms in the universe! You can overflow paper-and-pencil, too.]

Is the possibility of overflow important in practice? Maybe. Early personal computers used 16-bit words, so their signed integers had a range of about plus and minus 32 thousand.

[Digression: How do I know this? How can you quickly estimate powers of two? You just memorize one of them:

$$2^{10} = 1024 \approx 1000$$

Knowing this, we can easily approximate other powers of two:

$$2^{15} = 2^5 \cdot 2^{10} \approx 32 \text{ thousand}$$
$$2^{16} = 2^6 \cdot 2^{10} \approx 64 \text{ thousand}$$
$$2^{32} = 2^2 \cdot 2^{10} \cdot 2^{10} \cdot 2^{10} \approx 4 \text{ billion}$$

The small powers of two you just count on your fingers, so to get 2^6 you just think "2, 4, 8, 16, 32, 64." Each 2^{10} corresponds to one of the three-digit groups ordinarily separated by commas in human-readable numerals, so in the last example above you think "thousand, million, billion."

在低级语言中，程序员最了解所使用的特定计算机编译器是如何工作的;这给程序员带来了更多的负担，但也允许对计算机执行计算的精确方式进行更多控制。在高级语言中，程序员在更高的抽象级别上工作（这就是名称的来源），不需要知道这台计算机或一般计算机如何执行任务，并且更能够专注于正在解决的问题。这就是为什么高级语言最适合编写应用程序（文字处理器、Web 浏览器等），而低级语言最适合那些硬件知识是问题本身的一部分的任务：操作系统和编译器。

这就是为什么 61C 是关于低级编程的，作为 162（操作系统）、152（架构）和 164（编译器）的准备工作最重要，而 61A 和 61B 作为面向应用程序的课程（如 184（图形）和 188（人工智能））的准备工作更为重要。

上表中标题为 sizeof（int） 的列说明了某些语言如何在更高或更低级别上工作的一个具体示例。上周，你问的一些关于溢出的整个概念的问题表达了很多担忧：一些算术计算产生的答案在计算机用于数值的特定编码中是无法表示的。

[题外话：为什么这令人惊讶？在纯数学中，我们很容易谈论无限多的整数，它们都同样有效。我们可以证明关于这些无限多值的定理，例如存在无限多的素数这一事实。（矛盾证明：假设只有有限个素数，pthrough p表示一些有限的n。然后考虑整数

$$(p\cdot p\cdot \cdots p) + 1$$

这个数字显然大于任何一个 p，但它不能被任何一个整除;括号中的部分可以被每个 p 整除，因此当除以 p 时，整个事情会得到余数 1。所以这个数字要么是素数本身，要么是可以被我们所谓的完整列表中未包含的素数整除。但是，一旦你想用任何表达媒介来表示数字，就只能表示有限数量的数字。想想看，宇宙中只有有限数量的原子！你也可以溢出纸和铅笔。

溢出的可能性在实践中重要吗？或。早期的个人计算机使用 16 位字，因此它们的有符号整数范围约为正负 3.2 万。

[题外话：我怎么知道这个？如何快速估计 2 的幂？你只需记住其中之一：

$$2= 1024 \approx 1000$$

知道了这一点，我们可以很容易地近似出两个的其他幂：

$$2= 2\cdot2\approx3.2万$$

$$2= 2\cdot2\approx6.4万$$

$$2= 2\cdot2\cdot2\cdot2\approx40亿$$

2 的小幂你只需用手指数数，所以要得到 2，你只需考虑“2、4、8、16、32、64”。每个 2 对应于三位数组中的一个，通常用人类可读数字中的逗号分隔，因此在上面的最后一个示例中，您会想到“千、百万、十亿”。

In the computer industry, by the way, the symbols K (kilo) and M (mega) don't always mean 1,000 and 1,000,000; sometimes they mean 1024 and whatever 1024 squared is — 2^{10} and 2^{20}. "64K" is actually somewhat more than 65,000. Technically K is always 1000, and M is always 1,000,000, while there are other suffixes KiBi for 1024 and MeBi for 1048576.

But I have to admit that I have 2^{15} memorized — it's exactly 32,768 — because that's the size (in words) of the first computer I learned on, the IBM 7094, so it got into my head back when my neurons were more flexible than they are now!]

32,768 isn't that big a number, so in the early days of computing it was quite sensible to worry a lot about overflow. Remember the Therac-25 X-ray machine that killed several patients? It stored the value of a counter in one 8-bit byte, so the maximum (unsigned) value before overflow was a mere 255. In hindsight, it's easy to see that the small saving of memory space, scarce as memory was in those days, by using a byte instead of a full word for the counter was irresponsible design.

Today's computers can all represent signed integers in the range plus to minus 2 billion. That's enough for most purposes; only astronomers, cryptographers and accountants working for the U.S. military need to worry about integer overflow.

Anyway, each of the languages in our table takes a different attitude toward the question of what range of integer values should be representable.

In C, a variable declared to be of type `int` is represented in a form "typically reflecting the natural size of integers on the host machine." [K&R, p. 36.] So, on the first IBM PCs, an `int` would be 16 bits wide, but on a modern PC it would be 32 bits wide. This is appropriate because integer arithmetic can be done entirely by the hardware, so it's very fast.

The problem with the C approach is that it hurts *portability*—the ability to take a program written on one computer and transfer it to a different computer. The problems that arise because of varying word sizes can be subtle; they aren't necessarily the quickly-found bugs that lead to crashing the program. The famous and influential Berkeley Unix implementation for the 32-bit Vax computer had a bug for several years before anyone noticed: Its random number library procedure, taken essentially unchanged from the (16-bit) PDP-11 Unix library, always returned an even number!

It was with these portability nightmares in mind that the designers of Java chose to specify that a Java `int` is always 32 bits wide, regardless of the hardware on which the program is run. An old 16-bit PC is required to use double-precision integer software for arithmetic on `int`s (as it would even in C, for variables declared to be `long int`); a current state-of-the-art supercomputer, with 64-bit words, is required to truncate the results of arithmetic computations on `int`s to 32 bits, wasting some of the power of the machine.

But even the Java designers were willing to require application programmers to worry about the finite limitations of the integer hardware. The designers of Scheme, an even higher level language, think that programmers shouldn't have to worry about how computers work at all! Mathematically, there is no limit to the size of an integer; we can't quite achieve that in any physical medium, but we'll do the best we can, so Scheme integers are limited only by the amount of memory available to the Scheme interpreter. That's why only Scheme, among the three languages in the table, can compute the factorial of 100 — not such an absurdly large problem as to be plausibly beyond the limits of practical computation. (Of course, programmers in any language can write their own unbounded-precision integer arithmetic library. But Scheme programmers don't have to. Of course other languages have common library support such as BigNumber in Java.)

[Does this mean that integer arithmetic is unacceptably slow in Scheme? No, because most Scheme implementations use the hardware integer representation for numbers less than 2^{31}; only when the

顺便说一句，在计算机行业，符号 K（公斤）和 M（兆）并不总是表示 1,000 和 1,000,000;有时它们的意思是 1024 和 1024 的平方——2 和 2。“64K”实际上略高于 65,000。从技术上讲，K 始终是 1000，M 始终是 1,000,000，而还有其他后缀 KiBi 表示 1024，MeBi 表示 1048576。

但我不得不承认，我已经记住了 2 个——正好是 32,768 个——因为这是我学习的第一台计算机 IBM 7094 的大小（用文字表示），所以当我的神经元比现在更灵活时，它就进入了我的脑海！32,768 并不是一个很大的数字，所以在计算的早期，担心溢出是很明智的。还记得杀死几名患者的 Therac-25 X 光机吗？它将计数器的值存储在一个 8 位字节中，因此溢出前的最大（无符号）值仅为 255。事后看来，很容易看出，通过使用字节而不是完整的单词来表示计数器来节省内存空间，就像当时的内存一样稀缺，这是不负责任的设计。

今天的计算机都可以表示正负 20 亿范围内的有符号整数。对于大多数目的来说，这已经足够了;只有为美国工作的天文学家、密码学家和会计师

军方需要担心整数溢出。

无论如何，我们表中的每种语言都对整数值应该表示的范围问题采取了不同的态度。

在 C 语言中，声明为 int 类型的变量以“通常反映主机上整数的自然大小”的形式表示。[K&R，第36 页。因此，在第一台 IBM PC 上，int 的宽度为 16 位，但在现代 PC 上，它的宽度为 32 位。这是合适的，因为整数算术可以完全由硬件完成，所以它非常快。

C 方法的问题在于它损害了可移植性，即在一台计算机上编写的程序并将其传输到另一台计算机的能力。由于字数大小不同而产生的问题可能很微妙;它们不一定是导致程序崩溃的快速发现的错误。著名且有影响力的 32 位 Vax 计算机的 Berkeley Unix 实现在任何人注意到之前已经存在了好几年的错误：它的随机数库过程，从（16 位）PDP-11 Unix 库中获取，总是返回一个偶数！

