Lecture notes: Week 5

This week was a bit of a grab bag. We covered

- File APIs
- Allocating arrays inside a function
- Character arrays as strings
  - argc and argv
- Struct layout in memory

File API

We walked through the various aspects of the File API in class. In particular, we discussed

```
FILE * fopen(char * filename, char * mode)
```

This function opens a file in a mode. If it successfully opens the file, it returns a *pointer to a* `FILE` *struct* (otherwise it returns `NULL`). This struct holds various information about the file (e.g., where in the file the “read head” is), but you don’t get to know the details. Instead, the only way to manipulate the `FILE` struct is by passing it to various functions in the file API.

The modes of the file are:
* “`r`”: read mode. If the file doesn’t exist, fopen fails
* “`r+`”: read/write mode. If the file doesn’t exist, fopen fails
* “`w`”: write mode. If the file doesn’t exist, fopen will create a new file, unless the directory is read protected. If the file already exists, the existing data in the file is removed.
* “`w+`”: read/write mode. Unlike “`r+`” mode, the file opening rules work like “`w`”: the file is created if it doesn’t exist, and emptied if it does.
* “`a`”: append mode. Open a file for appending. Unlike “`w`” mode, the file is not erased. Instead, the “head” is at the end of the file, and future writes will add to the end of the file.
* “`a+`”: open for reading and appending. Writing to the file writes to the end. Reads from the file could read from the beginning of the file or the end.

```
int fscanf(FILE * fp, char * format, …)
```

Read data from a file pointed to by `fp`. The format string tells `fscanf` what kind of data to read (`%d` for integers, `%f` for floats, etc.). You then need to pass the addresses of locations to write the results to. For example, to read two integers from a file, you write:

```
fscanf(fp, “%d %d”, &x, &y);
```

`fscanf` will skip past white space to read in integers. But if it encounters the end of the file or a non-whitespace character that it cannot turn into an `int` (or whatever else it is trying to read in), it will stop. `fscanf` returns the number of elements that it successfully read. So, for example, if it only reads in one integer before running into a non-integer character or the end of the file, `fscanf` will return 1.

```
int fprintf(FILE * fp, char * format, …)
```

fprintf works just like printf, except that it writes to a file instead of to the console. It returns the number of characters (bytes) printed to the file.

Note that you can use fscanf to read from the console by passing it stdin, and fprintf to write to the console by passing it stdout (standard output) or stderr (error reporting output).

What's the difference between stdout and stderr? They are different output streams that your shell treats differently. When you run a program with an output redirect, like ./hw1 6 3 > output, the > tells the shell to redirect stdout to the file. But anything printed to stderr will still print to the screen.

Allocating arrays inside a function

If you want to allocate an array inside a function, one way to do it is to return the address of the allocated array:

```
int * allocArray(int numEls) {
   return (int *) malloc(numEls * sizeof(int));
}
```

Which will return the address of the box malloc creates. But what if you want to allocate two arrays inside the function? You can't really do it by passing the two arrays to the function:

```
void allocateTwoArrays(int numEls, int * arr1, int * arr2) {
   arr1 = (int *) malloc(numEls * sizeof(int));
   arr2 = (int *) malloc(numEls * sizeof(int));
}
```

Note that here we're passing the arrays to the function by passing the address of the beginning of the arrays to the function: a pointer to the beginning of the array. This would let us change the *contents* of the arrays (by writing, for example, arr1[2] = 9), but if we want to create a *new* array, we need to change the address the pointers hold. But inside the function, we're operating on *copies* of those pointers, not the pointers themselves. Updating arr1 and arr2 inside the function doesn't change what arr1 and arr2 point to *outside* the function.

Instead, we will use the same trick as we did when we wrote swap. Any time we want to change a variable that exists *outside* a function from *inside* a function, we pass the *address* of the variable to the function. So we need to pass the *address of the variable that points to the array* to the function. Since the variable that points to the array is an int *, the address of that variable is an int * *!

```
void allocateTwoArrays(int numEls, int * * arr1, int * * arr2) {
   * arr1 = (int *) malloc(numEls * sizeof(int));
   * arr2 = (int *) malloc(numEls * sizeof(int));
}
```

By dereferencing `arr1` and `arr2`, we are changing the original array pointers! We can call it like this:

```
// the arrays we want to allocate
int * outsideArr1;
int * outsideArr2;

//note that we’re passing their addresses to the function:
allocateTwoArrays(10, &outsideArr1, &outsideArr2);
```

This is an important general principle: if you have a variable `x` and you want to change its value inside a function, you need to pass the *address* of `x` to the function. If the type of `x` is `T`, the type of its *address* is `T *`. Inside the function, you will be working with a variable `T * v`. To change `x`, you just have to follow the pointer: every time you talk about `* v`, you’re talking about `x`. So updating `* v` *inside* the function updates `x` *outside* the function.

Character arrays as strings

C does not have a separate string data type. Instead, it uses *character arrays* to represent strings. A character array is an array of characters with one character per character in the string, and a special character, `‘\0’`, that marks the end of the string.

So to create the string “Hello” we would write:

```
char s[] = “Hello”;
```

And this creates a character array with *6* characters. This is equivalent to writing:

```
char s[] = {‘H’, ‘e’, ‘l’, ‘l’, ‘o’, ‘\0’};
```

We could have also written:

```
char * s = “Hello”;
```

Which creates an array with the characters `{‘H’, ‘e’, ‘l’, ‘l’, ‘o’, ‘\0’}` and a pointer `s` that points to that array.

