Lecture 16

CSE 110

28 July 1992

1 Dynamic Memory Allocation

We are finally going to learn how to allocate memory on the fly, getting more when we need it, a subject not usually covered in introductory C courses.

1.1 `malloc`

We've already seen `strdup`, which finds memory somewhere and reserves it. How did `strdup` find that memory?

Chances are it called `malloc`. `malloc` finds memory by asking the operating system for a big bunch of memory and then parceling it out a little at a time as you ask for it.[1] It's `malloc` that takes care of recording how big each parcel is so that `free` can free it properly.

To call `malloc`, pass as an argument the number of bytes of space you want to allocate. `malloc` will reserve that many bytes and return a pointer to the memory it found. If it fails for any reason, such as because all the memory is already reserved, `malloc` returns (all together now) the `NULL` pointer.

1.2 `sizeof`

Frequently you want to allocate enough memory to store an object of a certain type; you need to be able to tell `malloc` how many bytes of space the object takes up so it knows how much to allocate. There's an operator in C which tells you how big an object is: `sizeof`.

[1] `malloc` does it this way, instead of going to the operating system every time, because operating system requests are very slow.

If *type* is the name of a type, the value of the expression `sizeof(`*type*`)` is the number of bytes needed to hold that type; for example, on our machines, `sizeof(int)` is 2 and `sizeof(double)` is 8.

If we want to get enough space for an array of 13 <`int`>s, we can do `malloc(13 * sizeof(int))` or even `malloc(sizeof(int[13]))`.[2]

There's another way to use `sizeof`: If e is an expression with type t, then the value of `sizeof(`e`)` is the same as the value of the expression `sizeof(`t`)`.

2 `structs`

We've seen how to use an array to store many values of the same type, and even to treat the collection of values as a unit. Now we'll look at the complementary type, the *structure*, which allows you to group several different object together as a unit.

2.1 Creating a New Structure Type

The typical example of a structure is this: You want to keep track of the employees in your organization. Each employee has a name (a string of no more than 50 characters), a social security number (a <`long int`>), and a salary (an <`int`>). You want to handle several such records.

You can create a new type, a `struct employee`, which keeps all of this information in one place. First, you define the new type:

```
 struct employee {
   char name[51];
   long int ssn;
   int salary;
} ;
```

When you write this in your program, it defines a new type, the type `struct employee`. A variable of type `struct employee` has three *members*: `name`, an <array of 50 `chars`>, `ssn`, a <`long int`>, and `salary`, an <`int`>.

`employee` is called the *tag* of the structure.

[2] `int[13]` is the name of the type <array of 13 `ints`>. To construct the name of a certain type, just write a declaration for a variable of that type, and then discard the variable name and the semicolon from the declaration.

2.2 Creating `struct` Variables

To declare a variables of type `struct employee`, just write:

```
struct employee smithers, marketing[12], *the_employee;
```

`smithers` is a `struct employee`; `marketing` is an array of twelve of these structures, and `the_employee` is a pointer to one of these structures.

2.3 The . Operator

Suppose `smithers` is a `struct employee` variable, as above. Suppose we want to store the information that Smithers' social security number is `314-15-9265`. Here's how we would do that:

```
smithers.ssn = 314159265;
```

The `.` operator is the *structure member* operator. Its left operand must be a structure of some kind; its right operand must be the name of one of the members of that structure. The entire expression refers to the specified member of the specified structure. So similarly,

```
if ( smithers.salary > 10000 )
  ...
```

asks if Smithers' salary exceeds ten thousand dollars.

Similarly, let's suppose the array `marketing`, which is an array of 12 `struct employee`s, has been initialized with the names, salaries, and social security numbers of the members of the marketing department. Here's how we'd compute the average salary in the marketing department:

```
long int totalsalary = 0;

for (i=0; i<12; i++)
  totalsalary += marketing[i].salary;
printf("Average salary is %f.\n", totalsalary/12.0);
```

`marketing` is an array of `struct employee`s, so `marketing[i]` is a `struct employee`, and `marketing[i].salary` is the salary member of that `struct employee`.