正是考虑到这些可移植性的噩梦，Java 的设计者选择指定 Java int 始终是 32 位宽，而不管运行程序的硬件如何。一台旧的 16 位 PC 需要使用双精度整数软件对整数进行算术运算（就像在 C 语言中一样，对于声明为长整数的变量）;一台当前最先进的超级计算机，具有 64 位字，需要将整数的算术计算结果截断为 32 位，从而浪费了机器的一些能力。

但即使是Java设计者也愿意要求应用程序程序员担心整数硬件的有限限制。Scheme（一种更高级别的语言）的设计者认为，程序员根本不用担心计算机是如何工作的！从数学上讲，整数的大小没有限制;我们无法在任何物理介质中完全实现这一点，但我们会尽我们所能，因此 Scheme 整数仅受 Scheme 解释器可用的内存量的限制。这就是为什么在表中的三种语言中，只有 Scheme 可以计算 100 的阶乘——这不是一个荒谬的大问题，以至于可能超出了实际计算的极限。（当然，任何语言的程序员都可以编写自己的无限精度整数算术库。但 Scheme 程序员不必这样做。当然，其他语言也有通用的库支持，例如 Java 中的 BigNumber。

[这是否意味着整数算术在 Scheme 中慢得令人无法接受？否，因为大多数 Scheme 实现都使用小于 2 的数字的硬件整数表示形式;只有当

hardware arithmetic would overflow does Scheme convert to "bignum" representation.]

Even the meaning of the word "integer" depends on the abstraction level of the programming language. In C and Java, the word "integer" is the name of a particular *representation format* for numbers, so the value `3.0` is not an integer in those languages. In Scheme, "integer" names a kind of number, not a kind of numeral, so `(integer? 3.0)` returns true, not false.

Research shows that a good programmer can produce a more-or-less constant number of debugged lines of code per day of work, independent of the language used. That's why program development is fastest if the language crams a lot of ideas into each line of code, which depends in part on avoiding things like variable declarations that reflect the details of hardware rather than the problem being solved. Thus, Lisp is often used in industry for "quick prototyping," which means getting something running as fast as possible, paying no attention to efficiency, so that the user interface (often the hardest part of the problem) can be debugged by trials with users. But lower-level languages can often produce faster-running machine language programs, which is why they're more often used in industry for the actual production versions of programs. Java is an attempt at a compromise, sufficiently high-level to avoid the worst hardware dependencies, but sufficiently low-level to produce efficient compiled programs.

[Note: None of these rules about high-level versus low-level languages are absolute. Many successful application programs have been written in C, and at least one good operating system (on the Lisp Machine) has been written in Lisp. And it's worth noting that modern computing environments, which have to worry about malicious attacks, have moved the consensus of professional opinion away from an overriding concern with low-level efficiency and toward a more high-level view in at least one area, the question of bounds-checking arrays. In C, as we'll see, an array is just a name for a particular kind of arithmetic on pointers; in Java and Scheme, array references are handled in a slower but safer manner.]

3 C pitfalls

C is a terrible programming language. Here are a few of the reasons.

3.1 15 Levels of Precedence

Back in elementary school we all learned about a two-level system for infix operator precedence: multiplication and division happen before addition and subtraction, so the expression $2 + 3 \times 4$ has the value 14, not 20. Remembering this isn't a huge cognitive load, even for us memory-limited human beings.

But C has that intimidating chart with 15 levels of precedence for its 45 operators! [K&R, p. 53.] How are you expected to remember them all?

For example, what does `3 + 4 << 5` mean? Is the result 7×2^5 or is it $3 + (4 \times 2^5)$? I have no idea. Looking at the table, I can determine that the precedence of dyadic + is just above that of <<, so this expression means `(3+4)<<5`. But someone reading my program, including me a month from now, won't have K&R open to that page, and therefore won't have any idea.

Answer: Don't even try. Whenever there's the slightest doubt what an expression might mean, use parentheses to group operators as you want them.

[Digression: Isn't it nice that this problem doesn't come up in Scheme? That's one of the virtues of a prefix notation for functions, rather than infix notation. If C used prefix notation, you could say

```
<< + 3 4 5
```

if you want the addition to happen first, or

因为大多数 Scheme 实现都使用小于 2 的数字的硬件整数表示;只有当硬件算术溢出时，Scheme 才会转换为“bignum”表示。甚至“整数”一词的含义也取决于编程语言的抽象级别。在 C 和 Java 中，“整数”一词是数字的特定表示格式的名称，因此值 3.0 在这些语言中不是整数。在 Scheme 中，“integer”命名一种数字，而不是一种数字，因此 （integer？ 3.0） 返回 true，而不是 false。

研究表明，一个好的程序员每天可以产生或多或少恒定数量的调试代码行，而与所使用的语言无关。这就是为什么如果语言在每一行代码中塞入大量想法，那么程序开发是最快的，这在一定程度上取决于避免诸如反映硬件细节而不是正在解决的问题的变量声明之类的事情。因此，Lisp在工业中经常被用于“快速原型设计”，这意味着让一些东西尽可能快地运行，不关注效率，这样用户界面（通常是问题中最困难的部分）就可以通过与用户的试验来调试。但是低级语言通常可以生成运行速度更快的机器语言程序，这就是为什么它们在工业中更常用于程序的实际生产版本。Java 是一种折衷的尝试，足够高的级别以避免最糟糕的硬件依赖，但足够低的级别可以生成高效的编译程序。

[注意：这些关于高级语言与低级语言的规则都不是绝对的。许多成功的应用程序都是用C语言编写的，并且至少有一个好的操作系统（在Lisp机器上）是用Lisp编写的。值得一提的是，现代计算环境不得不担心恶意攻击，已经将专业意见的共识从对低级效率的压倒一切的关注转移到了至少在一个领域（边界检查数组问题）的更高层次的观点。在 C 语言中，正如我们将看到的，数组只是指针上特定类型的算术的名称;在 Java 和 Scheme 中，数组引用以较慢但更安全的方式处理。

3 C 陷阱

C 是一种糟糕的编程语言。以下是一些原因。

3.1 15 个优先级级别

早在小学时，我们都学习过中缀运算符优先级的两级系统：乘法和除法发生在加法和减法之前，因此表达式 2 + 3 × 4 的值为 14，而不是 20。记住这一点并不是一个巨大的认知负担，即使对于我们这些记忆力有限的人来说也是如此。

但是 C 有一个令人生畏的图表，它的 45 个运算符有 15 个优先级！[K&R，第53页。你期望如何记住它们?

例如，3 + 4 << 5 是什么意思？结果是 7 × 2 还是 3 + （4 × 2）？我不知道。查看表格，我可以确定二元 + 的优先级刚好高于 << 的优先级，所以这个表达式的意思是 （3+4）<<5。但是，如果一个人阅读我的程序，包括一个月后的我，将不会打开该页面的K&R，因此不会有任何想法。

答：甚至不要尝试。每当对表达式的含义有丝毫疑问时，请使用括号根据需要对运算符进行分组。

[题外话：这个问题在 Scheme 中没有出现不是很好吗？这是函数的前缀表示法的优点之一，而不是中缀表示法。如果 C 使用前缀表示法，你可以说

```
<< + 3 4 5
```

如果您希望先进行添加，或者

```
+ 3 << 4 5
```

if you want the shift first.]

There are a few exceptions to the "just use parentheses" rule, C idioms that combine two operators but are so common that everyone knows what they mean. The most important example is the notation `*p++` that both dereferences and increments a pointer. According to the table, the increment operator `++` has the same precedence as the (monadic) dereference operator `*`, but operators at this level associate right to left, so the expression means `*(p++)`.

Alas, this true answer may be misleading; the incrementing happens *before* the dereferencing, but the result of the increment is it's prior value, not the incremented one. The grouping tells us that the increment operator will increment `p`, not `*p` (the thing that `p` points to). But the fact that the `++` operator comes *after* `p` (as opposed to the notation `*++p`) implies that `p`'s original value, not the value after incrementing, provides the input to the dereference operator.

[You may notice that in most C code, especially C code written long ago, the forms `*p++` and `*--p` are very common, whereas the equally legal forms `*++p` and `*p--` are less often used. This is because the first two can be compiled into a single instruction on the PDP-11 computer, the first one on which Unix was widely used, whereas the last two require more than one PDP-11 instruction. This now-obsolete coding practice is a great example of the low-level language mindset.]

[Indeed, one of the persistent myths of the C world is that the `++` and `--` operators were included in C *because of* the autoincrement and autodecrement feature of the PDP-11. But the inventors of C have pointed out repeatedly that the first implementation of C was on the PDP-7, which didn't have that feature.]

3.2 Type casting

An unusual feature of C is that the programmer is allowed to pretend that a variable declared in one type is actually of another type. For example, consider this code fragment:

```
int x, y;
int *p;
...
y = *p;      /* legal */
y = *x;      /* illegal */
y = *((int *)x);      /* legal! */
```

The first assignment is straightforward: `p` is a pointer to an integer value, so `*p` *is* an integer, and it's sensible to assign that value to the integer variable `y`.

