Chapter 5: Pointers and Arrays

Page 93: Pointers are often thought to be the most difficult aspect of C. It’s true that many people have various problems with pointers, and that many programs founder on pointer-related bugs. Actually, though, many of the problems are not so much with the pointers per se but rather with the memory they point to, and more specifically, when there isn’t any valid memory which they point to. As long as you’re careful to ensure that the pointers in your programs always point to valid memory, pointers can be useful, powerful, and relatively trouble-free tools. (In these notes, we’ll be emphasizing techniques for ensuring that pointers always point where they should.)

If you haven’t worked with pointers before, they’re bound to be a bit baffling at first. Rather than attempting a complete definition (which probably wouldn’t mean anything, either) up front, I’ll ask you to read along for a few pages, withholding judgment, and after we’ve seen a few of the things that pointers can do, we’ll be in a better position to appreciate what they are.

Section 5.1: Pointers and Addresses

If you like to use concrete examples and to think about exactly what’s going on at the machine level, you’ll want to know how many bytes are occupied by shorts, longs, pointers, etc. It’s equally possible, though, to understand pointers at a more abstract level, thinking about them only in terms of boxes and arrows, as in the figures on pages 96, 98, 104, 107, and 114-5. (Not worrying about the exact size in bytes basically means not worrying about how big the boxes are.) The figure at the bottom of page 93 is probably the least pretty pointer picture in the whole book; don’t worry if it doesn’t mean much to you.

When we say that a pointer holds an “address,” and that unary & is the “address of” operator, our language is of course influenced by the fact that the underlying hardware assigns addresses to memory locations, but again, it is not necessary (nor necessarily desirable) to think about actual machine addresses when working with pointers. Thinking about the machine addresses can make certain aspects of pointers easier to understand, but doing so can also make certain mistakes and misunderstandings easier. In particular, a pointer in C is more than just an address; as we’ll see on the next page, a pointer also carries the notion of what type of data it points to.

Page 94: The presentation on this page is going to seem very artificial at first. At best, you’re going to say, “This makes sense, but what’s it for?” In fact, it is artificial, and no real program would ever do meaningless little pointer operations such as are embodied in the example on this page. However, this is the traditional way to introduce pointers from scratch, and once we’ve moved past it, we’ll be able to talk about some more meaningful uses of pointers, and to forget about these artificial ones. (Once we’re
done talking about the traditional, artificial introduction on page 94, we’ll also attempt a slightly more elaborate, slightly less traditional, slightly more meaningful parallel introduction, so stay tuned.)

**Deep sentence:** The declaration of the pointer ip,

\[
\text{int } \text{*ip;}
\]

is intended as a mnemonic; it says that the expression \text{*ip is an int.}

We’ll have more to say about this sentence in a bit.

As an even more traditional, even less meaningful, even simpler example, we could say

\[
\begin{align*}
\text{int } i &= 1; & \text{// an integer } & \\
\text{int } \text{*ip; } &= & \text{// a pointer-to-int } & \\
\text{ip } &= & \&i; & \text{// ip points to } i \text{ } \\
\text{printf}(\text{"%d\n", } \text{*ip}); & \text{// prints } i, \text{ which is } 1 \text{ } \\
\text{*ip } &= & 5; & \text{// sets } i \text{ to } 5 \text{ } \\
\end{align*}
\]

(The obvious questions are, “if you want to print i, or set it to 5, why not just do it? Why mess around with this `pointer' thing?” More on that in a minute.)

The unary \& and * operators are complementary. Given an object (i.e. a variable), \& generates a pointer to it; given a pointer, * “returns” the value of the pointed-to object. “Returns” is in quotes because, as you may have noticed in the examples, you’re not restricted to fetching values via pointers: you can also store values via pointers. In an assignment like

\[
\text{*ip } = 0;
\]

the subexpression \text{*ip is conceptually “replaced” by the object which ip points to, and since *ip appears on the left-hand side of the assignment operator, what happens to the pointed-to object is that it gets assigned to.}

One of the things that’s hard about pointers is simply talking about what’s going on. We’ve been using the words “return” and “replace” in quotes, because they don’t quite reflect what’s actually going on, and we’ve been using clumsy locutions like “fetch via pointers” and “store via pointers.” There is some jargon for referring to pointer use; one word you’ll often see is dereference, a term which, though its derivation is suspect, is used to mean “follow a pointer to get at, and use, the object it points to.” Thus, we sometimes call unary * the “pointer dereferencing operator,” and we may say that the expressions

\[
\text{printf}(\text{"%d\n", } \text{*ip});
\]

and

\[
\text{*ip } = 5;
\]

both “dereference the pointer ip.” We may also talk about indirecting on a pointer: to indirect on a pointer is again to follow it to see what it points to; and * may also be called the “pointer indirection operator.”
Our examples of pointers so far have been, admittedly, artificial and rather meaningless. Let’s try a slightly more realistic example. In the previous chapter, we used the routines atoi and atof to convert strings representing numbers to the actual numbers represented. Often the strings were typed by the user, and read with getline. As you may have noticed, neither atoi nor atof does any validity or error checking: both simply stop reading when they reach a character that can’t be part of the number they’re converting, and if there aren’t any numeric characters in the string, they simply return 0. (For example, atoi("49er") is 49, and atoi("three") is 0, and atof("1.2.3") is 1.2.) These attributes make atoi and atof easy to write and easy (for the programmer) to use, but they are not the most user-friendly routines possible. A good user interface would warn the user and prompt again in case of invalid, non-numeric input.

Suppose we were writing a simple inventory-control system. For each part stored in our warehouse, we might record the part number, location, and number of parts on hand. For simplicity, we’ll assume that the location is always a simple bin number.

Somewhere in the inventory-control program, we might find the variables

```c
int part_number;
int location;
int number_on_hand;
```

and there might be a routine that lets the user enter any of these numbers. Suppose that there is another variable,

```c
int which_entry;
```

which indicates which of the three numbers is being entered (1 for part_number, 2 for location, or 3 for number_on_hand). We might have code like this:

```c
char instring[30];
switch (which_entry) {
    case 1:
        printf("enter part number:\n");
        getline(instring, 30);
        part_number = atoi(instring);
        break;
    case 2:
        printf("enter location:\n");
        getline(instring, 30);
        location = atoi(instring);
        break;
    case 3:
        printf("enter number on hand:\n");
        getline(instring, 30);
        number_on_hand = atoi(instring);
        break;
}
```

Suppose that we now begin to add a bit of rudimentary verification to the input routines. The first case might look like
case 1:
    do {
        printf("enter part number:
\n");
getline(instring, 30);
        if(!isdigit(instring[0]))
            continue;
        part_number = atoi(instring);
    } while (part_number == 0);
    break;

    If the first character is not a digit, or if atoi returns 0, the code goes around the loop another time,
and prompts the user again, in hopes that the user will type some proper numeric input this time. (The
tests for numeric input are not sufficient, nor even wise if 0 is a possible input value, as it presumably is
for number on hand. In fact, the two tests really do the same thing! But please overlook these faults. If
you’re curious, you can learn about a new ANSI function, strtol, which is like atoi but gives you a bit
more control, and would be a better routine to use here.)

    The code fragment above is for just one of the three input cases. The obvious way to perform the
same checking for the other two cases would be to repeat the same code two more times, changing the
prompt string and the name of the variable assigned to (location or number_on_hand instead of
part_number). Duplicating the code is a nuisance, though, especially if we later come up with a better
way to do input verification (perhaps one not suffering from the imperfections mentioned above). Is
there a better way?

