Chapter 6: Structures

Page 127: There’s one other piece of motivation behind structures that it’s useful to discuss. Suppose we didn’t have structures (or didn’t know what they were or how to use them). Suppose we wanted to implement payroll records. We might set up a bunch of parallel arrays, holding the names, mailing addresses, social security numbers, and salaries of all of our employees:

```c
char *name[100];
char *address[100];
long ssn[100];
float salary[100];
```

The idea here is that name[0], address[0], ssn[0], and salary[0] would describe one employee, array slots with subscript [1] would describe the second employee, etc. There are at least two problems with this scheme: first, if we someday want to handle more than 100 employees, we have to remember to change the size of several arrays. (Using a symbolic constant like

```c
#define MAXEMPLOYEES 100
```

would certainly help.)

More importantly, there would be no easy way to pass around all the information associated with a single employee. Suppose we wanted to write the function print_employee, which will print all the information associated with a particular employee. What arguments would this function take? We could pass it the index to use to retrieve the information from the arrays, but that would mean that all of the arrays would have to be global. We could pass the function an individual name, address, SSN, and salary, but that would mean that whenever we added a new piece of information to the database (perhaps next week we’ll want to keep track of employee’s shoe sizes), we would have to add another argument to the print_employee function, and change all of the calls. (Pretty soon, the number of arguments to the print_employee function would become unwieldy.) What we’d really like is a way to encapsulate all of the data about a single employee into a single data structure, so we could just pass that data structure around.

The right solution to this problem, in languages such as C which support the idea, is to define a structure describing an employee. We can make one array of these structures to describe all the employees, and we can pass around single instances of the structure where they’re needed.

Section 6.1: Basics of Structures
Don’t get too excited about the prospect of doing graphics in C—there’s no one standard or portable way of doing it, so the points and rectangles we’re going to be discussing must remain abstract for now (we won’t be able to plot them out).

Page 128: To summarize the syntax of structure declarations: A structure declaration has about four parts, most of them optional: the keyword struct, a structure tag (optional), a brace-enclosed list of declarations for the members (also called “fields” or “components”) of the structure (optional), and a list of variables of the new structure type (optional). The arrangement looks like this:

```
struct tag {
    member declarations
} declared variables;
```

Normally, a structure declaration defines either a tag and the members, or some variables based on an existing tag, or sometimes all three at once. That is, we might first declare a structure:

```
struct point { /* 1 */
    int x;
    int y;
};
```

and then some variables of that type:

```
struct point here, there; /* 2 */
```

Or, we could combine the two:

```
struct point { /* 3 */
    int x;
    int y;
} here, there;
```

The list of members (if present) describes what the new structure “looks like inside.” The list of variables (if present) is (obviously) the list of variables of this new type which we’re defining and which the rest of the program will use. The tag (if present) is just an arbitrary name for the structure type itself (not for any variable we’re defining). The tag is used to associate a structure definition (as in fragment 1) with a later declaration of variables of that same type (as in fragment 2).

One thing to beware of: when you declare the members of a structure without defining any variables, always remember the trailing semicolon, as shown in fragment 1 above. (If you forget it, the compiler will wait until the next thing it finds in your source file, and try to define that as a variable or function of the structure type.)

Section 6.2: Structures and Functions

In this section, we’ll begin playing with structures more or less as if they were ordinary variables such as we’ve been using all along (which they more or less are). As we’ll see, we can declare variables of structure type, declare functions which accept structures as parameters and return them, declare pointers to structures, take the address of a structure (creating a pointer-to-structure) with &, and assign structures.
Notice that when we declare something as “a structure type,” we always have to say which structure type, usually by using the struct tag. If we’ve set up a “point” structure as above, then to declare a variable of this type, we say

```c
struct point thepoint;
```

Both

```c
struct thepoint;  /* WRONG */
```

and

```c
point thepoint;  /* WRONG */
```

would be errors.