The second statement is illegal. Since `x` is itself an integer, not a pointer — that is, not the memory address of something else — it makes no sense to dereference it. (That is, it makes no sense to ask what value is stored at memory location `x`, since `x`'s value is just a number, not a memory address.)

But the third, legal C statement says, "Pretend that `x` *does* contain a pointer to an integer, dereference that pointer, and set `y` to the value found at that location in memory."

The most likely result of this instruction is a runtime error, because the value of `x` won't in fact be a legal memory address, so the hardware will complain about an attempt to fetch a value from a nonexistent address.

Why does C allow this sort of thing? Here's one answer: Keep in mind that C was designed to enable Ken Thompson and Dennis Ritchie to write an operating system (Unix). One of the jobs of an

```
+ 3 << 4 5
```

如果你想要先换班。“只使用括号”规则有一些例外，C 成语组合了两个运算符，或者非常常见，每个人都知道它们的含义。最重要的示例是表示法 *p++，它既可以取消引用，也可以递增指针。根据该表，增量运算符 ++ 与（一元）取消引用运算符 * 具有相同的优先级，但此级别的运算符从右到左相关联，因此表达式表示 *（p++）。

唉，这个真实的答案可能具有误导性;递增发生在取消引用之前，但递增的结果是它的先验值，而不是递增的值。分组告诉我们，增量运算符将递增 p，而不是 *p（p 指向的事物）。但是，++ 运算符位于 p 之后（而不是符号 *++p）这一事实意味着 p 的原始值（而不是递增后的值）为取消引用运算符提供了输入。

[你可能会注意到，在大多数 C 代码中，尤其是很久以前编写的 C 代码中，*p++ 和 *--p 的形式非常常见，而同样合法的形式 *++p 和 *p-- 则不太常用。这是因为前两个可以在 PDP-11 计算机上编译成一条指令，第一个是 Unix 被广泛使用的指令，而后两个需要多个 PDP-11 指令。这种现在已经过时的编码实践是低级语言思维方式的一个很好的例子。

[事实上，C 世界的一个持续存在的神话是，由于 PDP-11 的自动增量和自递减功能，++ 和 -- 运算符被包含在 C 中。但 C 语言的发明者一再指出，C 语言的第一个实现是在 PDP-7 上，而 PDP-7 没有这个功能。

3.2 铸型

C 语言的一个不寻常的特性是，允许程序员假装在一种类型中声明的变量实际上是另一种类型的变量。例如，请考虑以下代码片段：

```
int x， y;
int *p;
...
y = *p;/* 合法 */ y = *x;/* 非法 */ y = *
（（int *）x）;/*法律！ */
```

第一个赋值很简单：p 是指向整数值的指针，因此 *p 是整数，将该值分配给整数变量 y 是明智的。

第二种说法是非法的。由于 x 本身是一个整数，而不是指针——也就是说，不是其他东西的内存地址——因此取消引用它是没有意义的。（也就是说，询问内存位置 x 存储的值是没有意义的，因为 x 的值只是一个数字，而不是内存地址。但第三个法律 C 语句说，“假设 x 确实包含指向整数的指针，取消引用该指针，并将 y 设置为在内存中该位置找到的值。

此指令最可能的结果是运行时错误，因为 x 的值实际上不是合法的内存地址，因此硬件会抱怨尝试从不存在的地址获取值。

为什么 C 允许这种事情发生？这里有一个答案：请记住，C 语言的设计是为了让 Ken Thompson 和 Dennis Ritchie 能够编写操作系统 （Unix）。的工作之一

OS is to control input/output devices. In most modern computers, the processor communicates with the I/O devices by means of *registers* that contain information about the status of the device, what the processor wants the device to do, and sometimes the actual values read or written by the device. To the processor, these registers look like memory, at particular addresses — each device has its own register addresses. The processor reads and writes the registers just like actual memory.

For example, suppose some device has a status register at memory address `0x1234`. (This is the C notation for a base-16 number.) We might say

```
int addr,val;
int *p;

addr = 0x1234;
val = *((int *)addr);

p = ((int *)0x1234);
val = *p;
```

Note that it would be illegal to say

```
p = 0x1234;
```

because a constant such as `0x1234` is an integer value, not a pointer.

[We'll see another example in which a value must be used both as a pointer and as an integer when we talk about writing a memory allocator.]

So, one pitfall is that it's easy to make mistakes when treating an integer as a pointer; you'd better pick an integer that really is a legal address that exists in your computer!

But another pitfall is that only *some* type casts mean "pretend this value is of that type." Consider this case:

```
int x;
float y;

x = 3;
y = (float)x;
```

This *doesn't* mean to take the 32 bits of `x` (in this case, the bits are `000...0011` since `x` has the value 3) and interpret them as if they were a floating-point number. This could be done, since `float` is also a 32-bit type; the result would be a very small number, which we can in fact compute with a different C program:

```
union {
  int i;
  float f;
} x;
float y;

x.i = 3;
y = x.f;
```

操作系统的工作之一是控制输入/输出设备。在大多数现代计算机中，处理器通过寄存器与 I/O 设备进行通信，寄存器包含有关设备状态、处理器希望设备执行的操作以及设备读取或写入的实际值的信息。对于处理器来说，这些寄存器在特定地址上看起来像内存——每个设备都有自己的寄存器地址。处理器读取和写入寄存器就像实际存储器一样。

例如，假设某些设备在内存地址0x1234处有一个状态寄存器。（这是以 16 为基数的数字的 C 表示法。我们可以说

```
int addr, val;
int *p;

addr = 0x1234;val = *( (int
*) addr) ;

p = ( (int *) 0x1234) ;瓦
尔 = *p;
```

请注意，说这是非法的

```
p = 0x1234;
```

因为像 0x1234 这样的常量是整数值，而不是指针。

[我们将看到另一个示例，其中当我们谈论编写内存分配器时，必须同时将值用作指针和整数。因此，一个陷阱是，将整数视为指针时很容易出错;您最好选择一个整数，该整数实际上是您计算机中存在的合法地址！但另一个陷阱是，只有某些类型转换意味着“假装此值属于该类型”。考虑以下情况：

```
整数 x;
浮点 Y;

x = 3;
y = （浮点数）x;
```

这并不意味着取 x 的 32 位（在本例中，位为 000...0011，因为 x 的值为 3）并将它们解释为浮点数。这是可以做到的，因为 float 也是一种 32 位类型;结果将是一个非常小的数字，我们实际上可以使用不同的 C 程序进行计算：

```
联合 {
  国际 I;
  浮点数 F;
} x;
浮点 Y;

x.i = 3;y =
x.f;
```

The value of `y` in this second program is 4.203895×10^{-45}. But in the first program, using the type cast, C interprets `(float)` not as "pretend this value is a float" but rather as "do the necessary work to convert this value from integer to floating point notation", so `y` gets the value `3.0`, numerically equal to `x`.

[In fact this particular situation doesn't require an explicit type cast; the C assignment operator automatically converts one numeric type to another as needed. But you might use such a type cast in a more complicated situation, such as this:

```
printf("%f\n", (float)x);
```

The arguments to `printf` can be any kind of number, and the C compiler generally doesn't know how `printf` formats work, so it doesn't know that `printf` will expect the second argument in floating-point representation rather than integer representation. Thus an explicit cast is required here.]

3.3 `switch` and `break`

C provides the `switch` statement for situations in which the program's actions should depend on the value of a given expression. For example, an interpreter for a programming language might look at a character that represents an operator and do different things for each operator:

```
switch (ch) {
  case '+':
            ...
            ...
            ...

  case '-':
            ...
            ...
            ...

  /* other cases */

  default:
            ...
            ...
            ...
}
```

But this doesn't mean what you probably think it does, if you're not an experienced C programmer. If `ch` contains the ASCII code for the + character, the program will do the commands under the first `case` label (`case '+'`), then the commands under `case '-'`, then the ones under `default`! If you want to keep the cases separate, you have to say this:

```
switch (ch) {
  case '+':
            ...
```

第二个程序中 y 的值是 4.203895 × 10。但是在第一个程序中，使用类型转换，C 不是将 （float） 解释为“假装此值是浮点数”，而是“执行必要的工作以将此值从整数转换为浮点表示法”，因此 y 得到值 3.0，数值等于 x。

[事实上，这种特殊情况不需要显式类型转换;C 赋值运算符会根据需要自动将一种数值类型转换为另一种数值类型。但是，您可能会在更复杂的情况下使用这样的类型转换，例如：

```
printf（“%f\n”， （浮点数）x）；
```

printf 的参数可以是任何类型的数字，C 编译器通常不知道 printf 格式是如何工作的，因此它不知道 printf 会期望第二个参数采用浮点表示而不是整数表示。因此，这里需要明确的演员转换。

3.3 开关和断开

C 为程序的操作应依赖于给定表达式的值的情况提供 switch 语句。例如，编程语言的解释器可能会查看表示运算符的字符，并为每个运算符执行不同的操作：

```
开关 （c） {
  案例“+”：
              ...
              ...
              ...

  大小写 '-'：
              ...
              ...
              ...

  /* 其他情况 */

  违约：
              ...
              ...
              ...
}
```

但这并不意味着你可能认为它的作用，如果你不是一个有经验的 C 程序员。
如果 ch 包含 + 字符的 ASCII 代码，则程序将在第一个大小写标签（大小写为“+”）下执行命令，然后在大小写“-”下执行命令，然后在默认情况下执行命令！如果你想把这些案例分开，你必须这样说：

```
开关 （c） {
  案例“+”：
              ...
```

```
            ...
            ...
    break;

  case '-':
            ...
            ...
            ...
    break;

  /* other cases */
  default:
            ...
            ...
            ...
    break;
}
```

(It doesn't hurt to put a `break` statement after the last case, the `default` in this example, too.)