    One way would be to use a temporary variable in the input loop, and then set one of the three real
variables to the value of the temporary variable, depending on which_entry:

    int temp;
    do {
        printf("enter the number:
\n");
getline(instring, 30);
        if(!isdigit(instring[0]))
            continue;
        temp = atoi(instring);
    } while (temp == 0);
    switch (which_entry) {
    case 1:
        part_number = temp;
        break;
    case 2:
        location = temp;
        break;
    case 3:
        number_on_hand = temp;
        break;
    }

    Another way, however, would be to use a pointer to keep track of which variable we’re setting. (In
this example, we’ll also get the prompt right.)
char instring[30];
int *numpointer;
char *prompt;

switch (which_entry) {
    case 1:
        numpointer = &part_number;
prompt = "part number";
break;
    case 2:
        numpointer = &location;
prompt = "location";
break;
    case 3:
        numpointer = &number_on_hand;
prompt = "number on hand";
break;
}

do {
    printf("enter %s:\n", prompt);
    getline(instring, 30);
    if(!isdigit(instring[0]))
        continue;
    *numpointer = atoi(instring);
} while (*numpointer == 0);

The idea here is that prompt is the prompt string and numpointer points to the particular numeric value we’re entering. That way, a single input verification loop can print any of the three prompts and set any of the three numeric variables, depending on where numpointer points. (We won’t officially see character pointers and strings until section 5.5, so don’t worry if the use of the prompt pointer seems new or inexplicable.)

This example is, in its own ways, quite artificial. (In a real inventory-control program, we’d obviously need to keep track of many parts; we couldn’t use single variables for the part number, location, and quantity. We probably wouldn’t really have a which_entry variable telling us which number to prompt for, and we’d do the numeric validation quite differently. We might well do numeric entry and validation in a separate function, removing this need for the pointers.) However, the pointer aspect of this example--using a pointer to refer to one of several different things, so that one generic piece of code can access any of the things--is a very typical (i.e. realistic) use of pointers.

There’s one nuance of pointer declarations which deserves mention. We’ve seen that

    int *ip;

declares the variable ip as a pointer to an int. We might look at that declaration and imagine that int * is the type and ip is the name of the variable being declared. (Actually, so far, these assumptions are both true.) We might therefore imagine that a more “obvious” way of writing the declaration would be

    int* ip;
This would work, but it is misleading, as we’ll see if we try to declare two int pointers at once. How shall we do it? If we try

```c
int* ip1, ip2; /* WRONG */
```

we don’t succeed; this would declare `ip1` as a pointer-to-int, but `ip2` as an int (not a pointer). The correct declaration for two pointers is

```c
int *ip1, *ip2;
```

As the authors said in the middle of page 94, the intent of pointer (and in fact all) declarations is that they give little miniature expressions indicating what type a certain use of the variables will have. The declaration

```c
int *ip1;
```

doesn’t so much say that `ip` is a pointer-to-int; it says that `*ip` is an int. (To be sure, `ip` is a pointer-to-int.) In the declaration

```c
int *ip1, *ip2;
```

both `*ip1` and `*ip2` are ints; so `ip1` and `ip2` are both pointers-to-int. You’ll hear this aspect of C declarations referred to as “declaration mimics use.” If it bothers you, or if you think you might accidentally write things like

```c
int *ip1, ip2;
```

then to stay on the safe side you might want to get in the habit of writing declarations on separate lines:

```c
int *ip1;
int *ip2;
```

I promised to point out the safe techniques for ensuring that pointers always point where they should. The examples in this section, which have all involved pointers pointing to single variables, are relatively safe; a single variable is not a very risky thing to point to, so code like the examples in this section is relatively unlikely to go awry and result in invalid pointers. (One potential problem, though, which we’ll talk more about later, is that since local, “automatic” variables are automatically deallocated when the function containing them returns, any pointer to a local variable also becomes invalid. Therefore, a function which returns a pointer must never return a pointer to one of its own local variables, and it would also be invalid to take a pointer to a local variable and assign it to a global pointer variable.)

**Section 5.2: Pointers and Function Arguments**

*Page 95*: This section discusses a very common use of pointers: setting things up so that a function can modify values in its caller, or return values, via its arguments. Remember that, normally, C passes arguments by value, and that if a function modifies one of its arguments, it modifies only its local copy,
not the value in the caller. (Normally, this is a good thing; having a function which inadvertently assigns to its arguments and hence inadvertently modifies a value in its caller can be a source of obscure bugs in languages which don’t use call-by-value.) However, what happens if a function wants to modify a value in its caller, and its caller wants to let it? How can a function return two values? (A function’s formal return value is always a single value.)

The answer to both questions is that a function can declare a parameter which is a pointer. The caller then passes a pointer to (that is, the “address of”) a variable in the caller which is to be modified or which is to receive the second return value. In fact, we’ve seen examples of this already: getline returns the length of the line it reads as well as the line itself, and the getop routine in the calculator example in section 4.3 returned both a code for an operator and a string representing the full text of the operator. (We needed that string when the operator was ‘0’ indicating numeric input, so that the string could return the full numeric input.) Though we didn’t say so at the time, we were actually using pointers in these examples. (We’ll explore the relationship between arrays and pointers, which made this possible, in section 5.3.)

With all of this in mind, make sure that you understand why the swap example on page 95 would not work, and how and why the swap example on page 96 does work, and what the figure on page 96 shows.

The swap example demonstrated a function which modified some variables (a and b) in its caller. The getint example demonstrates how to return two values from a function by returning one value as the normal function return value and the other one by writing to a pointer. (There is no fundamental difference, though, between “modifying a variable in the caller” and “returning a value by writing to a pointer”; these are just two applications of pointer parameters.)

The version of getint on page 97 is somewhat complicated because it allows free-form input, that is, the values need only be separated by whitespace or punctuation; they are not restricted to being one per line or anything. (C source code is also free-form in this regard; see page 4 of chapter 1 of these notes.) To see more clearly the essence of what getint is supposed to do, imagine for a moment that the input is restricted to one value per line, as in the “primitive calculator” example on page 72 of section 4.2. In that case, we might use the following simpler (i.e. more primitive) code:

```c
int getint(int *pn)
{
    char line[20];
    if (getline(line, 20) <= 0)
        return EOF;
    *pn = atoi(line);
    return 1;
}
```

The getint function on page 97 is documented as returning nonzero for a valid number and 0 for something other than a number. Our stripped-down version does not, and as it happens, the example code at the bottom of page 96 does not make use of the valid/invalid distinction. Can you see a way to rewrite the code at the bottom of page 96 to fill in the cells of the array with only valid numbers?

You might also notice, again from the code at the bottom of page 96, that & need not be restricted to single, simple variables; it can take the address of any data object, in this case, one cell of the array. Just as for all of C’s other operators, & can be applied to arbitrary expressions, although it is restricted to
expressions which represent addressable objects. Expressions like &1 or &(2+3) are meaningless and illegal.

You may remember a discussion from section 1.5.1 on page 16 of how C’s getchar routine is able to return all possible characters, plus an end-of-file indication, in its single return value. Why does getint need two return values? Why can’t it use the same trick that getchar does?

The examples in this section are again relatively safe. The pointers have all been parameters, and the callers have passed pointers (that is, the “addresses” of) their own, properly-allocated variables. That is, code like

```c
int a = 1, b = 2;
swap(&a, &b);
```

and

```c
int a;
getint(&a);
```

is correct and quite safe.

Something to beware of, though, is the temptation to inadvertently pass an uninitialized pointer variable (rather than the “address” of some other variable) to a routine which expects a pointer. We know that the getint routine expects as its argument a pointer to an int in which it is to store the integer it gets. Suppose we took that description literally, and wrote

```c
int *ip; /* a pointer to an int */
getint(ip);
```

Here we have in fact passed a pointer-to-int to getint, but the pointer we passed doesn’t point anywhere! When getint writes to (”dereferences”) the pointer, in an expression like *pn = 0, it will scribble on some random part of memory, and the program may crash. When people get caught in this trap, they often think that to fix it they need to use the & operator:

```c
getint(&ip); /* WRONG */
```

or maybe the * operator:

```c
getint(*ip); /* WRONG */
```

but these go from bad to worse. (If you think about them carefully, &ip is a pointer-to-pointer-to-int, and *ip is an int, and neither of these types matches the pointer-to-int which getint expects.) The correct usage for now, as we showed already, is something like

```c
int a;
getint(&a);
```

In this case, a is an honest-to-goodness, allocated int, so when we generate a pointer to it (with &a) and call getint, getint receives a pointer that does point somewhere.
Section 5.3: Pointers and Arrays

Page 97: For some people, section 5.3 is evidently the hardest section in this book, or even if they haven’t read this book, the most confusing aspect of the language. C introduces a novel and, it can be said, elegant integration of pointers and arrays, but there are a distressing number of ways of misunderstanding arrays, or pointers, or both. Take this section very slowly, learn the things it does say, and don’t learn anything it doesn’t say (i.e. don’t make any false assumptions).