As a final example, this is some code which prints out the name and social security number of the highest-paid member of the marketing department:

```
int highest_salary = 0;
int i, highest_salary_index;

for (i=0; i<12; i++)
  if (marketing[i].salary > highest_salary) {
    highest_salary = marketing[i].salary;
    highest_salary_index = i;
  }

printf("The highest-paid employee is %s (%d).\n",
       marketing[highest_salary_index].name,
       marketing[highest_salary_index].ssn);
```

3 The Linked List

Suppose we want to write a program which reads an input file, counts the number of occurrences of each different word in the input, and then prints out a report of the form "`There were 27 occurrences of the word 'I', 53 occurrences of the word 'the'...`". You might want to have an array that will hold words and another that will hold counts, but that doesn't work, because the arrays have a fixed size, say 500 elements each, and if the input contains 501 different words, you're stumped. We need to have a data structure that can grow arbitrarily large if we need it to.

The simplest example of such a data structure is the *linked list*. A linked list is a collection of *nodes* in some order; each node contains some information about how to find the 'next' node, and perhaps some auxiliary information as well. The last node has some kind of marker that says it's last.

To keep track of a linked list, all we need to do is remember where the first node is. Then that first node contains information (called a *link*) that tells us where the second node is; the second node has a link to tell us where the third node is, and soforth. Clearly the list can be any length.

3.1 The Nodes are Structs

Here's a simple way to implement a linked list: it keeps track of as many numbers as we like:

```
struct listnode {
  int data;
  struct listnode *next;
} ;
```

This `struct` has two members: `data`, which actually holds a number, and `next`, which points to the next node in the list.

3.2 Adding a New Node to a List

Suppose the address of the first node in the list is stored in the variable `firstnode`, whose type is `struct listnode *`. Suppose also we discover that we need to remember one more number, `newnumber`. How can we add that number onto the list of numbers in a list of these nodes?

```
struct listnode *temp;

temp = malloc(sizeof(struct listnode));

(*temp).next = firstnode;
(*temp).data = newnumber;
firstnode = temp;
```

First we create enough space for a new list node, with `malloc`; then we link the node to the rest of the list by setting its `next` pointer to point to the old first node. Then we store the number we want to remember into the data member of the new node. Then we remember that the new node is now first, with `firstnode = temp`. We can do this as many times as we want, until the memory runs out.

3.3 Getting the Data Back Out of the List

Now suppose we want to average the numbers stored in each node. It's easy:

```
struct listnode *current;
int average;
long int sum = 0L;
int nodecount = 0;

for (current = firstnode;
     current != NULL;
     current = (*current).next) {
  sum += (*current).data;
  nodecount += 1;
}

average = sum / nodecount; /* round down */
```

Here we've assumed that the `next` pointer of the last node in the list is `NULL`. This makes perfect sense, and we can be sure that no other node but the last has `NULL` for its `next` pointer, because all the other `next` pointers point to other nodes, whose addresses are not `NULL`.

I hope it's obvious how this is all relevant to the problem of managing an arbitrarily large stack for your calculator.

3.4 The -> Operator

The operation of accessing the member of a structure via a pointer to the structure is common. We've used the construction `(*foo).bar` several times already. C lets you use a short cut.

In all circumstances, the expression `foo->bar` is identical with `(*foo).bar`. `foo` must be a pointer to a structure, and `bar` must be the name of one of the members of that structure.

3.5 Other Operations on Structures

The things you can do with a structure are limited. You can find its address with `&`, and you can access its members with `.` .

Since the ANSI standard, some things which were Too Much Work are no longer considered to be too Much Work:

You can pass a structure as an argument to a function (in which case the function gets a copy of the structure, the same way it does when you pass an

<int>) and you can return a structure value as a return value from a function.

You can compare two structures of the same type for equality or inequality with `==` and `!=`; two structures are 'equal' if each pair of corresponding members are equal.

You can assign a structure value to a variable of the right structure type with `=`; the members are copied one at a time.

You can't do anything else with a structure.

4 Global Variables and Type Declarations

Normally, you declare a variable inside a function. Then the variable is created when the function is called, is destroyed when the function returns, and the name of the variable is known only within the function.