The above list of things the language lets us do with structures lets us keep them and move them around, but there isn’t really anything defined by the language that we can do with structures. It’s up to us to define any operations on structures, usually by writing functions. (The addpoint function on page 130 is a good example. It will make a bit more sense if you think of it as adding not isolated points, but rather vectors. [We can’t add Seattle plus Los Angeles, but we could add (two miles south, one mile east) plus (one mile east, two miles north)].)

Page 131: As an aside, how safe are the min() and max() macros defined at the top of page 131, with respect to the criteria discussed on pages 15 and 16 of the notes on section 4.11.2 (page 90 in the text)?

The precise meaning of the “shorthand” -> operator is that sp->m is, by definition, equivalent to (*sp).m, for any structure pointer sp and member m of the pointed-to structure.

Section 6.3: Arrays of Structures

Page 132: In the previous section we introduced pointers to structures and functions returning structures without fanfare. But now let’s pay attention to the fact that structures fit the pattern of the other types: a structure is a type, so we can have pointer-to-struct, array-of-struct, and function-returning-struct. (We can also say, following our ongoing pattern of recursive definitions, that for any list of types $t_1$, $t_2$, $t_3$, ..., we can make a new type

```c
struct tag {
   t1 m1;
   t2 m2;
   t3 m3;
   ...
};
```

which is a structure composed of members of those types.)

Page 134: We glossed over the binary search routine on page 58 in section 3.3, so we can skip the details of this one, too. This illustrates another benefit of breaking functionality out into functions, though: as long as you know what a function does, you can understand a program that it’s in without necessarily understanding all of it. In this case, binsearch searches an array tab, containing n cells of type
struct key, looking for one whose word field matches the parameter word. If it finds a matching cell, it returns its index in the array; otherwise, it returns -1.

The sizeof operator

Page 135: This may seem like an excessively roundabout or low-level way of finding the number of elements in an array, but it is the way it’s done in C, and it’s perfectly safe and straightforward once you get used to it. (I would, however, be hard-pressed to defend against the accusation that it’s a bit too low-level.)

Note that sizeof works on both type names (things like int, char *, struct key, etc.) and variables (strictly speaking, any expression). Parentheses are required when you’re using sizeof with a type name and optional when you’re using it with a variable or expression (just like return), but it’s safe to just always use parentheses.

sizeof returns the size counted in bytes, where the C definition of “byte” is “the size of a char.” In other words, sizeof(char) is always 1. (It turns out that it’s not necessarily the case, though, that a byte or a char is 8 bits.) When we start doing our own dynamic memory allocation (which will be pretty soon), we’ll always be needing to know the size of things so that we can allocate space for them, so it’s just as well that we’re meeting and getting used to the sizeof operator now.

The sentence “But the expression in the #define is not evaluated by the preprocessor” means that, as far as the preprocessor is concerned, the “value” of the macro NKEYS (like the value of any macro) is just a string of characters like

```
(sizeof(keytab) / sizeof keytab[0])
```

which it replaces wherever NKEYS is used, and which will then be evaluated by the compiler as usual, so it doesn’t matter that the preprocessor wouldn’t have known how to deal with the sizeof operator, or how big the keytab array or a struct key were.

A third way of defining NKEYS would be

```
#define NKEYS (sizeof(keytab) / sizeof *keytab)
```

Note that the definition of NKEYS depends on the definition of the keytab array (which appears on page 133), and both of them will have to precede the use of NKEYS in main on page 134. (Also, all three will have to be in the same source file, unless other steps are taken.)

Page 136: Notice that getword has a lot in common with the getop function of the calculator example (section 4.3, page 80).