Why does C behave this way? Sometimes it's what you want. Consider this possibility:

```
switch (ch) {
  case '-':
    arg = -arg;
  case '+':
            ...
            ...
            ...
    break;

  /* other cases */
  default:
            ...
            ...
            ...
}
```

But it's common practice in these situations to include a comment showing that the missing `break` is deliberate:

```
switch (ch) {
  case '-':
    arg = -arg;
    /* falls through */
  case '+':
            ...
```

```
                ...
                ...
    破;

  大小写 '-':
                ...
                ...
                ...
    破;

  /* 其他情况 */
  违约:
                ...
                ...
                ...
    破;
}
```

(在最后一种情况之后加上一个中断语句也没有什么坏处，这也是本例中的默认语句。为什么 C 会这样？有时这是你想要的。请考虑以下可能性：

```
开关 (c) {
  大小写 '-':
    参数 = -arg;
  案例“+”:
                ...
                ...
                ...
    破;

  /* 其他情况 */
  违约:
                ...
                ...
                ...
}
```

但在这些情况下，通常的做法是包含一条注释，表明缺少中断是故意的：

```
开关 (c) {
  大小写 '-':
    参数 = -arg;
    /* 落空 */
  案例“+”:
                ...
```

```
            ...
            ...
    break;

  /* other cases */

  default:
            ...
            ...
            ...
}
```

4 Storage Classes

C allows modifiers to be given with type declarations, syntactically similar to things like the `public` modifier that can be used with methods in Java. There are four of them:

```
extern
static
auto
register
```

Of these, `auto` is never seen in practice because it's the default for local variables and not permitted for globals; `register` is never seen in practice, in new C code, because modern compilers are much better than human beings at deciding which local variables to put in processor registers for optimal performance. So we'll just consider `extern` and `static`.

4.1 `extern`.

K&R use the word "external" to mean, more or less, global. This is not quite what the keyword `extern` means, although the two are related. A variable that's defined outside of any procedure is automatically global. What `extern` means is "this thing is defined somewhere else."

To clarify that statement, we have to talk about the distinction between *declaration* and *definition*. C compilers use variable-typing statements such as

```
int i;
```

for two purposes:

1. These statements tell the compiler what kind of thing this variable is, i.e., what representation convention is used.

For example, how does the compiler handle an arithmetic expression like `x+y`? The answer depends on whether `x` and `y` are integers or floats. When we talk about the MIPS machine language, we'll see that there's an instruction called `ADD` that adds two integer values in specified registers, putting the result in a third register; a different instruction called `ADD.S` adds two 32-bit floats; yet a third instruction `ADD.D` adds double-precision (64-bit) floats. The compiler doesn't know which instruction to use unless the variable types are declared before this expression is used.

```
                ...
                ...
    破;

  /* 其他情况 */

  违约:
                ...
                ...
                ...
}
```

4 存储类

C 允许使用类型声明来指定修饰符，这在语法上类似于 Java 中可以与方法一起使用的 public 修饰符之类的东西。其中有四个：

外部
静态的
auto
注册

其中，auto 在实践中从未见过，因为它是局部变量的默认值，而全局变量不允许;在新的 C 代码中，寄存器在实践中从未出现过，因为现代编译器在决定将哪些局部变量放入处理器寄存器以获得最佳性能方面比人类要好得多。因此，我们只考虑 extern 和 static。

4.1 外部。

K&R使用“外部”一词或多或少地表示全球性。这并不完全是关键字 extern 的意思，尽管两者是相关的。在任何过程之外定义的变量自动是全局变量。extern 的意思是“这个东西在别的地方被定义了”。

为了澄清这一说法，我们必须谈谈声明和定义之间的区别。C 编译器使用变量类型语句，例如

```
  国际 I;
```

有两个目的：1.这些语句告诉编译器这个变量是什么样的东西，即使用什么表示约定。

例如，编译器如何处理像 x+y 这样的算术表达式？答案取决于 x 和 y 是整数还是浮点数。当我们谈论MIPS机器语言时，我们会看到有一条名为ADD的指令，它将两个整数值相加指定的寄存器，将结果放在第三个寄存器中;另一条指令称为 ADD。S 添加两个 32 位浮点数;还有第三条指令 ADD。D 添加双精度（64 位）浮点数。编译器不知道要使用哪条指令，除非在使用此表达式之前声明了变量类型。

[Digression. So how does Scheme get away with letting you say `(+ x y)` without declaring whether `x` and `y` are integers or not? You learned the answer in 61A: tagged data! It's a misnomer to call languages like Scheme "untyped." They do have data types, but the types are associated with *values*, not with *variables*. The + procedure examines its argument values' types, and decides whether to use integer or floating-point instructions accordingly. We'll see shortly why this makes the programmer's life easier.]

2. The compiler must actually allocate memory for the variables. (We'll see that for local variables this allocation actually happens while the program is running, but memory for global variables is allocated as part of the compilation process.)

In C, a *declaration* serves only the first of these two purposes, telling the compiler the type of the thing being declared, but not allocating storage for it. A *definition* serves both purposes.

The easiest place to see this distinction is with respect to procedures. Here's a procedure *declaration*:

```
int foo(int x, char *p);
```

and here's a *definition* for the same procedure:

```
int foo(int x, char *p) {
    return x+strlen(p);
}
```

What's the point of using a declaration for a procedure separate from its definition? In C, a procedure must be declared before it can be called. This is because the compiler has to know the number and types of arguments the procedure expects, and the type of the value it returns, in order to compile the machine instructions needed to call it.

[This isn't a law of nature. Some languages use *two-pass* compilers that read the entire program, finding all the definitions (in C terminology) of procedures and variables, so that they know all the type information, and then read the entire program again to produce the machine language output. But C was designed to be compilable in one pass.]

It may seem a little weird to think of a procedure *definition* as allocating storage, but it does, namely the storage that contains the machine instructions to carry out the procedure.

[Note: A definition is, of course, also a declaration. So it's possible to avoid separate declarations by arranging your program so that procedures are always defined before they're used. This strategy leads to what might be called an upside-down program, in which the lowest-level helper procedures come first, and `main()` comes last. Indeed, you'll see C programs written that way.

But in some cases you can't use the upside-down strategy. One is a program with two procedures that use mutual recursion; they can't both be defined before each other. A more common example is that a large program is typically divided into several files, so procedures that are defined in one file but used in another file must be declared in the second one. (In practice, these declarations are often collected in a *header* file that's included in the compilation of every source file in the program.)]

The same principle applies to variables: They must be declared prior to every use, and they must be defined somewhere. Here's how `extern` distinguishes declarations from definitions:

```
int x;          /* see below */
int x=10;       /* definition, can only appear once */
extern int x;   /* declaration, explicitly not a definition */
```

[题外话。那么，Scheme 是如何让你说 （+ x y） 而不声明 x 和 y 是否是整数的呢？您在 61A 中学到了答案：标记数据！将 Scheme 等语言称为“非类型化”是用词不当。它们确实具有数据类型，但这些类型与值相关联，而不是与变量相关联。+ 过程检查其参数值的类型，并相应地决定是使用整数指令还是浮点指令。我们很快就会看到为什么这让程序员的生活更轻松。

2. 编译器必须实际为变量分配内存。（我们将看到，对于局部变量，这种分配实际上是在程序运行时发生的，但全局变量的内存是作为编译过程的一部分分配的。在 C 语言中，声明仅服务于这两个目的中的第一个目的，它告诉编译器所声明的事物的类型，但不为其分配存储。定义有两个目的。

最容易看到这种区别的地方是关于程序。下面是一个过程声明：

```
int foo（int x， char *p）；
```

以下是相同过程的定义：

```
int foo（int x， char *p） {
    返回 x+strlen（p）；
}
```

对与其定义分开的过程使用声明有什么意义？在 C 语言中，必须先声明过程，然后才能调用它。这是因为编译器必须知道过程期望的参数的数量和类型，以及它返回的值的类型，以便编译调用它所需的机器指令。