It’s not necessarily true that “the pointer version will in general be faster”; efficiency is (or ought to be) a secondary concern when considering the use of pointers.

Page 98: On the top half of this page, we aren’t seeing anything we haven’t seen before. We already knew (or should have known) that the declaration int a[10]; declares an array of ten contiguous int’s numbered from 0 to 9. We saw on page 94 and again on page 96 that & can be used to take the address of one cell of an array.

What’s new on this page are first the nice pictures (and they are nice pictures; I think they’re the right way of thinking about arrays and pointers in C) and the definition of pointer arithmetic. If the phrase “then by definition pa+1 points to the next element” alarms you; if you hadn’t known that pa+1 points to the next element; don’t worry. You hadn’t known this, and you aren’t expected even to have suspected it: the reason that pa+1 points to the next element is simply that it’s defined that way, as the sentence says. Furthermore, subtraction works in an exactly analogous way: If we were to say

pa = &a[5];

then *(pa-1) would refer to the contents of a[4], and *(pa-i) would refer to the contents of the location i elements before cell 5 (as long as i <= 5).

Note furthermore that we do not have to worry about the size of the objects pointed to. Adding 1 to a pointer (or subtracting 1) always means to move over one object of the type pointed to, to get to the next element. (If you’re too worried about machine addresses, or the actual address values stored in pointers, or the actual sizes of things, it’s easy to mistakenly assume that adding or subtracting 1 adds or subtracts 1 from the machine address, but as we mentioned, you don’t have to think at this low level. We’ll see in section 5.4 how pointer arithmetic is actually scaled, automatically, by the size of the object pointed to, but we don’t have to worry about it if we don’t want to.)

Deep sentence: The meaning of “adding 1 to a pointer,” and by extension, all pointer arithmetic, is that pa+1 points to the next object, and pa+i points to the i-th object beyond pa.

This aspect of pointers—that arithmetic works on them, and in this way—is one of several vital facts about pointers in C. On the next page, we’ll see the others.

Page 99: Deep sentences: The correspondence between indexing and pointer arithmetic is very close. By definition, the value of a variable or expression of type array is the address of element zero of the array.

This is a fundamental definition, which we’ll now spend several pages discussing.
Don’t worry too much yet about the assertion that “pa and a have identical values.” We’re not surprised about the value of pa after the assignment \texttt{pa = \&a[0];} we’ve been taking the address of array elements for several pages now. What we don’t know—we’re not yet in a position to be surprised about it or not—is what the “value” of the array \texttt{a} is. What is the value of the array \texttt{a}?

In some languages, the value of an array is the entire array. If an array appears on the right-hand sign of an assignment, the entire array is assigned, and the left-hand side had better be an array, too. C does not work this way; C never lets you manipulate entire arrays.

In C, by definition, the value of an array, when it appears in an expression, is a pointer to its first element. In other words, the value of the array \texttt{a} simply is \texttt{\&a[0]}. If this statement makes any kind of intuitive sense to you at this point, that’s great, but if it doesn’t, please just take it on faith for a while. This statement is a fundamental (in fact the fundamental) definition about arrays and pointers in C, and if you don’t remember it, or don’t believe it, then pointers and arrays will never make proper sense. (You will also need to know another bit of jargon: we often say that, when an array appears in an expression, it decays into a pointer to its first element.)

Given the above definition, let’s explore some of the consequences. First of all, though we’ve been saying

\texttt{pa = \&a[0];}

we could also say

\texttt{pa = a;}

because by definition the value of \texttt{a} in an expression (i.e. as it sits there all alone on the right-hand side) is \texttt{\&a[0]}. Secondly, anywhere we’ve been using square brackets [ ] to subscript an array, we could also have used the pointer dereferencing operator *. That is, instead of writing

\texttt{i = a[5];}

we could, if we wanted to, instead write

\texttt{i = *(a+5);}

Why would this possibly work? How could this possibly work? Let’s look at the expression \texttt{*(a+5)} step by step. It contains a reference to the array \texttt{a}, which is by definition a pointer to its first element. So \texttt{*(a+5)} is equivalent to \texttt{*(\&a[0]+5)}. To make things clear, let’s pretend that we’d assigned the pointer to the first element to an actual pointer variable:

\texttt{int *pa = \&a[0];}

Now we have \texttt{*(a+5)} is equivalent to \texttt{*(\&a[0]+5)} is equivalent to \texttt{*(pa+5)}. But we learned on page 98 that \texttt{*(pa+5)} is simply the contents of the location 5 cells past where \texttt{pa} points to. Since \texttt{pa} points to \texttt{a[0]}, \texttt{*(pa+5)} is \texttt{a[5]}. Thus, for whatever it’s worth, any time you have an array subscript \texttt{a[i]}, you could write it as \texttt{*(a+i)}. 
The idea of the previous paragraph isn’t worth much, because if you’ve got an array `a`, indexing it using the notation `a[i]` is considerably more natural and convenient than the alternate `*(a+i)`. The significant fact is that this little correspondence between the expressions `a[i]` and `*(a+i)` holds for more than just arrays. If `pa` is a pointer, we can get at locations near it by using `*(pa+i)`, as we learned on page 98, but we can also use `pa[i]`. This time, using the “other” notation (array instead of pointer, when we thought we had a pointer) can be more convenient.

At this point, you may be asking why you can write `pa[i]` instead of `*(pa+i)`. You may be wondering how you’re going to remember that you can do this, or remember what it means if you see it in someone else’s code, when it’s such a surprising fact in the first place. There are several ways to remember it; pick whichever one suits you:

1. It’s an arbitrary fact, true by definition; just memorize it.

2. If, for an array `a`, instead of writing `a[i]`, you can also write `*(a+i)` (as we proved a few paragraphs back); then it’s only fair that for a pointer `pa`, instead of writing `*(pa+i)`, you can also write `pa[i]`.

3. Deep sentence: “In evaluating `a[i]`, C converts it to `*(a+i)` immediately; the two forms are equivalent.”

4. An array is a contiguous block of elements of a particular type. A pointer often points to a contiguous block of elements of a particular type. Therefore, it’s very handy to treat a pointer to a contiguous block of elements as if it were an array, by saying things like `pa[i]`.

5. [This is the most radical explanation, though it’s also the most true; but if it offends your sensibilities or only seems to make things more confusing, please ignore it.] When you said `a[i]`, you weren’t really subscripting an array at all, because an array like `a` in an expression always turns into a pointer to its first element. So the array subscripting operator `[]` always finds itself working on pointers, and it’s a simple identity (another definition) that `pa[i]` is `*(pa+i)`.

(But do pick at least one reason to remember this fact, as it’s a fact you’ll need to remember; expressions like `pa[i]` are quite common.)

The authors point out that “There is one difference between an array name and a pointer that must be kept in mind,” and this is quite true, but note very carefully that there is in fact every difference between an array and a pointer. When an array name appears in most expressions, it turns into a pointer (to the array’s first element), but that does not mean that the array is a pointer. You may hear it stated that “an array is just a constant pointer,” and this is a convenient explanation, but it is a simplified and potentially misleading explanation.

With that said, do make sure you understand why `a=pa` and `a++` (where `a` is an array) cannot mean anything.