Section 6.4: Pointers to Structures

The bulk of this section illustrates how to rewrite the binsearch function (which we’ve already been glossing over) in terms of pointers instead of arrays (an exercise which we’ve been downplaying). There are a few important points towards the end of the section, however.
Page 138: When we began talking about pointers and arrays, we said that it was important never to access outside of the defined and allocated bounds of an array, either with an out-of-range index or an out-of-bounds pointer. There is one exception: you may compute (but not access, or “dereference”) a pointer to the imaginary element which sits one past the end of an array. Therefore, a common idiom for accessing an array using a pointer looks like

```c
int a[10];
int *ip;
for (ip = &a[0]; ip < &a[10]; ip++)
...
```
or

```c
int a[10];
int *endp = &a[10];
int *ip;
for (ip = a; ip < endp; ip++)
...
```
The element a[10] does not exist (the allocated elements run from [0] to [9]), but we may compute the pointer &a[10], and use it in expressions like ip < &a[10] and endp = &a[10].

Deep sentence: Don’t assume, however, that the size of a structure is the sum of the sizes of its members.

If this isn’t the sort of thing you’d be likely to assume, you don’t have to remember the reason, which is mildly esoteric (having to do with memory alignment requirements).

Section 6.5: Self-referential Structures

Page 139: In section 4.10, we met recursive functions. Now, we’re going to meet recursively-defined data structures. Don’t throw up your hands: the two should be easier to understand in combination.

The mention of “quadratic running time” is tangential, but it’s a useful-enough concept that it might be worth a bit of explanation. If we were keeping a simple list (“linear array”) in order, each time we had a new word to install, we’d have to scan over the old list. On average, we’d have to scan over half the old list. (Even if we used binary search to find the position, we’d still have to move some part of the list to insert it.) Therefore, the more words that were in the list, the longer it would take to install each new word. It turns out that the running time of this linear insertion algorithm would grow as the square of the number of items in the list (that’s what “quadratically” means). If you doubled the size of the list, the running time would be four times longer. An algorithm like this may seem to work fine when you run it on small test inputs, but then when you run it on a real problem consisting of a thousand or ten thousand or a million words, it bogs down hopelessly.

A binary tree is a great way to keep a set of words (or other values) in sorted order. The definition of a binary tree is simply that, at each node, all items in the left subtree are less than the item at that node, and all items in the right subtree are greater. (Note that the top item in the left subtree is not necessarily immediately less than the item at that node or anything; the immediately-preceding item is merely down in the left subtree somewhere, along with all the rest of the preceding items. In the “now
is the time” example, the word “now” is neither the first, last, nor middle word in the sorted list; it’s merely the word that happened to be installed first. The word preceding it is “men”; the word following it is “of.” The first word in the sorted list is “aid,” and the last word is “to.”

The binary tree may not immediately seem like much of an improvement over the linear array—we still have to scan over part of the existing tree in order to insert each new word, and the time to add each new word will get longer as there are more words in the tree. But, if you do the math, it turns out that on average you have to scan over a much smaller part of the tree, and it’s not a simple fraction like half or one quarter, but rather the log (base two) of the number of items already in the tree. Furthermore, inserting a new node doesn’t involve reshuffling any old data. For these reasons, the running time of binary tree insertion doesn’t slow down nearly as badly as linear insertion does.

By the way, the reason that the word “binary” comes up so often is because it simply means “two.” The binary number system has two digits (0 and 1); a binary operator has two operands; binary search eliminates half (one over two) of the possibilities at each step; a binary tree has two subtrees at each node.

One other bit of nomenclature: the word “node” simply refers to one of the structures in a set of structures that is linked together in some way, and as we’re about to see, we’re going to use a set of linked structures to implement a binary tree. Just as we talk about a “cell” or “element” of an array, we talk about a “node” in a tree or linked list.

When we look at the description of the algorithm for finding out whether a word is already in the tree, we may begin to see why the binary tree is more efficient than the linear list. When searching through a linear list, each time we discard a value that’s not the one we’re looking for, we’ve only discarded that one value; we still have the entire rest of the list to search. In a binary tree, however, whenever we move down the tree, we’ve just eliminated half of the tree. (We might say that a binary tree is a data structure which makes binary search automatic.) Consider guessing a number between one and a hundred by asking “Is it 1? Is it 2? Is it 3?” etc., versus asking “Is it less than 50? Is it greater than 25? Is it less than 12?”