This holds for structure type definitions also; if you define a structure type inside a function, the type and its name are only known within that function. That's not too useful; we would like the structure type definition to be visible everywhere.

If you write a declaration or type definition outside of all the functions, it is called a *global* variable or definition. It becomes visible to every function that follows the declaration or definition in the entire file. We have been doing this with function prototypes all along; in fact you can put a function prototype inside another function, and the information that the prototype communicates is only known locally, and now outside the function in which the prototype appears. Doing this is rarely useful.

If you write `int foo;` at the top of your program, outside all the functions, then every function in your entire file will have access to the variable `foo`. If one function changes the value of `foo`, the others can see the change; the name `foo` refers to the same variable, no matter which function is using it. Furthermore, unlike a local variable, which is created when the function that owns it is called and destroyed again when the function returns, a global variable is created when your program is run and is not destroyed until the program completes.

A global variable is an abnormal communication mechanism; functions can use it to communicate values back and forth even though it may not be obvious that they're doing so; for this reason, you should avoid using global variables in your program. There are occasions when it is appropriate: when some large piece of information is really global, and most of the functions make frequent

references to it.[3]

Function prototypes, structure type definitions, manifest constants, and other information that doesn't change, on the other hand, are appropriate things to make global.

5 A Program to Count Words

At the end of these notes you'll find the complete code for the example we chipped away at in class: It reads 'words' from the standard input, counts the number of occurrences of each word, and prints a report when it's done. It uses a linked list to keep track of what words it seen and how many times it's seen each word. There is no arbitrary upper bound on the number of different words it can handle.

What follows here are notes on the code; the numbers in the margins are source code line numbers. I hope you will read the code carefully and try to understand what each part is doing. The notes only exlpain the fine points; most of the details of how the program works are not explained in the notes.

16 `<ctype.h>` is here to provide the definition of `isspace`, which we use later, in function `getword`.

18 `<malloc.h>` declares `malloc`, `free`, and other related functions.

27 This is the defintion of the ystructure type we'll use for a node in our linked list. Each node has three parts: A pointer to the word it represent, a count of the number of occurrences of that word seen so far, and a pointer to the next node.

41 We could describe a list of n nodes this way: It has a head node, which has some data associated with it, and which has a pointer to a list of $n-1$ elements. So a list of 2 nodes has a head node, which has some data and which has a pointer to a list of 1 node. A list of 1 node has a head node, which has some data and which has a pointer to a list of 0 nodes. But this latter pointer is actually `NULL`, because the head node is the last node in the list, so we've just suggested that it might be wise to consider the `NULL` pointer as a 'pointer to a list with 0 nodes'.

[3] For example, a program might read a *configuration file* that specifies certain details about how the program should work; the information in the configuration file is stored in a large structure. Nearly every function has to consult one or another member of this structure, so it might be appropriate to make the structure variable global.

When we initialize `list`, we initialize it with a 'pointer to the empty list', `NULL`, and it turns out that the function `count`, which manipulates lists, does the right thing if we pass it this pointer.

45 `count` might append a new node onto the head of the list, and we need a way to apprise `main` of that fact, so that it can remember where its list starts. We do this by returning a pointer to the new head node from `count` after we've attached it, and main just stores this pointer in `list`, the variable it was using to hold a pointer to the first node in the list.

We wouldn't have to bother with this if `count` just attached the new node to the tail of the list instead of to the head, but that's uaully harder to do, because the head of a list is usually easier to find than the tail.

46 `getword` allocated space for `newword`; we passed `newword` to `count`, which in turn made a copy of it to store in the list, and so the copy we got back from `getword` can be freed now.

Why have `count` copy `newword` at all? If you're writing a function like `count` and you need to save some data that was passed in, it's a good idea to make a copy because you never know when your calling function might decide to destroy the original.

51 Note that this code works even if the input contained no words at all: In that case, the body of the `while` loop on lines 44–49 was never executed at all, and `list` is still `NULL`; we exit the program without printing anything.