*Deep sentence: When an array name is passed to a function, what is passed is the location of the initial element.*
Though perhaps surprising, this sentence doesn’t say anything new. A function call, and more importantly, each of its arguments, is an expression, and in an expression, a reference to an array is always replaced by a pointer to its first element. So given

```c
int a[10];
f(a);
```

it is not the entire array `a` that is passed to `f` but rather just a pointer to its first element. For an example closer to the text on page 99, given

```c
char string[] = "Hello, world!";
int len = strlen(string);
```

it is not the entire array `string` that is passed to `strlen` (recall that C never lets you do anything with a string or an array all at once), but rather just a pointer to its first element.

We now realize that we’ve been operating under a gentle fiction during the first four chapters of the book. Whenever we wrote a function like `getline` or `getop` which seemed to accept an array of characters, and whenever we thought we were passing arrays of characters to these routines, we were actually passing pointers. This explains, among other things, how `getline` and `getop` were able to modify the arrays in the caller, even though we said that call-by-value meant that functions can’t modify variables in their callers since they receive copies of the parameters. When a function receives a pointer, it cannot modify the original pointer in the caller, but it can definitely modify what the pointer points to.

If that doesn’t make sense, make sure you appreciate the full difference between a pointer and what it points to! It is entirely possible to modify one without modifying the other. Let’s illustrate this with an example. If we say

```c
char a[] = "hello";
char b[] = "world";
```

we’ve declared two character arrays, `a` and `b`, each containing a string. If we say

```c
char *p = a;
```

we’ve declared `p` as a pointer-to-char, and initialized it to point to the first character of the array `a`. If we then say

```c
*p = 'H';
```

we’ve modified what `p` points to. We have not modified `p` itself. After saying `*p = 'H';` the string in the array `a` has been modified to contain "Hello".

If we say

```c
p = b;
```

on the other hand, we have modified the pointer `p` itself. We have not really modified what `p` points to. In a sense, “what `p` points to” has changed—it used to be the string in the array `a`, and now it’s the string in the array `b`. But saying `p = b` didn’t modify either of the strings.
Since, as we’ve just seen, functions never receive arrays as parameters, but instead always receive pointers, how have we been able to get away with defining functions (like getline and getop) which seemed to accept arrays? The answer is that whenever you declare an array parameter to a function, the compiler pretends that you actually declared a pointer. (It does this mostly so that we can get away with the "gentle fiction" of pretending that we can pass arrays to functions.)

When you see a statement like “char s[]; and char *s; are equivalent” (as in fact you see at the top of page 100), you can be sure that (and you must remember that) it is only function formal parameters that are being talked about. Anywhere else, arrays and pointers are quite different, as we’ve discussed.

Expressions like p[-1] (at the end of section 5.3) may be easier to understand if we convert them back to the pointer form *(p + -1) and thence to *(p-1) which, as we’ve seen, is the object one before what p points to.

With the examples in this section, we begin to see how pointer manipulations can go awry. In sections 5.1 and 5.2, most of our pointers were to simple variables. When we use pointers into arrays, and when we begin using pointer arithmetic to access nearby cells of the array, we must be careful never to go off the end of the array, in either direction. A pointer is only valid if it points to one of the allocated cells of an array. (There is also an exception for a pointer just past the end of an array, which we’ll talk about later.) Given the declarations

```c
int a[10];
int *pa;
```

the statements

```c
pa = a;
*pa = 0;
*(pa+1) = 1;
pa[2] = 2;
pa = &a[5];
*pa = 6;
*(pa-1) = 4;
pa[1] = 6;
pa = &a[9];
*pa = 9;
pa[-1] = 8;
```

are all valid. These statements set the pointer pa pointing to various cells of the array a, and modify some of those cells by indirecting on the pointer pa. (As an exercise, verify that each cell of a that receives a value receives the value of its own index. For example, a[6] is set to 6.)

However, the statements

```c
pa = a;
*(pa+10) = 0; /* WRONG */
*(pa-1) = 0; /* WRONG */
pa = &a[5];
*(pa+10) = 0; /* WRONG */
pa = &a[10];
*pa = 0;    /* WRONG */
```

And
are all invalid. The first examples set \( pa \) to point into the array \( a \) but then use overly-large offsets (+10, -1) which end up trying to store a value outside of the array \( a \). The statements in the last set of examples set \( pa2 \) to point outside of the array \( a \). Even though no attempt is made to access the nonexistent cells, these statements are illegal, too. Finally, the code

```c
int a[10];
int *pa, *pa2;
pa = &a[5];
pa2 = pa + 10;    /* WRONG */
pa2 = pa - 10;    /* WRONG */
*pa2 = 0;        /* WRONG */
```

would be very wrong, because it not only computes a pointer to the nonexistent 15th cell of a 10-element array, but it also tries to store something there.

**Section 5.4: Address Arithmetic**

This section is going to get pretty hairy. Some of it talks about things we’ve already seen (adding integers to pointers); some of it talks about things we need to learn (comparing and subtracting pointers); and some of it talks about a rather sophisticated example (a storage allocator). Don’t worry if you can’t follow all the details of the storage allocator, but do read along so that you can pick up the other new points. (In other words, make sure you read from “Zero is the sole exception” in the middle of page 102 to “that is, the string length” on page 103, and also the last paragraph on page 103.)

What is a storage allocator for? So far, we’ve used pointers to point to existing variables and arrays, which the compiler allocated for us. But eventually, we may want to allocate data structures (arrays, and others we haven’t seen yet) of a size which we don’t know at compile time. Earlier, we spoke briefly about a hypothetical inventory-management system, which recorded information about each part stored in a warehouse. How many different parts could there be? If we used fixed-size arrays, there would be a fixed upper limit on the number of parts we could enter into the system, and we’d be annoyed if that limit were reached. A better solution is not to allocate a fixed array at compile time, but rather to use a run-time storage allocator to allocate memory for the data structures used to describe each part. That way, the number of parts which the system can hold is limited only by available memory, not on any static limit built into the program. Using a storage allocator to allocate memory at run time in this way is called dynamic allocation.

However, dynamic memory allocation is where C programming can really get tricky, because you the programmer are responsible for most aspects of it, and there are plenty of things you can do wrong (e.g. not allocate quite enough memory, accidentally keep using it after you deallocate it, have random invalid pointers pointing everywhere, etc.). Therefore, we won’t be talking about dynamic allocation for a while, which is why you can skim over the storage allocator in this section for now.

*Page 102:* The first new piece of information in this section (which you’ll need to remember even if you’re not following the details of the storage allocator example) is the introduction of the “null pointer.”
So far, all of our pointers have pointed somewhere, and we’ve cautioned about pointers which don’t. To help us distinguish between pointers which point somewhere and pointers which don’t, there is a single, special pointer value we can use, which is guaranteed not to point anywhere. When a pointer doesn’t point anywhere, we can set it to this value, to make explicit the fact that it doesn’t point anywhere.

This special pointer value is called the null pointer. The way to set a pointer to this value is to use a constant 0:

```c
int *ip = 0;
```

The 0 is just a shorthand; it does not necessarily mean machine address 0. To make it clear that we’re talking about the null pointer and not the integer 0, we often use a macro definition like

```c
#define NULL 0
```

so that we can say things like

```c
int *ip = NULL;
```

(If you’ve used Pascal or LISP, the nil pointer in those languages is analogous.)

In fact, the above #definition of NULL has been placed in the standard header file `<stdio.h>` for us (and in several other standard header files as well), so we don’t even need to #define it. I agree completely with the authors that using NULL instead of 0 makes it more clear that we’re talking about a null pointer, so I’ll always be using NULL, too.

Just as we can set a pointer to NULL, we can also test a pointer to see if it’s NULL. The code

```c
if(p != NULL)
    *p = 0;
else printf("p doesn’t point anywhere\n");
```

tests p to see if it’s non-NULL. If it’s not NULL, it assumes that it points somewhere valid, and writes a 0 there. Otherwise (i.e. if p is the null pointer) the code complains.