Page 140: Make sure you’re comfortable with the idea of a structure which contains pointers to other instances of itself. If you draw some little box-and-arrow diagrams for a binary tree, the idea should fall into place easily. (As the authors point out, what would be impossible would be for a structure to contain not a pointer but rather another entire instance of itself, because that instance would contain another, and another, and the structure would be infinitely big.)

Page 141: Note that addtree accepts as an argument the tree to be added to, and returns a pointer to a tree, because it may have to modify the tree in the process of adding a new node to it. If it doesn’t have to modify the tree (more precisely, if it doesn’t have to modify the top or root of the tree) it returns the same pointer it was handed.

Another thing to note is the technique used to mark the edges or “leaves” of the tree. We said that a null pointer was a special pointer value guaranteed not to point anywhere, and it is therefore an excellent marker to use when a left or right subtree does not exist. Whenever a new node is built, addtree initializes both subtree pointers (“children”) to null pointers. Later, another chain of calls to addtree may replace one or the other of these with a new subtree. (Eventually, both might be replaced.)
If you don’t completely see how addtree works, leave it for a moment and look at treeprint on the next page first.

The bottom of page 141 discusses a tremendously important issue: memory allocation. Although we only have one copy of the addtree function (which may call itself recursively many times), by the time we’re done, we’ll have many instances of the tnode structure (one for each unique word in the input). Therefore, we have to arrange somehow that memory for these multiple instances is properly allocated. We can’t use a local variable of type struct tnode in addtree, because local variables disappear when their containing function returns. We can’t use a static variable of type struct tnode in addtree, or a global variable of type struct tnode, because then we’d have only one node in the whole program, and we need many.

What we need is some brand-new memory. Furthermore, we have to arrange it so that each time addtree builds a brand-new node, it does so in another new piece of brand-new memory. Since each node contains a pointer (char *) to a string, the memory for that string has to be dynamically allocated, too. (If we didn’t allocate memory for each new string, all the strings would end up being stored in the word array in main on page 140, and they’d step all over each other, and we’d only be able to see the last word we read.)

For the moment, we defer the questions of exactly where this brand-new memory is to come from by defining two functions to do it. talloc is going to return a (pointer to a) brand-new piece of memory suitable for holding a struct tnode, and strdup is going to return a (pointer to a) brand-new piece of memory containing a copy of a string.

Page 142: treeprint is probably the cleanest, simplest recursive function there is. If you’ve been putting off getting comfortable with recursive functions, now is the time.

Suppose it’s our job to print a binary tree: we’ve just been handed a pointer to the base (root) of the tree. What do we do? The only node we’ve got easy access to is the root node, but as we saw, that’s not the first or the last element to print or anything; it’s generally a random node somewhere in the middle of the eventual sorted list (distinguished only by the fact that it happened to be inserted first). The node that needs to be printed first is buried somewhere down in the left subtree, and the node to print just before the node we’ve got easy access to is buried somewhere else down in the left subtree, and the node to print next (after the one we’ve got) is buried somewhere down in the right subtree. In fact, everything down in the left subtree is to be printed before the node we’ve got, and everything down in the right subtree is to be printed after. A pseudocode description of our task, therefore, might be

```plaintext
print the left subtree (in order)
print the node we’re at
print the right subtree (in order)
```