[这不是自然法则。一些语言使用双通道编译器来读取整个程序，找到过程和变量的所有定义（用 C 术语），以便它们知道所有类型信息，然后再次读取整个程序以产生机器语言输出。但是 C 被设计为可以一次性编译。

将过程定义视为分配存储似乎有点奇怪，但确实如此，即包含执行过程的机器指令的存储。

[注：当然，定义也是一种声明。因此，可以通过安排程序来避免单独的声明，以便始终在使用过程之前定义过程。这种策略导致了所谓的颠倒程序，其中最低级别的帮助程序放在第一位，main（） 放在最后。事实上，你会看到 C 程序是这样编写的。

但在某些情况下，您不能使用颠倒策略。一个是具有两个使用相互递归的过程的程序;它们不能在彼此之前定义。一个更常见的示例是，一个大型程序通常被划分为多个文件，因此必须在第二个文件中声明在一个文件中定义但在另一个文件中使用的过程。（在实践中，这些声明通常收集在头文件中，该文件包含在程序中每个源文件的编译中。

同样的原则也适用于变量：它们必须在每次使用之前声明，并且必须在某个地方定义。以下是 extern 区分声明和定义的方式：

```
int x; /* 见下文 */ int x=10; /* 定义，只能出现一次 */ extern int x;/* 声明，明确不是定义 */
```

```
extern int x=10;      /* error!  Tries to have it both ways. */
```

In all of these cases, we're thinking about what they mean outside of procedure definitions, so that `x` is meant to be a global variable.

The second form, with an "initializer" provided, has to be a definition (allocating the memory for `x`), because the compiler will put the value 10 in the allocated memory location as part of the compilation process. Although this looks like the executable assignment statement `x=10` that might appear inside a procedure, don't be confused; no machine language statements are compiled to load the value 10 and store it into memory at location `x`. Rather, this value is stored in `x` before the compiled program even begins running.

What about the first form, the plain `int x` with neither an `extern` nor an initializer? Technically, this is considered a definition (allocating storage), but it's a funny special case; this form can appear more than once in the program (typically, in different source files), and all of the appearances taken together allocate a single space for `x`. Other definitions, such as procedure definitions and variable definitions with initializers, can only appear once in the entire program. [But see the discussion of `static` below.]

It's rare to see `extern` used inside a procedure, but it's allowed; it tells the compiler that this procedure will use the declared variable or procedure, which is defined elsewhere. The `extern` declaration, like all local declarations, is in effect only inside this procedure.

4.2 `static`.

Sadly, the `static` keyword has two different meanings, depending on whether it's used inside or outside a procedure.

To get this straight, we have to start by recalling the meaning of *scope* and *extent* of variables.

The *scope* of a variable means where, in the program, this variable is available for use. Scheme uses *lexical* scope, which means that variables are available anywhere inside the procedure where they're defined, including internally defined procedures. Logo uses *dynamic* scope, which means that variables are available within the defining procedure and within any procedure invoked (directly or indirectly) by the defining procedure.

C, like Java, uses a degenerate (in the technical sense, not the moral sense!) form of lexical scope. Since there are no internal procedure definitions, local variables are available only within the same procedure that defines them.

But C also has two kinds of global scope. A global variable with "external linkage" is available throughout the program; a global variable with "internal linkage" is available only in the source file in which it is defined. Two different source files can define two different internally-linked variables with the same name, even if they have completely different types. Or a file can have an internal-linkage variable with the same name as an external-linkage variable shared among all the other source files of the program.

The *extent* of a variable means how long it exists. In pretty much every programming language, global variables have infinite extent, which means that they exist for the entire time that the program is running. But languages may differ regarding the extent of local variables. In Scheme, all variables have infinite extent; this is important because, combined with the first-class status of internally defined procedures, it makes possible the use of object-oriented programming "for free," without special additions to the language. [As you know, in practice, a Scheme interpreter is allowed to reclaim the storage

明确不是定义 */ extern int x=10;/*错误！尝试两者兼而有之。*/

在所有这些情况下，我们都在考虑它们在过程定义之外的含义，因此 x 应该是一个全局变量。

第二种形式，提供了“初始值设定项”，必须是一个定义（为 x 分配内存），因为编译器会将值 10 放在分配的内存位置，作为编译过程的一部分。尽管这看起来像可能出现在过程中的可执行赋值语句 x=10，但不要混淆;不会编译任何机器语言语句来加载值 10 并将其存储到位置 X 的内存中。相反，此值在编译后的程序开始运行之前就存储在 x 中。

第一种形式，既没有 extern 也没有初始值设定项的普通 int x 呢？从技术上讲，这被认为是一个定义（分配存储），但这是一个有趣的特例;此表单可以在程序中多次出现（通常，在不同的源文件中），并且所有表单加在一起都会为 X 分配一个空间。其他定义（如过程定义和带有初始值设定项的变量定义）在整个程序中只能出现一次。[但请参阅下面关于静态的讨论。

在程序中使用 extern 的情况很少见，但这是允许的;它告诉编译器，此过程将使用声明的变量或过程，该变量或过程在别处定义。与所有本地声明一样，外部声明仅在此过程中有效。

4.2 静态的。

可悲的是，静态关键字有两种不同的含义，具体取决于它是在过程内部还是外部使用。

为了弄清楚这一点，我们必须首先回顾变量范围和范围的含义。变量的作用域是指该变量在程序中可供使用的位置。Scheme 使用词法作用域，这意味着变量在定义变量的过程内的任何位置都可用，包括内部定义的过程。徽标使用动态作用域，这意味着变量在定义过程以及定义过程（直接或间接）调用的任何过程中都可用。

像Java一样，C语言使用一种退化的（在技术意义上，而不是道德意义上的！）形式的词汇范围。由于没有内部过程定义，因此局部变量仅在定义它们的同一过程中可用。

但 C 也有两种全局作用域。具有“外部链接”的全局变量在整个程序中可用;具有“内部链接”的全局变量仅在定义它的源文件中可用。两个不同的源文件可以定义两个具有相同名称的不同内部链接变量，即使它们具有完全不同的类型。或者，文件可以具有与程序的所有其他源文件共享的外部链接变量同名的内部链接变量。

变量的范围意味着它存在的时间。在几乎所有的编程语言中，全局变量都具有无限的范围，这意味着它们在程序运行的整个过程中都存在。但是语言在局部变量的范围方面可能会有所不同。在 Scheme 中，所有变量都具有无限范围;这很重要，因为结合内部定义过程的一流地位，它使得“免费”使用面向对象编程成为可能，而无需对语言进行特殊添加。[如您所知，在实践中，允许 Scheme 解释器回收存储

used by a variable if it can prove that the variable can't be accessed any longer, because nothing points to it.] In Logo, local variables have local extent; they are destroyed and their space reclaimed when the procedure that defines them returns to its caller.

In C, both infinite and local extent are available; they are called "static" and "automatic," respectively.

Now we can see what the `static` keyword means:

	defined inside procedure	defined globally
STATIC used	local scope, infinite extent	same-file scope, infinite extent
STATIC not used	local scope, local extent	whole-program scope, infinite extent

As you can see by reading down the columns, inside a procedure the `static` keyword determines a variable's extent, but outside a procedure, the same keyword determines a variable's scope. This is my least favorite C misfeature, because it could so easily have been avoided by using different keywords for the different meanings. [The execrable C++ language retains both of these meanings and adds a third one, for even greater confusion.]

5 The `main()` Procedure

When a C program starts up, what runs first is a little piece of machine language code that's the same for every program, whose job is to set up some machine registers; it then calls your procedure named `main`. Here's the definition of `main`:

```
int main(int argc, char **argv);
```

[Actually there's an optional third argument, the environment pointer, which allows access to the shell variables set with the `setenv` shell command. But we'll ignore that for the moment.]

The return value from `main` should be zero if the program does its job successfully, or a nonzero value indicating what went wrong.

The arguments to `main` are the arguments typed by the user on the shell command line when starting up the program. For example, if you type the shell command

```
foo hello 87
```

so that the shell starts a program named `foo`, the program will see three command line arguments, in the form of character strings:

```
argv[0] = "foo"
argv[1] = "hello"
argv[2] = "87"
```

(Notice that the last argument, even though it consists of digits, is not converted to the internal integer format; all arguments are given to the program as character strings.)

Since the program name is used as the first argument, there is always at least one program argument! The first argument to `main`, the integer `argc`, is the number of arguments used, namely 3 in this example.

