CS 61C Lecture Notes Week 2

Topic: C: Intro, Pointers, Arrays

Reading: Kernighan & Ritchie, Chapters 1–5

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:

• **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.

As one example, consider the history of number representation. Last week you learned about the now-universal convention that integers are represented in computer hardware using twos complement 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 $2^3 = 8$ 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 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.]

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 are 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.

• 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.

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       operating systems, compilers
                       hardware
```

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) 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."

In the computer industry, by the way, the symbols K (kilo) and M (mega) don't really mean 1,000 and 1,000,000; they mean 1024 and whatever 1024 squared is — 2^{10} and 220. "64K" is actually somewhat more than 65,000.

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 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.)

[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 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.]

C pitfalls

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

• 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

```
+ 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 doesn't happen *before* the dereferencing. 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.]

• **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 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;
```

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.]

• `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 '+':
    ...
    ...
    ...
    break;

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

  /* other cases */

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

(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 '+':
    ...
    ...
    ...
    break;

  /* other cases */

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

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`.

- `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.

[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 */

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.

• `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 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,                    same-file scope,
                    infinite extent                 infinite extent

STATIC not used     local scope,                    whole-program scope,
                    local extent                    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.]

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.

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.]

Arrays and Pointers

• **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.]

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:

```
    <--- 32 bits ---->
 ^  +----------------+
 |  |                |
 |  |                |
2^30|                |
 |  |                |
 |  |                |
 V  +----------------+
```

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

```
    <-8 bits->
 ^  +--------+
 |  |        |
 |  |        |
 |  |        |
 |  |        |
 |  |        |
 |  |        |
2^32|        |
 |  |        |
 |  |        |
 |  |        |
 |  |        |
 |  |        |
 |  |        |
 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 2^0=1, 2^1=2, 2^2=4, 2^3=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.

• **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 2^0 to 2^7 (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.]

• **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 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.

• **`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 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 */
```

- **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`.]