How can we print the left subtree, in order? The left subtree is, in general, another tree, so printing it out sounds about as hard as printing an entire tree, which is what we were supposed to do. In fact, it’s exactly as hard: it’s the same problem. Are we going in circles? Are we getting anywhere? Yes, we are: the left subtree, even though it is still a tree, is at least smaller than the full tree we started with. The same is true of the right subtree. Therefore, we can use a recursive call to do the hard work of printing the subtrees, and all we have to do is the easy part: print the node we’re at. The fact that the subtrees are smaller gives us the leverage we need to make a recursive algorithm work.
In any recursive function, it is (obviously) important to terminate the recursion, that is, to make sure that the function doesn’t recursively call itself forever. In the case of binary trees, when you reach a “leaf” of the tree (more precisely, when the left or right subtree is a null pointer), there’s nothing more to visit, so the recursion can stop. We can test for this in two different ways, either before or after we make the “last” recursive call:

```c
void treeprint(struct tnode *p)
{
    if(p->left != NULL)
        treeprint(p->left);
    printf("%4d %s\n", p->count, p->word);
    if(p->right != NULL)
        treeprint(p->right);
}
```

or

```c
void treeprint(struct tnode *p)
{
    if(p == NULL)
        return;
    treeprint(p->left);
    printf("%4d %s\n", p->count, p->word);
    treeprint(p->right);
}
```

Sometimes, there’s little difference between one approach and the other. Here, though, the second approach (which is equivalent to the code on page 142) has a distinct advantage: it will work even if the very first call is on an empty tree (in this case, if there were no words in the input). As we mentioned earlier, it’s extremely nice if programs work well at their boundary conditions, even if we don’t think those conditions are likely to occur.

(One more thing to notice is that it’s quite possible for a node to have a left subtree but not a right, or vice versa; one example is the node labeled “of” in the tree on page 139.)

Another impressive thing about a recursive treeprint function is that it’s not just a way of writing it, or a nifty way of writing it; it’s really the only way of writing it. You might try to figure out how to write a nonrecursive version. Once you’ve printed something down in the left subtree, how do you know where to go back up to? Our struct tnode only has pointers down the tree, there aren’t any pointers back to the “parent” of each node. If you write a nonrecursive version, you have to keep track of how you got to where you are, and it’s not enough to keep track of the parent of the node you’re at; you have to keep a stack of all the nodes you’ve passed down through. When you write a recursive version, on the other hand, the normal function-call stack essentially keeps track of all this for you.

We now return to the problem of dynamic memory allocation. The basic approach builds on something we’ve been seeing glimpses of for a few chapters now: we use a general-purpose function which returns a pointer to a block of n bytes of memory. (The authors presented a primitive version of such a function in section 5.4, and we used it in the sorting program in section 5.6.) Our problem is then reduced to (1) remembering to call this allocation function when we need to, and (2) figuring out how many bytes we need. Problem 1 is stubborn, but problem 2 is solved by the sizeof operator we met in section 6.3.
You don’t need to worry about all the details of the “digression on a problem related to storage allocators.” The vast majority of the time, this problem is taken care of for you, because you use the system library function malloc.

The problem of malloc’s return type is not quite as bad as the authors make it out to be. In ANSI C, the void * type is a “generic” pointer type, specifically intended to be used where you need a pointer which can be a pointer to any data type. Since void * is never a pointer to anything by itself, but is always intended to be converted (“coerced”) into some other type, it turns out that a cast is not strictly required: in code like

```
struct tnode *tp = malloc(sizeof(struct tnode));
```

or

```
return malloc(sizeof(struct tnode));
```

the compiler is willing to convert the pointer types implicitly, without warning you and without requiring you to insert explicit casts. (If you feel more comfortable with the casts, though, you’re welcome to leave them in.)

Page 143: strdup is a handy little function that does two things: it allocates enough memory for one of your strings, and it copies your string to the new memory, returning a pointer to it. (It encapsulates a pattern which we first saw in the readlines function on page 109 in section 5.6.) Note the +1 in the call to malloc! Accidentally calling malloc(strlen(s)) is an easy but serious mistake.

As we mentioned at the beginning of chapter 5, memory allocation can be hard to get right, and is at the root of many difficulties and bugs in many C programs. Here are some rules and other things to remember:

1. Make sure you know where things are allocated, either by the compiler or by you. Watch out for things like the local line array we’ve been tending to use with getline, and the local word array on page 140. When a function writes to an array or a pointer supplied by the caller, it depends on the caller to have allocated storage correctly. When you’re the caller, make sure you pass a valid pointer! Make sure you understand why

```
char *ptr;
getline(ptr, 100);
```

is wrong and can’t work. (For one thing: what does that 100 mean? If getline is only allowed to read at most 100 characters, where have we allocated those 100 characters that getline is not allowed to write to more of than?)