如果 Scheme 解释器可以证明变量无法再被访问，则允许它回收变量使用的存储，因为没有任何指向它。在 Logo 中，局部变量具有局部范围;当定义它们的过程返回到其调用方时，它们将被销毁并回收其空间。
在 C 中，无限范围和局部范围都可用;它们分别被称为“静态”和“自动”。

现在我们可以看到 static 关键字的含义：

在过程中定义	全局	静态
使用局部作用域、同文件作用域、	无限范围	无限范围
STATIC 未使用本地范围，	局部范围	全程序范围，无限范围

通过向下阅读列可以看出，在过程内部，static 关键字确定变量的范围，但在过程之外，同一关键字确定变量的范围。这是我最不喜欢的 C 错误功能，因为通过使用不同的关键字来表达不同的含义，可以很容易地避免它。[可执行的 C++ 语言保留了这两种含义，并添加了第三种含义，以造成更大的混淆。

5 main（） 过程

当一个 C 程序启动时，首先运行的是一小段机器语言代码，每个程序都是相同的，其工作是设置一些机器寄存器;然后，它将名为 Main 的过程调用。以下是 main 的定义：

```
在 int（int argk， char **argav） 中;
```

[实际上，还有一个可选的第三个参数，即环境指针，它允许访问使用 setenv shell 命令设置的 shell 变量。但我们暂时忽略这一点。如果程序成功完成其工作，则 main 的返回值应为零，或者为非零值，表示出了什么问题。
main 的参数是用户在启动程序时在 shell 命令行上键入的参数。例如，如果键入 shell 命令

```
Foo Hello 87（你好 87）
```

因此，Shell 启动一个名为 Foo 的程序，程序将看到三个命令行参数，以字符串的形式出现：

```
argv[0] = “foo”
Argav[1] = “你好”
argv[2] = “87”
```

（请注意，最后一个参数，即使它由数字组成，也不会转换为内部整数格式;所有参数都以字符串的形式提供给程序。由于程序名称用作第一个参数，因此始终至少有一个程序参数！main 的第一个参数，整数 argc，是使用的参数数，即本例中的 3。

The second argument to `main`, called `argv`, is an array of pointers to strings. In the declaration above, it's shown as `char **argv`, meaning that it's a pointer to a pointer to a character. It could also have been declared this way:

```
int main(int argc, char *argv[]);
```

which indicates, as the example above suggests, that `argv` is an array of pointers to characters. How can `argv` be both an array and a pointer? Why are these two declarations equivalent? That gets us to the next topic.

[Digression: If you use an asterisk in a shell command, as in the famous bad idea

```
rm *
```

(don't do that!), the result is not that `rm` is given `argv[1]="*"`. Rather, the shell turns the asterisk into a list of all files in the current directory, so we have

```
argv[0] = "rm"
argv[1] = "aardvark"
argv[2] = "ablative"
argv[3] = "abnegate"
...
```

or whatever your filenames are.]

6 Arrays and Pointers

6.1 Memory addresses.

The computer's memory is itself an array of data, but the data are of different sizes. On most computers today the situation is this:

```
character = 1 byte = 8 bits
integer pointer   = 1 word = 4 bytes = 32 bits float
double    = 2 words = 8 bytes = 64 bits
```

[Digression: Always use `double`, not `float`. 32-bit floats just don't have enough precision; in particular, any `int` can be represented exactly as a `double`, but not all `int`s can be represented exactly as a `float`. 32-bit floating point is a leftover from the days when memory was scarce and also when loading two words into the processor took a long time.]

[Digression: The sizes shown above are not laws of nature. On my favorite old computer, the PDP-10, a word was 36 bits wide. A byte could have any width from 1 to 36 bits! The original ASCII code had only 128 defined codes, enough to represent all the letters and digits and punctuation of English with some left over, so we could fit five ASCII characters in a 36-bit word with one bit left over. (And some software found a use for that bit!) But Emacs users wanted to be able to apply the Control and Meta modifiers to any ASCII character, so for that purpose we used nine-bit bytes, seven ASCII code bits plus two modifier bits, with four characters just fitting in a word. The days of variable-width bytes are over, but there are already computers with 64-bit words. That may seem like overkill, but after all, there are 2^{80} atoms in the universe according to one of the students, so even 64 bits might not be enough. More realistically, cryptography uses extremely wide integers; 128 bits is currently viewed as the minimum acceptable width for low-security keys.]

main 的第二个参数称为 argv，是指向字符串的指针数组。在上面的声明中，它显示为 char **argv，这意味着它是指向字符的指针的指针。它也可以这样声明：

```
在 int （int argk， char *argav[]）；
```

如上例所示，它表示 argv 是指向字符的指针数组。
argv 如何既是数组又是指针？为什么这两个声明是等价的？这就引出了下一个话题。

[题外话：如果你在shell命令中使用星号，就像著名的坏主意一样

```
rm *
```

（不要那样做！），结果不是 rm 被赋予 argv[1]= “*” 。相反，shell 将星号转换为当前目录中所有文件的列表，因此我们有

```
argv[0] = “rm” argv[1] = “土
豚” argv[2] = “烧蚀” argv[3] =
“abnegate” ...或者你的文件名是
什么。
```

6 数组和指针

6.1 内存地址。

计算机的内存本身就是一个数据数组，但数据的大小不同。在当今的大多数计算机上，情况是这样的：

```
字符 = 1 字节 = 8 位整数指针 = 1 字 = 4 字节 = 32 位浮点数 double = 2
字 = 8 字节 = 64 位
```

[题外话：始终使用双精度，而不是浮点数。 32 位浮点数不够精确;特别是，任何 int 都可以精确地表示为双精度，但并非所有 int 都可以精确地表示为浮点数。 32 位浮点是内存稀缺的时代遗留下来的，而且将两个字加载到处理器中需要很长时间。

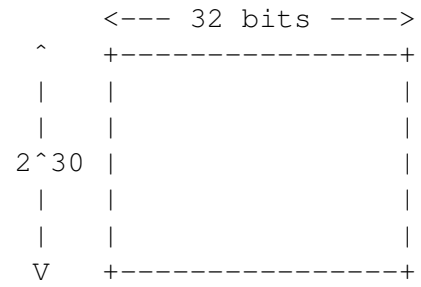
[题外话：上面显示的尺寸不是自然法则。在我最喜欢的旧电脑 PDP10 上，一个字是 36 位宽。一个字节可以有 1 到 36 位的任何宽度！最初的 ASCII 代码只有 128 个定义的代码，足以表示英语的所有字母、数字和标点符号，还有一些剩余的，因此我们可以在一个 36 位单词中容纳五个 ASCII 字符，剩下一位。（一些软件发现了该位的用途！但是 Emacs 用户希望能够将 Control 和 Meta 修饰符应用于任何 ASCII 字符，因此为此，我们使用了 9 位字节、7 个 ASCII 代码位和 2 个修饰符位，四个字符刚好适合一个单词。可变宽度字节的时代已经结束，但已经有 64 位字的计算机。这似乎有点矫枉过正，但毕竟，根据其中一位学生的说法，宇宙中有 2 个原子，所以即使是 64 位也可能不够。更现实地说，密码学使用极宽的整数;128 位目前被视为低安全性密钥的最小可接受宽度。

Today it's a universal *de facto* standard that the smallest addressible unit of memory is the eight-bit byte, and the widespread use of the C language played a role in making that the standard. It's not obviously optimal. Consider another data type that C doesn't include, but other languages do: the Boolean type whose only possible values are True and False. (Scheme is one language in which this is a primitive type; C uses integer types for the purpose, with zero for False and any nonzero value representing True.) It only takes one bit to represent a Boolean variable; if we need an array of 1000 Booleans, we can save a lot of memory by cramming each array element into a single bit, instead of using a whole byte (the smallest addressible unit) for each element. C does make it possible to extract a single bit from memory, but it's not as simple as just dereferencing a pointer.

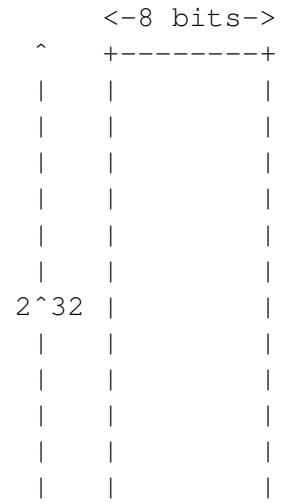
Every byte of memory has an address. An *address* is an unsigned integer, so the highest possible address on a 32-bit computer is $2^{32} - 1$, not $2^{31} - 1$.

[Digression: Do any of you own a 4-gigabyte computer? (No.) How about 1-gigabyte? (A few.) Half a gig? (Lots.) Indeed, 512Mb (2^{30} bytes) is the typical size of personal computer memories today, so we are within a factor of four of needing a wider architecture. And there are 4-gig computers, generally owned by people who run big servers, like Google. By the way, there are various architecture kludges available to allow for more memory than the architecture can directly address; if history is a guide, we'll see such architectures for a while just after 8-gig memories become cheap and before 64-bit processors become cheap.]