Though we can use null pointers as markers to remind ourselves of which of our pointers don’t point anywhere, it’s up to us to do so. It is not guaranteed that all uninitialized pointer variables (which obviously don’t point anywhere) are initialized to NULL, so if we want to use the null pointer convention to remind ourselves, we’d best explicitly initialize all unused pointers to NULL. Furthermore, there is no general mechanism that automatically checks whether a pointer is non-null before we use it. If we think that a pointer might not point anywhere, and if we’re using the convention that pointers that don’t point anywhere are set to NULL, it’s up to us to compare the pointer to NULL to decide whether it’s safe to use it.

The next new piece of information in this section (which we’ve already alluded to) is pointer comparison. You can compare two pointers for equality or inequality (== or !): they’re equal if they point to the same place or are both null pointers; they’re unequal if they point to different places, or if
one points somewhere and one is a null pointer. If two pointers point into the same array, the relational
comparisons <, <=, >, and => can also be used.

Page 103: \(\ldots n\) is scaled according to the size of the objects \(p\) points to, which is determined by the
declaration of \(p\). If an int is four bytes, for example, the int will be scaled by four.

This is something we’ve seen already, but may only confuse the issue. We’ve said informally that in
the code

```c
int a[10];
int *pa = &a[0];
*(pa+1) = 1;
```

\(pa\) contains the “address” of the int object \(a[0]\), but we’ve discouraged thinking about this address
as an actual machine memory address. We’ve said that the expression \(pa+1\) moves to the next int in the
array (in this case, \(a[1]\)). Thinking at this abstract level, we don’t even need to worry about any “scaling
by the size of the objects pointed to.”

If we do look at a lower, machine level of addressing, we may learn that an int occupies some
number of bytes (usually two or four), such that when we add 1 to a pointer-to-int, the machine address
is actually increased by 2 or 4. If you like to consider the situation from this angle, you’re welcome to,
but if you don’t, you certainly don’t have to. If you do start thinking about machine addresses and sizes,
make extra sure that you remember that C does do the necessary scaling for you. Don’t write something
like

```c
int a[10];
int *pa = &a[0];
*(pa+sizeof(int)) = 1;
```

where \(\text{sizeof(int)}\) is the size of an int in bytes, and expect it to access \(a[1]\).

Since adding an int to a pointer gives us another pointer:

```c
int a[10];
int *pal = &a[0];
int *pa2 = pal + 5;
```

we might wonder if we can rearrange the expression

```c
pa2 = pal + 5
```

to get

```c
pa2 - pal 5
```

(where this is no longer a C assignment, we’re just wondering if we can subtract \(\text{pal}\) from \(\text{pa2}\), and
what the result might be). The answer is yes: just as you can compare two pointers which point into the
same array, you can subtract them, and the result is, naturally enough, the distance between them, in
cells or elements.
Section 5.5: Character Pointers and Functions

Page 104: Since text strings are represented in C by arrays of characters, and since arrays are very often manipulated via pointers, character pointers are probably the most common pointers in C.

Deep sentence: C does not provide any operators for processing an entire string of characters as a unit.

We’ve said this sort of thing before, and it’s a general statement which is true of all arrays. Make sure you understand that in the lines

```c
char *pmessage;
pmessage = "now is the time";
pmessage = "hello, world";
```

we are doing is assigning two pointers, not copying two entire strings.

At the bottom of the page is a very important picture. We’ve said that pointers and arrays are different, and here’s another illustration. Make sure you appreciate the significance of this picture: it’s probably the most basic illustration of how arrays and pointers are implemented in C.

We also need to understand the two different ways that string literals like "now is the time" are used in C. In the definition

```c
char amessage[] = "now is the time";
```

the string literal is used as the initializer for the array amessage. amessage is here an array of 16 characters, which we may later overwrite with other characters if we wish. The string literal merely sets the initial contents of the array. In the definition

```c
char *pmessage = "now is the time";
```

on the other hand, the string literal is used to create a little block of characters somewhere in memory which the pointer pmessage is initialized to point to. We may reassign pmessage to point somewhere else, but as long as it points to the string literal, we can’t modify the characters it points to.

As an example of what we can and can’t do, given the lines

```c
char amessage[] = "now is the time";
char *pmessage = "now is the time";
```

we could say

```c
amessage[0] = ‘N’;
```

to make amessage say "Now is the time". But if we tried to do

```c
pmessage[0] = ‘N’;
```

(which, as you may recall, is equivalent to `*pmessage = 'N'), it would not necessarily work; we’re not allowed to modify that string. (One reason is that the compiler might have placed the “little block of characters” in read-only memory. Another reason is that if we had written

```c
char *pmessage = "now is the time";
char *qmessage = "now is the time";
```

the compiler might have used the same little block of memory to initialize both pointers, and we wouldn’t want a change to one to alter the other.)

*Deep sentence:* The first function is `strcpy(s,t)`, which copies the string `t` to the string `s`. It would be nice just to say `s=t` but this copies the pointer, not the characters.

This is a restatement of what we said above, and a reminder of why we’ll need a function, `strcpy`, to copy whole strings.

*Page 105:* Once again, these code fragments are being written in a rather compressed way. To make it easier to see what’s going on, here are alternate versions of `strcpy`, which don’t bury the assignment in the loop test. First we’ll use array notation:

```c
void strcpy(char s[], char t[]) {
    int i;
    for(i = 0; t[i] != '\0'; i++)
        s[i] = t[i];
    s[i] = '\0';
}
```

Note that we have to manually append the `\0` to `s` after the loop. Note that in doing so we depend upon `i` retaining its final value after the loop, but this is guaranteed in C, as we learned in Chapter 3.

Here is a similar function, using pointer notation:

```c
void strcpy(char *s, char *t) {
    while(*t != '\0')
        *s++ = *t++;
    *s = '\0';
}
```

Again, we have to manually append the `\0`. Yet another option might be to use a do/while loop.

All of these versions of `strcpy` are quite similar to the copy function we saw on page 29 in section 1.9.

*Page 106:* The version of `strcpy` at the top of this page is my least favorite example in the whole book. Yes, many experienced C programmers would write `strcpy` this way, and yes, you’ll eventually need to be able to read and decipher code like this, but my own recommendation against this kind of cryptic code is strong enough that I’d rather not show this example yet, if at all.
We need strcmp for about the same reason we need strcpy. Just as we cannot assign one string to another using =, we cannot compare two strings using ==. (If we try to use ==, all we’ll compare is the two pointers. If the pointers are equal, they point to the same place, so they certainly point to the same string, but if we have two strings in two different parts of memory, pointers to them will always compare different even if the strings pointed to contain identical sequences of characters.)

Note that strcmp returns a positive number if s is greater than t, a negative number if s is less than t, and zero if s compares equal to t. “Greater than” and “less than” are interpreted based on the relative values of the characters in the machine’s character set. This means that ‘a’ < ‘b’, but (in the ASCII character set, at least) it also means that ‘B’ < ‘a’. (In other words, capital letters will sort before lowercase letters.) The positive or negative number which strcmp returns is, in this implementation at least, actually the difference between the values of the first two characters that differ.

Note that strcmp returns 0 when the strings are equal. Therefore, the condition

```c
if(strcmp(a, b))
    do something...
```

doesn’t do what you probably think it does. Remember that C considers zero to be “false” and nonzero to be “true,” so this code does something if the strings a and b are unequal. If you want to do something if two strings are equal, use code like

```c
if(strcmp(a, b) == 0)
    do something...
```

(There’s nothing fancy going on here: strcmp returns 0 when the two strings are equal, so that’s what we explicitly test for.)

To continue our ongoing discussion of which pointer manipulations are safe and which are risky or must be done with care, let’s consider character pointers. As we’ve mentioned, one thing to beware of is that a pointer derived from a string literal, as in

```c
char *pmessage = "now is the time";
```

is usable but not writable (that is, the characters pointed to are not writable.) Another thing to be careful of is that at any time you copy strings, using strcpy or some other method, you must be sure that the destination string is a writable array with enough space for the string you’re writing. Remember, too, that the space you need is the number of characters in the string you’re copying, plus one for the terminating ‘\0’.