2. Be aware of any situations where a single array or data structure is used to store multiple different things, in succession. Think again about the local line array we’ve been tending to use with getline, and the local word array on page 140. These arrays are overwritten with each new line, word, etc., so if you need to keep all of the lines or words around, you must copy them immediately to allocated memory (as the line-sorting program on pages 108-9 in section 5.6 did, but as the longest line program on page 29 in section 1.9 and the pattern-
matching programs on page 69 in section 4.1 and pages 116-7 in section 5.10 did not have to do).

3. Make sure you allocate enough memory! If you allocate memory for an array of 10 things, don’t accidentally store 11 things in it. If you have a string that’s 10 characters long, make sure you always allocate 11 characters for it (including one for the terminating \0’).

4. When you free (deallocation) memory, make sure that you don’t have any pointers lying around which still point to it (or if you do, make sure not to use them any more).

5. Always check the return value from memory-allocation functions. Memory is never infinite: sooner or later, you will run out of memory, and allocation functions generally return a null pointer when this happens.

6. When you’re not using dynamically-allocated memory any more, do try to free it, if it’s convenient to do so and the program’s not just about to exit. Otherwise, you may eventually have so much memory allocated to stuff you’re not using any more that there’s no more memory left for new stuff you need to allocate. (However, on all but a few broken systems, all memory is automatically and definitively returned to the operating system when your program exits, so if one of your programs doesn’t free some memory, you shouldn’t have to worry that it’s wasted forever.)

Unfortunately, checking the return values from memory allocation functions (point 5 above) requires a few more lines of code, so it is often left out of sample code in textbooks, including this one. Here are versions of main and addtree for the word-counting program (pages 140-1 in the text) which do check for out-of-memory conditions:

```c
/* word frequency count */
main()
{
  struct tnode *root;
  char word[MAXWORD];

  root = NULL;
  while (getword(word, MAXWORD) != EOF) {
    if (isalpha(word[0])) {
      root = addtree(root, word);
      if(root == NULL) {
        printf("out of memory\n");
        return 1;
      }
    }
  }
  treeprint(root);
  return 0;
}

struct tnode *addtree(struct tnode *p, char *w)
{
  int cond;

  if (p == NULL) { /* a new word has arrived */
    p = talloc();/* make a new node */
if (p == NULL)
    return NULL;
p->word = strdup(w);
if (p->word == NULL) {
    free(p);
    return NULL;
}
p->count = 1;
p->left = p->right = NULL;
} else if ((cond = strcmp(w, p->word)) == 0)
    p->count++; /* repeated word */
else if (cond < 0) { /* less than: into left subtree */
    p->left = addtree(p->left, w);
    if(p->left == NULL)
        return NULL;
}
else { /* greater than: into right subtree */
    p->right = addtree(p->right, w);
    if(p->right == NULL)
        return NULL;
}
return p;
}

In practice, many programmers would collapse the calls and tests:

struct tnode *addtree(struct tnode *p, char *w)
{
    int cond;

    if (p == NULL) { /* a new word has arrived */
        if ((p = talloc()) == NULL)
            return NULL;
        if ((p->word = strdup(w)) == NULL) {
            free(p);
            return NULL;
        }
        p->count = 1;
        p->left = p->right = NULL;
    } else if ((cond = strcmp(w, p->word)) == 0)
        p->count++; /* repeated word */
    else if (cond < 0) { /* less than: into left subtree */
        if ((p->left = addtree(p->left, w)) == NULL)
            return NULL;
    }
    else { /* greater than: into right subtree */
        if ((p->right = addtree(p->right, w)) == NULL)
            return NULL;
    }
    return p;
}