On most current architectures, including the MIPS architecture we'll be studying, an `int` must have an address that's divisible by 4. So if you think of memory as being full of integers, it looks like this:



But if you think of memory as full of characters, it looks like this (not to scale):

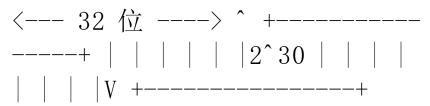


今天，内存的最小可寻址单位是 8] 位字节，这是一个普遍的事实标准，而 C 语言的广泛使用在使其成为标准方面发挥了作用。这显然不是最佳的。考虑 C 不包含但其他语言包含的另一种数据类型：布尔类型，其唯一可能的值为 True 和 False。（Scheme 是一种语言，其中这是一种原始类型;C 使用整数类型，零表示 False，任何非零值表示 True。表示布尔变量只需要一位;如果我们需要一个 1000 布尔值的数组，我们可以通过将每个数组元素塞进一个位来节省大量内存，而不是为每个元素使用整个字节（最小的可寻址单位）。C 确实可以从内存中提取单个位，但它并不像取消引用指针那么简单。

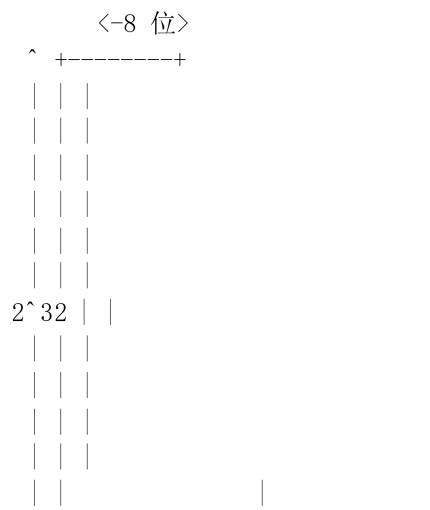
内存的每个字节都有一个地址。地址是无符号整数，因此 32 位计算机上可能的最高地址是 2− 1，而不是 2− 1。

[题外话：你们中有人拥有 4 GB 的计算机吗？（编号）1 GB 怎么样？（一些。半场演出？（很多。事实上，512Mb（2字节）是当今个人计算机内存的典型大小，因此我们需要更广泛的架构。还有 4 GB 计算机，通常由运行大型服务器的人拥有，例如 Google。顺便说一句，有各种架构 kludges 可用来允许比架构可以直接处理的内存更多;如果以史为鉴，那么在8-G内存变得便宜之后，在64位处理器变得便宜之前，我们将看到这样的架构。

在大多数当前的架构上，包括我们将要研究的 MIPS 架构，int 必须具有可被 4 整除的地址。因此，如果您认为内存充满了整数，它看起来像这样：



但是，如果您将内存视为充满字符，它看起来像这样（不按比例缩放）：



```
|    |          |
V    +---------+
```

Saying this another way, the rightmost two bits in any `int` address are zero.

[Why? Remember that in binary integer notation, each bit represents a power of two. Reading from right to left, the rightmost bits represent 20=1, 21=2, 22=4, 23=8, etc. All those values except for the first two are divisible by four, whereas the rightmost two values aren't divisible by four. So the entire number will be divisible by four if and only if those rightmost two bits are zero. The rightmost two bits encode the remainder on dividing the entire number by four.]

By the way, on most current computers, including the MIPS, double-precision floating point values are *not* required to be at addresses that are a multiple of 8. The address must be a multiple of 4. This may seem strange, but it's a consequence of the fact that these machines have a 32-bit bus architecture – they can transfer 32 bits at a time to or from memory. So a `double` has to be loaded or stored in two steps anyway, which means that there's no hardware advantage to a multiple-of-eight address.

6.2 Endianness.

Okay, so the integer at address 1000 includes the four bytes at addresses 1000, 1001, 1002, and 1003. But in what order are those four bytes assembled to make up the word? That is, does byte number 1000 represent the values from 20 to 27 (the rightmost eight bits of the word), or the values from 2^{24} to 2^{31} (the leftmost eight bits)? There is no universal standard about this. Machines in which byte 1000 is on the left are called *big-endian*, and machines in which byte 1000 is on the right are called *little-endian*, using terminology introduced to computer science by Danny Cohen.

[Digression: He didn't invent the names; they come from *Gulliver's Travels* by Jonathan Swift. When Gulliver visits Lilliput, the land of the really small people, he finds the Lilliputians in the middle of a war between two groups, the big endians and the little endians. The reason for the dispute is that they can't agree about whether to open an egg by cracking the big end or the little end. Swift presented this story as an allegory for the religious wars between Christian sects, such as Catholics versus Protestants, which Swift thought were about doctrinal differences no more important than this egg-cracking one. Cohen's point is a little different, as he says in his paper: As in Lilliput, there's no really strong reason to prefer big-endian or little-endian computer architecture, but unlike the situation in Lilliput, it would be really beneficial if everyone agreed — just as there's no strong reason why green has to mean go and red has to mean stop, but it'd be problematic if different people followed different conventions about this, and so everyone agrees on the one arbitrary but universal traffic light standard. So far Cohen hasn't succeeded in his quest to get all computer designers to agree on a single endianness, though, even though everyone cites his paper approvingly.]

6.3 C Pointer Types.

Since a pointer contains a memory address, which is an unsigned integer, the pointer type is fundamentally the same as the unsigned integer type in its size (32 bits) and its represented meaning (each bit is a power of two). But the *operations* allowed on pointers are quite different from those on integers.

You're well acquainted with the integer (including unsigned int) operations: arithmetic (+, −, ⋆, /), comparison (<, >, ==), and so on.

For pointers, we'll see that a restricted set of arithmetic operations are allowed, but the most important operation is one that isn't allowed with ints, namely the *dereference* operator, represented as

```
| | |
V +---------+
```

换句话说，任何 int 地址中最右边的两位都是零。
[为什么？请记住，在二进制整数表示法中，每个位表示 2 的幂。从右到左读取，最右边的位表示 20=1、21=2、22=4、23=8 等。除前两个值外，所有这些值都可以被 4 整除，而最右边的两个值不能被 4 整除。因此，当且仅当最右边的两位为零时，整个数字才能被四整除。最右边的两位在将整数除以 4 时对余数进行编码。

顺便说一句，在大多数当前计算机（包括 MIPS）上，双精度浮点值不需要位于 8 的倍数地址。地址必须是 4 的倍数。这可能看起来很奇怪，但这是由于这些机器具有 32 位总线架构——它们可以一次将 32 位传输到内存或从内存传输。因此，无论如何，双精度地址必须分两步加载或存储，这意味着八分之八的倍数地址没有硬件优势。

6.2 字节序。

好的，地址 1000 处的整数包括地址 1000、1001、1002 和 1003 处的四个字节。但是这四个字节是按什么顺序组合起来构成单词的呢？也就是说，字节号 1000 是表示从 20 到 27（单词最右边的 8 位）的值，还是从 2 到 2（最左边的 8 位）的值？对此没有通用标准。字节 1000 在左边的机器称为 big-endian，字节 1000 在右边的机器称为 little-endian，使用 Danny Cohen 引入计算机科学的术语。

[题外话：这些名字不是他发明的，它们来自乔纳森·斯威夫特的《格列佛游记》。
当格列佛访问小人国时，他发现小人国正处于两个群体之间的战争中，大端人和小端人。争论的原因是，他们无法就是否通过破解大端或小端来打开鸡蛋达成一致。斯威夫特将这个故事作为基督教教派之间宗教战争的寓言，例如天主教徒与新教徒，斯威夫特认为这是关于教义差异的，并不比这个鸡蛋破裂更重要。Cohen 的观点有点不同，正如他在论文中所说： 就像在小人国一样，没有真正强烈的理由偏爱大端或小端计算机架构，但与小人国的情况不同，如果每个人都同意，那将是非常有益的——就像没有强有力的理由为什么绿色必须意味着去，红色必须意味着停止一样， 但是，如果不同的人对此遵循不同的惯例，那将是有问题的，因此每个人都同意一个武断但通用的交通信号灯标准。到目前为止，科恩还没有成功地让所有计算机设计者就单一的字节序达成一致，尽管每个人都赞许地引用了他的论文。

6.3 C 指针类型。

由于指针包含一个内存地址，该地址是一个无符号整数，因此指针类型在大小（32 位）和表示含义（每个位是 2 的幂）方面与无符号整数类型基本相同。但是指针上允许的操作与整数上的操作完全不同。

您非常熟悉整数（包括无符号整数）运算：算术（+、-、*、/）、比较（<、>、==）等。

对于指针，我们将看到允许一组受限制的算术运算，但最重要的运算是不允许使用 int 的运算，即取消引用运算符，表示为

a monadic `*`, as in `*p`. This means, "Get me the contents of the memory location whose address is in variable `p`," or, if `*p` is used to the left of the assignment operator (=), "store something into the memory location whose address is in variable `p`."

Suppose pointer `p` contains the value 1000, and I dereference the pointer by saying `*p`. Does this mean that I want the byte at address 1000, or the *word* at address 1000 (which comprises the four bytes at addresses 1000, 1001, 1002, and 1003)?

This ambiguity explains why C doesn't just have a `pointer` type analogous to the `int` type. A variable can't just be a pointer in general; it has to be a pointer to some particular data type, so that the compiler knows whether (for example) to compile `*p` into a MIPS `LB` (Load Byte) instruction or into a `LW` (Load Word) instruction.