For the above reasons, all three of these examples are incorrect:

```c
char *p1 = "Hello, world!";
char *p2;
strcpy(p2, p1); /* WRONG */

char *p = "Hello, world!";
char a[13];
strcpy(a, p); /* WRONG */

char *p3 = "Hello, world!";
```
char *p4 = "A string to overwrite";
strcpy(p4, p3);  /* WRONG */

In the first example, p2 doesn’t point anywhere. In the second example, a is a writable array, but it
doesn’t have room for the terminating ‘\0’. In the third example, p4 points to memory which we’re not
allowed to overwrite. A correct example would be

cchar *p = "Hello, world!";  
char a[14];  
strcpy(a, p);

(Another option would be to obtain some memory for the string copy, i.e. the destination for strcpy,
using dynamic memory allocation, but we’re not talking about that yet.)

Page 106 continued (bottom): Expressions like *p++ and *--p may seem cryptic at first sight, but they’re
actually analogous to array subscript expressions like a[i++] and a[--i], some of which we were using
back on page 47 in section 2.8.

Section 5.6: Pointer Arrays; Pointers to Pointers

Page 107: Deep sentence: Since pointers are variables themselves, they can be stored in arrays just as
other variables can.

This is just one aspect of the generality of C’s data types, which we’ll be seeing in the next few
sections.

We’ve used a recursive definition of “expression”: a constant or variable is an expression, an
expression in parentheses is an expression, an expression plus an expression is an expression, etc. There
are obviously an infinite number of expressions, of arbitrary complexity. In exactly the same way, there
are an infinite number of data types in C. We’ve already seen the basic data types: int, char, double, etc.
But then we have the derived data types such as array-of-char and pointer-to-int and function-
returning-double. So we can say that for any type, array-of-type is another type, and pointer-to-type is
another type, and function-returning-type is another type. Once we’ve said that, we can see that there
is also the possibility of arrays of pointers, and arrays of arrays, and functions returning pointers, and
even (in section 5.11, though this is a deeper topic) pointers to functions. (The only possibilities that C
doesn’t support are functions returning arrays, and arrays of functions, and functions returning
functions.)

Make sure you understand why an integer is something that can be “compared or moved in a single
operation,” but that a string (that is, an array of char) is not. Then, realize that a pointer is also
something that can be “compared or moved in a single operation.” (Actually, though, the string
comparisons we’ll be doing are not single operations.) From time to time you’ll hear me caution you not
to worry too much about certain aspects of efficiency. Here, it’s true that the overhead of copying entire
strings from one place to another, a character at a time (which is the overhead we’ll be getting around
by manipulating pointers instead) can be significant, but that’s not the only concern: once we’re
comfortable with the idea, manipulating pointers will be somewhat easier on us, too. (Copying lots of
characters around is a nuisance, and it can also be dangerous, if the destination isn’t big enough or isn’t
in the right place.)
Don’t worry about the “one long character array” that the “lines to be sorted are stored end-to-end in.” Instead, look at the picture at the bottom of page 107, which shows the pointers that might be set up after reading the lines

```
defghi
jklmnopqrst
abc
```

On the left are the pointers before sorting, and on the right are the pointers after sorting. The three strings have not been moved, but by reshuffling the pointers, the three pointers in order now point to the lines

```
abc
defghi
jklmnopqrst
```

**Page 108:** Once again, we see a nice simple decomposition of the problem, which might seem deceptively simple except that when problems are decomposed in simple ways like this, and then implemented faithfully, they really can be this simple. Deferring the sorting step is an excellent idea, especially if we didn’t quite follow the details of the sorting functions in the previous chapter. (Actually, in practice, we can usually defer the sorting step forever, since there’s often a general-purpose sort routine provided for us somewhere. C is no exception: a qsort function is a required part of its standard library. For the most part, the only people who have to write sort routines are programming students and the few people who get stuck implementing system functions.)

The main program at the bottom of page 108 looks a bit more elaborate than the pseudocode at the top of the page, but the essence of the program is the three calls to readlines, qsort, and writelines. Everything else is declarations, plus an error message which is printed if readlines is for some reason not able to read the input. Eventually, you should be able to understand why all of the various declarations are required, but you can skim over them at first.

**Page 109:** The readlines function first calls our old friend getline to read each line into a local array, line. On page 29 in section 1.9, we saw a program for finding the longest line in the input: it read each line into a local array line, and kept a copy of the longest line in a second array longest. In that program, it didn’t matter that the input array line was continually overwritten with each new input line, and that most lines (except the longest one) were lost and forgotten. Here, however, we do need to save all of the input lines somewhere, so that we can sort them and print them later.

The lines are saved by calling alloc, a function which we wrote in section 5.4 but may have skimmed over. alloc allocates n bytes of new memory for something which we need to save. Each time we read another line, we call alloc to allocate some new memory to store it, then call strcpy to copy the line from the line array to the newly allocated memory. This way, it’s okay that the next line is read into the same line array; we save each line, as it’s read, into its own little alloc’ed piece of memory.

Note that memory allocated with a routine such as alloc persists, just as global and static variables do; it does not disappear when the function that allocated it returns.

Hopefully you’re getting used to reading compressed condition statements by now, because here’s another doozy:
if (nlines >= maxlines || (p == alloc(len)) == NULL)

This line checks to make sure we have enough room to store the new line we just read. We need two things: (1) a slot in the lineptr array to store the pointer, and (2) space allocated by alloc to store the line itself. If we don't have either of these things, we return -1, indicating that we ran out of memory. We don't have a slot in the lineptr array if we've already read maxlines lines, and we don't have room to store the line itself if alloc returns NULL. The subexpression (p = alloc(len)) == NULL is equivalent in form to other assign-and-test combinations we've been using involving getchar and getline: it assigns alloc's return value to p, then compares it to NULL.

Normally, we might be suspicious of the call alloc(len). Why? Remember that strings are always terminated by '\0', so the space required to store a string is always one more than the the number of characters in it. Normally, we'll call things like alloc(len + 1), and accidentally calling alloc(len) is usually a bug. Here, it happens to be okay, because before we copy the line to the newly-allocated memory, we strip the newline '\n' from the end of it, by overwriting it with '\0', hence making the string one shorter than len. (Why is the last character in line, namely the '\n', at line[len-1], and not line[len]?)

The fragments

if (nlines >= maxlines ... and

lineptr[nlines++] = p;

deserve some attention. These represent a common way of filling in an array in C. nlines always holds the number of lines we've read so far (it's another invariant). It starts out as 0 (we haven't read any lines yet) and it ends up as the total number of lines we've read. Each time we read a new line, we store the pointer in the slot indexed by the previous value of nlines, which is what we want, because arrays are 0-based in C. The first time through the loop, nlines is 0, so we store a pointer to the first line in lineptr[0], and then increment nlines to 1. If nlines ever becomes equal to maxlines, we've filled in all the slots of the array, and we can't use any more (even though, at that point, the highest-filled cell in the array is lineptr[maxlines-1], which is the last cell in the array, again because arrays are 0-based). We test for this condition by checking nlines >= maxlines, as a little measure of paranoia. The test nlines == maxlines would also work, but if we ever accidentally introduce a bug into the program such that we fill past the last slot without noticing it, we wouldn't want to keep on filling farther and farther past the end.

Deep sentences: ...lineptr is an array of MAXLINES elements, each element of which is a pointer to a char. That is, lineptr[i] is a character pointer...

We can see that lineptr[i] has to be a character pointer, by looking at two things: in the function readlines, the line

lineptr[nlines++] = p;

has a character pointer on the right-hand side, and the only thing we can assign a character pointer to is another character pointer. Also, in the function writelines, in the line
printf("%s\n", lineptr[i]);

printf’s %s format expects a pointer to a character, so that’s what lineptr[i] had better be.