The `‘\0’` is called the “null terminating character,” and it is critical in the way that C handles strings. Note that this is literally the value `0` — it’s *not* the same thing as the *character* `‘0’` (that has the value `48`).

String API

C also provides some special functions to help you manipulate strings, contained in `<string.h>`:

```
int strlen(char * s);
```

Returns how long the string is, *not counting* the null terminator (the length of “Hello” is 5, even though there are six elements in the array).

```
int strcmp(char * s1, char * s2);
```

Compares the strings `s1` and `s2` alphabetically (well, sort of: it compares them by character value, and all capital letters are “smaller” than all lowercase letters, so `“aardvark”` is smaller than `“zebra”`, but `“Zebra”` is smaller than `“aardvark”`)

You can also copy one string to another:

```
char * strcpy(char * src, char * dest)
```

Note that dest needs to have enough space to hold `strlen(src) + 1` characters (the + 1 is needed to account for the `‘\0’`) If it doesn’t have enough space, your program could segfault or worse. (This is a common cause of security bugs)

Note that all of these functions can break in bad ways if the character arrays aren’t null terminated. They all look for the ‘\0’ as their cue to stop, and if they don’t find it, your program could crash. The string API in C is pretty fragile.

argc and argv

Now that we know how character strings works, we can understand what the arguments to `main` are:

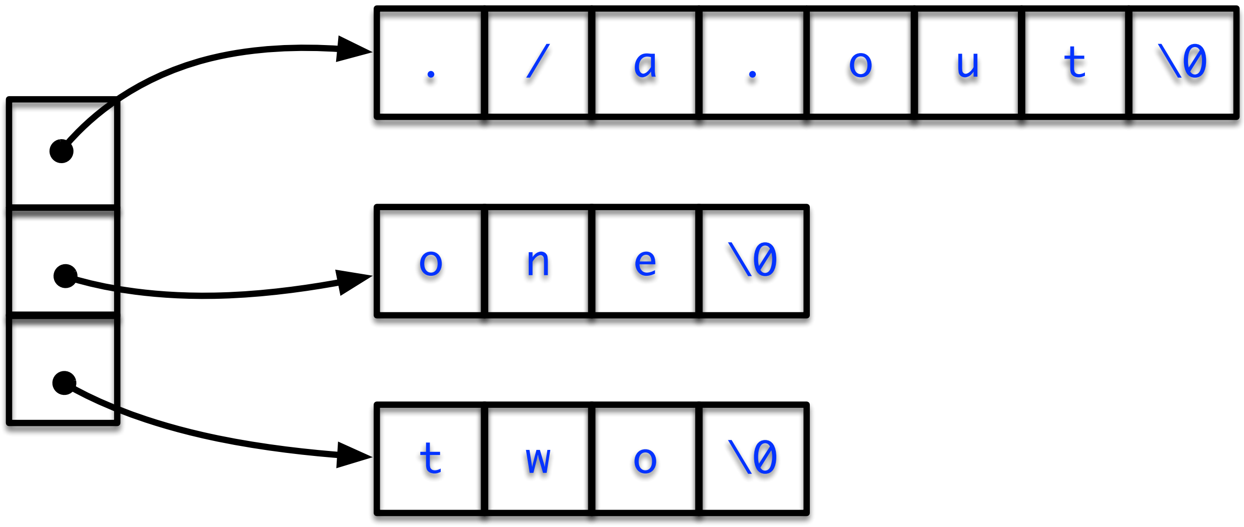
```
int main (int argc, char ** argv)
```

Here, `argv` is an *array of strings*. It’s an array of `char *` pointers, where each pointer in the array points to a string. `argc` tells you how big that array is. Note that the first string in the array is *always* the name of the program itself.

If we run a program like:

```
./a.out one two
```

Then `argc` is 3, and `argv` looks like this:



Struct layout in memory

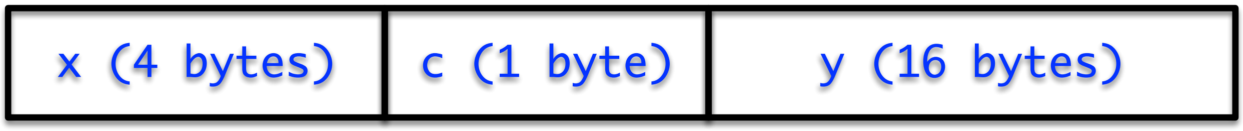
The way that fields in a struct are represented in memory is generally up to the compiler. However, you can tell the compiler to pack the fields together and place them in order:

```
#pragma packed(1)
```

If you do this, then a compiler will lay out the elements of a struct consecutively in memory:

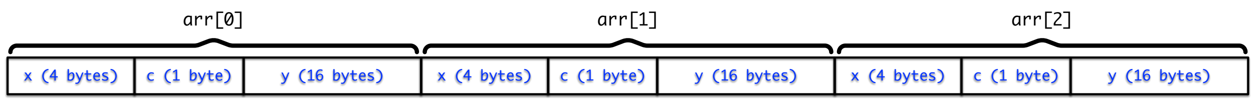
```
typedef struct {
   int x;
   char c;
   int y[4];
} myStruct

myStruct s;
```

`s` will look like this in memory:



And if we create an *array* of `myStruct`s, they are all laid out next to each other:

```
myStruct arr[3];
```



arr[0]
arr[1]
arr[2]
x (4 bytes)
c (1 byte)
y (16 bytes)
x (4 bytes)
c (1 byte)
y (16 bytes)
x (4 bytes)
c (1 byte)
y (16 bytes)