In particular, `char **argv` means that `argv` is a pointer to data of type `char *` — a pointer to a pointer to a character.

In my examples so far, the targets of pointers have all been either one-byte or four-byte data. But we'll see later, when we talk about structs, that we can have pointers to a chunk of memory of any size.

6.4 `void` pointers

I said just now that a variable can't just be a generic pointer, but must be a pointer to a specific data type, so that the compiler knows how to dereference it. This is generally true, but there are situations in which it's useful to be able to defer the specification of the type — to tell the compiler, "this is a pointer (so it is itself four bytes wide), but I'll tell you later what kind of thing it points to." We say this with the declaration

```
void *p;
```

Why would we need this? Well, in many situations, this whole business of declaring types of variables is a pain in the neck. My favorite example is the Scheme procedure

```
(define (square x)
  (* x x))
```

How can we translate this into C, or into Java? Answer: We can't. Instead we have to write two (or more) procedures:

```
int isquare(int i) {
    return i * i;
}

double dsquare(double d) {
    return d * d;
}
```

[In Java, by making these methods of different classes, we can give them both the name `square`, but we still need two of them.]

Now suppose we want to write a higher-order function like `map`. The first argument to `map` will be a pointer to a procedure, perhaps the procedure `isquare` or `dsquare`. [Procedures aren't first-class data in C, so the procedure itself can't be the argument, but C does allow pointers to procedures as

表示为一元 *，如 *p。这意味着，“获取地址在变量 p 中的内存位置的内容”，或者，如果在赋值运算符 （=） 的左侧使用 *p，“将某些内容存储到地址在变量 p 中的内存位置。

假设指针 p 包含值 1000，我通过说 *p 来取消引用指针。这是否意味着我想要地址 1000 处的字节或地址 1000 处的字（包括地址 1000、1001、1002 和 1003 处的四个字节）？

这种歧义解释了为什么 C 不仅具有类似于 int 类型的指针类型。一般来说，变量不能只是一个指针;它必须是指向某些特定数据类型的指针，以便编译器知道（例如）是将 *p 编译为 MIPS LB（加载字节）指令还是编译为 LW（加载字）指令。

具体而言，char **argv 表示 argv 是指向 char * 类型数据的指针 — 指向指向字符的指针的指针。

到目前为止，在我的示例中，指针的目标都是单字节或四字节数据。但是我们稍后会看到，当我们讨论结构时，我们可以有指向任何大小的内存块的指针。

6.4 void 指针

我刚才说过，变量不能只是一个泛型指针，而必须是指向特定数据类型的指针，这样编译器才能知道如何取消引用它。这通常是正确的，但在某些情况下，能够推迟类型的规范是有用的——告诉编译器，“这是一个指针（所以它本身有四个字节宽），但我稍后会告诉你它指向什么样的东西。我们在宣言中这样说

```
无效 *p;
```

我们为什么需要这个？好吧，在许多情况下，声明变量类型的整个业务是一件令人头疼的事情。我最喜欢的例子是 Scheme 过程

```
（定义（平方 x）
  （* x x））
```

我们怎样才能把它翻译成C语言，或者翻译成Java呢？答：我们不能。相反，我们必须编写两个（或更多）过程：

```
int isquare(int i) {
    返回 i*i;
}

双 dsquare(双 d) {
    返回 d * d;
}
```

[在 Java 中，通过将这些方法创建为不同的类，我们可以给它们都命名为 square，但我们仍然需要其中两个。现在假设我们想编写一个像 map 这样的高阶函数。要映射的第一个参数将是指向过程的指针，可能是过程 isquare 或 dsquare。[过程在 C 语言中不是一流的数据，因此过程本身不能作为参数，但 C 确实允许指向过程的指针

first-class data.] The second argument will be a pointer to (let's say) an array, but will it be an array of `int`s or an array of `double`s? We don't know, so we'll declare the pointer as a pointer to anything.

Of course, since we don't know the size of the thing to which the pointer points, we can't do anything with it: we can't dereference it, and we can't do arithmetic on it. To use the pointer, we must cast it to some specific pointer type:

```
*p              /* illegal */
*((int *)p)     /* legal */
```

6.5 Arrays.

An array is a bunch of things, all of the same type, next to each other in memory. For example, here is an array of 10 integers, starting at memory address 1000:

```
1000:    a[0]
1004:    a[1]
1008:    a[2]
     ...
1036:    a[9]
```

Note that there is no `a[10]`, but if it existed, it would be at address 1040 — the starting address plus 4*10, the size of an integer times the number of integers in the array.

Suppose we want to do something with each element in the array. One way to do it would be to set up a pointer-to-integer that starts out pointing to the first element:

```
int *p = &a[0];
```

The ampersand (`&`) operator is the opposite of dereferencing; it takes a variable (must be a variable, not an arbitrary expression) as its operand, and returns the memory address at which that variable is stored. So, for our example array, the value of `&a[0]` is 1000; the value of `&a[2]` is 1008. But, as discussed earlier, we can't just assign 1000 to `p` because `p` is a variable of type pointer, whereas 1000 is a value of type integer. (And in any case, in practice we wouldn't know the address at which the compiler decided to put the array `a`.)

Given this pointer `p` that points to the first element of `a`, how do we get to the next element? C provides two similar but different notations to describe this process:

```
p+1     /* compute the value of a pointer to the next element */
p++     /* same, but replace p with the new pointer value */
```

But shouldn't that be `p+4`? The next element of the array has an address four greater than what's in `p`, not just one greater.

But `p+1` is correct. C's pointer arithmetic takes the size of the pointer's target into account. So if `p` is declared as a pointer to an integer, then `p+1` or `p++` will actually add 4 to the value of `p`. But if `p` is a pointer to character, then `p+1` and `p++` add 1 to the pointer. Similarly, if `p` is a pointer to double, incrementing the pointer adds 8 to it, and so on for other types.

[Slightly tricky example: What does `argv++` mean? Since `argv` was declared as `char **argv`, a too-quick reading might suggest that `argv++` adds 1 to it. But that would be true only if the declaration were `char *argv`. The extra asterisk means that `argv` is of type pointer-to-pointer, so its target (a pointer) is 4 bytes wide, so `argv++` adds 4 to `argv`.]

一流的数据。第二个参数将是指向（比方说）数组的指针，但它是整数数组还是双精度数组？我们不知道，因此我们将指针声明为指向任何内容的指针。

当然，由于我们不知道指针所指向的事物的大小，我们不能对它做任何事情：我们不能取消引用它，我们不能对它进行算术运算。要使用指针，我们必须将其转换为某些特定的指针类型：

```
*p /* 非法 */ *（（int *）p） /* 合
法 */
```

6.5 阵列。

数组是一堆东西，都是相同的类型，在内存中彼此相邻。例如，下面是一个包含 10 个整数的数组，从内存地址 1000 开始：

```
1000：一个[0]
1004年：一个[1]
1008年：一个[2]
      ...
1036年：一个[9]
```

请注意，没有 a[10]，但如果它存在，它将位于地址 1040 — 起始地址加上 4*10，整数的大小乘以数组中的整数数。

假设我们想对数组中的每个元素做一些事情。一种方法是设置一个指向整数的指针，该指针从指向第一个元素开始：

```
int *p = &a[0];
```

& （&） 运算符与取消引用相反;它采用变量（必须是变量，而不是任意表达式）作为其操作数，并返回存储该变量的内存地址。因此，对于我们的示例数组，&a[0] 的值为 1000;&a[2] 的值为 1008。但是，如前所述，我们不能只将 1000 分配给 p，因为 p 是指针类型的变量，而 1000 是整数类型的值。（无论如何，在实践中，我们不知道编译器决定将数组放在哪个地址。

给定这个指向 a 的第一个元素的指针 p，我们如何到达下一个元素？C 提供了两个相似但不同的符号来描述这个过程：

```
p+1 /* 计算指向下一个元素的指针的值 */ p++ /* 相同，但将 p 替换为新的指针值 */
```

但这不应该是 p+4 吗？数组的下一个元素的地址比 p 中的地址大 4，而不仅仅是 1 大。

但是 p+1 是正确的。C 的指针算术考虑了指针目标的大小。因此，如果将 p 声明为指向整数的指针，则 p+1 或 p++ 实际上会在 p 的值上加 4。但是，如果 p 是指向字符的指针，则 p+1 和 p++ 将指针加 1。同样，如果 p 是双精度的指针，则递增指针将增加 8，以此类推。

[稍微棘手的例子：argv++ 是什么意思？由于 argv 被声明为 char **argv，因此读取速度过快可能会表明 argv++ 会加 1。但只有当声明是 char *argv 时，这才是正确的。额外的星号表示 argv 的类型是指针到指针，因此其目标（指针）的宽度为 4 个字节，因此 argv++ 向 argv 加了 4 个字节。