Note that writelines prints a newline after each line, since newlines were stripped out of the input lines by readlines.

Don’t worry too much about the discussion at the bottom of page 109. We saw in section 5.3 that due to the “strong relationship” between pointers and arrays, it is always possible to manipulate an array using pointer-like notation, and to manipulate a pointer using array-like notation. Since lineptr is an array, it is possible to manipulate it using pointer-like notation, but since what it’s an array of is other pointers, it can start to get a bit confusing. Though many programmers do write things like

printf("%s\n", *lineptr++);

and though this is correct code, and though one should probably understand it to have a 100% complete understanding of C, I’ve decided that code like that is just a bit too hard to follow, and I’d always write (perhaps more pedestrian and mundane) things like

printf("%s\n", lineptr[i]);

or

printf("%s\n", lineptr[i++]);

Page 110: Since I didn’t ask you to follow the qsort example in section 4.10 in complete detail, I won’t ask you to work through this one completely, either. But if you compare the code here to the code on pages 87-88, you will see that the only significant differences are that the variables and arrays containing the things being sorted have been changed from int to char * (pointer-to-char), and the comparison

if (v[i] < v[left])

has been changed to

if (strcmp(v[i], v[left]) < 0)

Section 5.7: Multi-dimensional Arrays

Page 111: The month_day function is another example of a function which simulates having multiple return values by using pointer parameters. month_day is declared as void, so it has no formal return value, but two of its parameters, pmonth and pday, are pointers, and it fills in the locations pointed to by these two pointers with the two values it wants to “return.” One line of the definition of month_day on page 111 is cut off in all printings I have seen: it should read

void month_day(int year, int yearday, int *pmonth, int *pday)

As we’ve said, although any nonzero value is considered “true” in C, the built-in relational and Boolean operators always “return” 0 or 1. Therefore, the line
int leap = year%4 == 0 && year%100 != 0 || year%400 == 0;

sets leap to 1 or 0 ("true" or "false") depending on the condition

year%4 == 0 && year%100 != 0 || year%400 == 0

which is the condition for leap years in the Gregorian calendar. (It’s a little-known fact that century years are not leap years unless they are also divisible by 400. Thus, 2000 will be a leap year.) The 1/0 value that leap receives is what the authors are referring to when they say that “the arithmetic value of a logical expression... can be used as a subscript of the array daytab.” This line could also have been written

```c
int leap;
if (year%4 == 0 && year%100 != 0 || year%400 == 0)
    leap = 1;
else
    leap = 0;
```

or

```c
int leap = (year%4 == 0 && year%100 != 0 || year%400 == 0) ? 1 : 0;
```

Page 112: The daytab array holds small integers (in the range 0-31), so it can legally be made an array of char, though whether this is a legitimate use is a question of style.

Deep sentence: In C, a two-dimensional array is really a one-dimensional array, each of whose elements is an array.

Earlier we said that “array-of-type is another type,” and here we must believe it: since array-of-type is a type, array-of-(array-of-type) is yet another type.

The statement that “Elements are stored by rows, so the rightmost subscript, or column, varies fastest as elements are accessed in storage order” probably won’t make much sense unless you’ve done a lot of work with other languages, such as FORTRAN, which do have true multi-dimensional arrays. It’s pretty arbitrary what you call a “row” and what you call a “column”; the most important thing to know is which subscript goes with which dimension. If you have

```c
int a[10][20];
```

then in the reference a[i][j], i can range from 0 to 9 and j can range from 0 to 19. In other words, you might write

```c
for (i = 0; i < 10; i++)
    for (j = 0; j < 20; j++)
        do something with a[i][j]
```

We also want to know what a actually is. Is it an array of 10 arrays, each of size 20, or is it an array of 20 arrays, each of size 10? There are other ways of convincing ourselves of the answer, but for now let’s just say that the “closer” dimensions are closer to what a is. Therefore, a is first an array of size 10, and
what it is is an array of arrays of 20 ints. This also tells us that if we ever refer to a[i] (without a second subscript), then we’re referring to just one of those 10 arrays (of size 20) in its entirety.

When we look back at the initialization of the daytab array on page 111, everything lines up. daytab is defined as

```c
char daytab[2][13]
```

and we can see from the initializer that there are two (sub)arrays, each of size 13. (We can also see that there is some justification for saying that the first subscript refers to “rows” and the second to “columns.”)

The authors illustrate one way of dealing with C’s 0-based arrays when you have an algorithm that really wants to treat an array as if it were 1-based. Here, rather than remembering to subtract one from the 1-based month number each time, they chose to waste a “column” of the array, and declare it one larger than necessary, so that they could refer to subscripts from [1] to [12].

One last note about the initialization of daytab: you may have seen code in other programming books that kept an array of the cumulative days of all the months:

```c
{0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334, 365}
```

Precomputing an array like that might make things a tiny bit easier on the computer (it wouldn’t have to loop through the entire array each time, as it does in the day_of_year function), but it makes it considerably harder to see what the numbers mean, and to verify that they are correct. The simple table of individual month lengths is much clearer, and if the computer has to do a bit more grunge work, well, that’s what computers are for. As explained in another book co-authored by Brian Kernighan:

A cumulative table of days must be calculated by someone and checked by someone else. Since few people are familiar with the number of days up to the end of a particular month, neither writing nor checking is easy. But if instead we use a table of days per month, we can let the computer count them for us. (“Let the machine do the dirty work.”)

The bottom of page 112 begins to get confusing. The “number of rows” of an array like daytab “is irrelevant” when passed to a function such as the hypothetical f because the compiler doesn’t need to know the number of rows when calculating subscripts. It does need to know the number of columns or “width,” because that’s how it knows that the second element on the second row of a 10-column array is actually 12 cells past the beginning of the array, which is essentially what it needs to know when it goes off and actually accesses the array in memory. But it doesn’t need to know how long the overall array is, as long as we promise not to run off the end of it, and that’s always up to us. (This is why we haven’t specified the array sizes in the definitions of functions such as getline on pages 29 and 69, or atoi on pages 43, 61, and 73, or readlines on page 109, although we did carry the array size as a separate argument to getline and readlines, to assist us in our promise not to run off the end.)

The third version of f on page 112 comes about because of the “gentle fiction” involving array parameters. We learned on page 99 that functions don’t really receive arrays as parameters; they receive arrays (since any array passed by the caller decayed immediately to a pointer). On page 39 we wrote a strlen function as
int strlen(char s[])

but on page 99 we rewrote it as

int strlen(char *s)

which is closer to the way the compiler sees the situation. (In fact, when we write int strlen(char s[]),
the compiler essentially rewrites it as int strlen(char *s) for us.) In the same way, a function declared as

f(int daytab[][13])

can be rewritten by us (or if not, is rewritten by the compiler) to

f(int (*daytab)[13])

which declares the daytab parameter as a pointer-to-array-of-13-ints. Here we see two things: (1) the rewrite which changes an array parameter to a pointer parameter happens only once (we end up with a pointer to an array, not a pointer to a pointer), and (2) the syntax for pointers to arrays is a bit messy, because of some required extra parentheses, as explained in the text.

If this seems obscure, don’t worry about it too much; just declare functions with array parameters matching the arrays you call them with, like

f(int daytab[2][13])

and let the compiler worry about the rewriting.

Deep sentence: More generally, only the first dimension (subscript) of an array is free; all the others have to be specified.

This just says what we said already: when declaring an array as a function parameter, you can leave off the first dimension because it is the overall length and not knowing it causes no immediate problems (unless you accidentally go off the end). But the compiler always needs to know the other dimensions, so that it knows how the rows and columns line up.

Section 5.8: Initialization of Pointer Arrays

Page 113: This section is short and sweet, and there are only two things I feel the need to comment on. The sentence “The characters of the i-th string are placed somewhere” simply refers to the fact that string literals always work that way (except when they’re used as array initializers, as explained on page 104). We don’t really care where the characters are, as long as we can keep hold of a pointer to them.