66 `count` accepts two arguments: a word and a pointer to the first node in a list. It searches the list for a node whose `word` member is `word`; if it finds it, it performs its primary function and increments the count in that node. Otherwise, it manufactures, initializes, and links in a new node. In either case it returns a pointer to the new head node of the list (which might be the same as the old head node) to tell its caller whether or not the list has gotten longer.

71 Note that this code also works if the calling function passes in `NULL` for `list`: the condition on the `for` loop fails immediately and we proceed to like 77, where we begin the process of adding a new node to the list.

81 We used the `exit` function here. `exit` completely terminates the program when it is called; control returns to the operating system. Accordingly, `exit` does not have a return value. Contrast `exit`, which terminates 'normally', with `abort`, which terminates 'abnormally'. `exit` accepts one argument, which is treated the same way a return value from `main` is; in fact, the effect of `exit(n)` is identical to that of a `return n;` from `main`.[4]

[4]It might be more accurate to say that a `return` from main is identical to a call to `exit`, because compilers frequently compile it as one.

Here we `exit` with a return status of `1`, signalling that something went wrong.

90 We copy `word` here; see the note for line 46.

92 If we're initializing a new node, we initialize its count to 1, because we've seen exactly one occurrence of the word that the node represents.

100 `create_new_node` is a separate function for the usual reasons:

1. It's functionally separate from the other functions.
2. It might one day be called from more than one place, since it performs a generally useful function.
3. It might one day change and include more complicated node-building apparatus such as initializations.
4. If we ever again need a function that allocates and returns a list node, we might be able to come steal this one.

103 My compiler complains about this line, because the return value from `malloc` is a <pointer to `char`>, which is implicitly converted to a <pointer to `struct wordcount_node`> when we return it from the function. Usually, if you're converting pointers in this way, it means you made a mistake. On some machines, certain pointer conversions will fail altogether.[5] However, the value returned from `malloc` is an exception: The memory it points to is guaranteed to be suitable for storing any variable whatever, and the pointer is guaranteed to be freely convertible to any other pointer type. `malloc` must take special pains to ensure that this is the case. The compiler complains only because it doesn't know enough about `malloc`.

111 I wrote this function two years ago, and I've been re-using it ever since.

113 We allocate a buffer of `MAXWORDLEN` characters to hold the input word, but this function never checks to make sure it isn't writing past the end of the buffer. If the input contains an extremely long word, `getword` will happily write past the end of `buf`. This is a serious error.

117 `isspace` is a function whose argument is a character; it returns **true** if the argument is a white space character and **false** otherwise. White space characters include the blank, tab, and newline. `isspace` was declared in `<ctype.h>`. (Line 16.)

[5] For example, consider a computer with a main processor and an auxiliary processor which is very good a floating-point arithmetic and which has its own auxiliary memory, optimized for storing floating-point numbers. The compiler might very well decide to store <`int`>s into the main memory and <`float`>s into the auxilliary memory; in that case you wouldn't be able to interconvert <pointer to `int`> and <pointer to `float`>. This is a somewhat contrived example.

120 We haven't seen `ungetc` yet; it's like the opposite of `getc`: it 'unreads' a character. After you `ungetc` a character onto a stream, the next attempt to read the stream will proceed as if you had never read that character; the first character returned by the reading function will be the character you 'unread'. `ungetc` is limited: You can't un-get more than one character at a time, and you can't un-get a second character until you've read the first one back. `ungetc` is declared in `<stdio.h>`.

128 the loop on lines 123–124 reads characters from the standard input into the buffer `buf`[6], and stops when it reads a space or hits `EOF`. Now suppose the last word in the file was not followed by a space; say the last word is `visible`, and then after the `e`, nothing. `getword` has read `visible` into the buffer, and has just hit `EOF`. What do we want it to do?

We do not want `getword` to return `NULL` the instant it sees `EOF`, because if it did our caller would never find out about `visible`. So instead, we return `NULL` only if there is nothing in the buffer; that is, when `i == 0`. In the case of `visible`, `i` is 7, so we duplicate `visible` and return the copy; then the *next* time `getwords` is called, it hits `EOF` right away, without reading any characters into the buffer `buf`, and so `i` is 0 and it returns `NULL`.

[6] *buffer* is a generic word that describes a part of memory that input is being read into.