The other thing to notice is that the month_name function does verify that its argument is valid. If it didn’t check n against the boundary values 1 and 12, what would happen if we called month_name(123)?

Section 5.9: Pointers vs. Multi-dimensional Arrays
Actually, some people (and not just newcomers) are sometimes confused about the difference between a one-dimensional array and a single pointer, too; moving to two-dimensional arrays, arrays of pointers, and pointers to pointers only makes things worse. (But don’t lose heart: if you pay attention and keep your head screwed on straight, you should be able to keep the differences clearly in mind.)

The adjective “syntactically” in the paragraph at the bottom of the page is significant: after saying

```c
int *b[10];
```

an immediate reference to `b[3][4]` would not be completely legal. It wouldn’t be a syntax error or anything, but when the compiler tried to fetch the third pointer and then the fourth integer pointed to, it would go off into deep space, because there isn’t a third pointer yet and it doesn’t point anywhere.

You might want to draw a picture of the data structures that would result “[a]ssuming that each element of `b` does point to a twenty-element array,” and verify that there are “200 ints set aside, plus ten cells for the pointers.” (The picture will be similar to the one on the next page.)

Actually, I’m not sure if having rows of different lengths is the only important advantage of using a pointer array. Another is that the size of the arrays (as we’ll see later) can be decided at run-time; another is that the pointers make certain manipulations easier (such as the sorting example we worked through in section 5.6).

Page 114: Do study the pictures on this page carefully, and make sure you understand the representations of the name and aname arrays and how they differ. (You might want to refer back to the similar discussion of pmessage and amessage on page 104 in section 5.5.)

Section 5.10: Command-line Arguments

Page 115: The picture at the top of page 115 doesn’t quite match the declaration

```c
char *argv[]
```

it’s actually a picture of the situation declared by

```c
char **argv
```

which is what main actually receives. (The array parameter declaration `char *argv[]` is rewritten by the compiler to `char **argv`, in accordance with the discussion in sections 5.3 and 5.8.) Also, the “0” at the bottom of the array is just a representation of the null pointer which conventionally terminates the argv array. (Normally, you’ll never encounter the terminating null pointer, because if you think of argv as an array of size argc, you’ll never access beyond argv[argc-1].)

The loop

```c
for (i = 1; i < argc; i++)
```

looks different from most loops we see in C (which either start at 0 and use <, or start at 1 and use <=). The reason is that we’re skipping argv[0], which contains the name of the program.
The expression

    printf("%s", argv[i], (i < argc-1) ? " " : "")

is a little nicety to print a space after each word (to separate it from the next word) but not after the last word. (The nicety is just that the code doesn’t print an extra space at the end of the line.) It would also be possible to fold in the following printf of the newline:

    printf("%s", argv[i], (i < argc-1) ? " " : \\
          "\n");

As I mentioned in comment on the bottom of page 109, it’s not necessary to write pointer-incrementing code like

    while(--argc > 0)
        printf("%s", *++argv, (argc > 1) ? " " : "");

if you don’t feel comfortable with it. I used to try write code like this, because it seemed to be what everybody else did, but it never sat well, and it was always just a bit too hard to write and to prove correct. I’ve reverted to simple, obvious loops like

    int argi;
    char *sep = "";

    for (argi = 1; argi < argc; argi++) {
        printf("%s", sep, argv[argi]);
        sep = " ";
    }
    printf("\n");

Often, it’s handy to have the original argc and argv around later, anyway. (This loop also shows another way of handling space separators.)

Page 116: shows a simple improvement on the matching-lines program first presented on page 69; page 117 adds a few more improvements. The differences between page 69 and page 116 are that the pattern is read from the command line, and strstr is used instead of strindex. The difference between page 116 and page 117 is the handling of the -n and -x options. (The next obvious improvement, which we’re not quite in a position to make yet, is to allow a file name to be specified on the command line, rather than always reading from the standard input.)

Page 117: Several aspects of this code deserve note.

The line

    while (c = *++argv[0])

is not in error. (In isolation, it might look like an example of the classic error of accidentally writing = instead of == in a comparison.) What it’s actually doing is another version of a combined set-and-test: it assigns the next character pointed to by argv[0] to c, and compares it against ‘\0’. You can’t see the comparison against ‘\0’, because it’s implicit in the usual interpretation of a nonzero expression as “true.” An explicit test would look like this:
while ((c = *++argv[0]) != '\0')

argv[0] is a pointer to a character in a string; ++argv[0] increments that pointer to point to the next character in the string; and *++argv[0] increments the pointer while returning the next character pointed to. argv[0] is not the first string on the command line, but rather whichever one we’re looking at now, since elsewhere in the loop we increment argv itself.

Some of the extra complexity in this loop is to make sure that it can handle both

-x -n

And

-xn

In pseudocode, the option-parsing loop is

```pseudocode
for (each word on the command line)
  if (it begins with '-')
    for (each character c in that word)
      switch (c)
      ...
```

For comparison, here is another way of writing effectively the same loop:

```c
int argi;
char *p;
for (argi = 1; argi < argc && argv[argi][0] == '-'; argi++)
  for (p = &argv[argi][1]; *p != '\0'; p++)
    switch (*p) {
      case 'x':
        ...
```

This uses array notation to access the words on the command line, but pointer notation to access the characters within a word (more specifically, a word that begins with ‘-’). We could also use array notation for both:

```c
int argi, chari;
for (argi = 1; argi < argc && argv[argi][0] == '-'; argi++)
  for (chari = 1; argv[argi][chari] != '\0'; chari++)
    switch (argv[argi][chari]) {
      case 'x':
        ...
```

In either case, the inner, character loop starts at the second character (index [1]), not the first, because the first character (index [0]) is the ‘-‘.

It’s easy to see how the -n option is implemented. If -n is seen, the number flag is set to 1 (a.k.a. “true”), and later, in the line-matching loop, each time a line is printed, if the number flag is true, the line number is printed first. It’s harder to see how -x works. An except flag is set to 1 if -x is present, but how is except used? It’s buried down there in the line
if ((strstr(line, *argv) != NULL) != except)

What does that mean? The subexpression

(strstr(line, *argv) != NULL)

is 1 if the line contains the pattern, and 0 if it does not. except is 0 if we should print matching lines, and 1 if we should print non-matching lines. What we’ve actually implemented here is an “exclusive OR,” which is “if A or B but not both.” Other ways of writing this would be

```c
int matched = (strstr(line, *argv) != NULL);
if (matched && !except || !matched && except) {
    if (number)
        printf("%ld:", lineno);
    printf("%s", line);
    found++;
}
```

Or

```c
int matched = (strstr(line, *argv) != NULL);
if (except ? !matched : matched) {
    if (number)
        printf("%ld:", lineno);
    printf("%s", line);
    found++;
}
```

or

```c
int matched = (strstr(line, *argv) != NULL);
if (!except) {
    if (matched) {
        if (number)
            printf("%ld:", lineno);
        printf("%s", line);
        found++;
    }
} else {
    if (!matched) {
        if (number)
            printf("%ld:", lineno);
        printf("%s", line);
        found++;
    }
}
```

There’s clearly a tradeoff: the last version is in some sense the most clear (and the most verbose), but it ends up repeating the line-number printing and any other processing which must be done for found lines. Therefore, the compressed, perhaps slightly more cryptic forms are better: some day, it’s a virtual certainty that more processing will be added for printed lines (for example, if we’re searching multiple files, we’ll want to print the filename for matching lines, too), and if the printing is duplicated in two places, it’s far too likely that we’ll overlook that fact and add the new code in only one place.
One last point on the pattern-matching program: it’s probably clearer to declare a pointer variable

    char *pat;

    and set it to the word from argv to be used as the search pattern (argv[1] or *argv, depending on
whether we’re looking at page 116 or 117), and then use that in the call to strstr:

    if (strstr(line, pat) != NULL ...