For-Loops and Arrays of Strings

You can make an array of various types with the idea that a string and an array of bytes are the same thing. The next step is to do an array that has strings in it. We'll also introduce your first looping construct, the `for-loop`, to help print out this new data structure.

The fun part of this is that there's been an array of strings hiding in your programs for a while now: the `char *argv[]` in the `main` function arguments. Here's code that will print out any command line arguments you pass it:

ex13.c

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i = 0;

    // go through each string in argv
    // why am I skipping argv[0]?
    for (i = 1; i < argc; i++) {
        printf("arg %d: %s\n", i, argv[i]);
    }

    // let's make our own array of strings
    char *states[] = {
        "California", "Oregon",
        "Washington", "Texas"
    };

    int num_states = 4;

    for (i = 0; i < num_states; i++) {
        printf("state %d: %s\n", i, states[i]);
    }

    return 0;
}
```

The format of a `for-loop` is this:

```
for(INITIALIZER; TEST; INCREMENTER) {
    CODE;
}
```

Here's how the `for-loop` works:

- The `INITIALIZER` is code that's run to set up the loop, which in this case is `i = 0`.
- Next, the TEST Boolean expression is checked. If it's false (0), then `CODE` is skipped, doing nothing.
- The `CODE` runs and does whatever it does.
- After the `CODE` runs, the `INCREMENTER` part is run, usually incrementing something, such as in `i++`.
- And it continues again with step 2 until the TEST is false (0).

This `for-loop` is going through the command line arguments using `argc` and `argv` like this:

- The OS passes each command line argument as a string in the `argv` array. The program's name (./ex10) is at 0, with the rest coming after it.
- The OS also sets `argc` to the number of arguments in the `argv` array, so you can process them without going past the end. Remember that if you give one argument, the program's name is the first, so `argc` is 2.
- The `for-loop` sets up with `i = 1` in the initializer.
- It then tests that `i` is less than `argc` with the test `i < argc`. Since $1 < 2$, it'll pass.
- It then runs the code that just prints out the `i` and uses `i` to index into `argv`.
- The incrementer is then run using the `i++` syntax, which is a handy way of writing `i = i + 1`.
- This then repeats until `i < argc` is finally false (0), the loop exits, and the program continues on.

What You Should See

To play with this program, then, you have to run it two ways. The first way is to pass in some command line arguments so that `argc` and `argv` get set. The second is to run it with no arguments so you can see that the first `for-loop` doesn't run if `i < argc` is false.

Exercise 13 Session

```
$ make ex13
cc -Wall -g    ex13.c   -o ex13
$ ./ex13 i am a bunch of arguments
arg 1: i
arg 2: am
arg 3: a
arg 4: bunch
arg 5: of
arg 6: arguments
state 0: California
state 1: Oregon
state 2: Washington
state 3: Texas
$
$ ./ex13
state 0: California
state 1: Oregon
state 2: Washington
state 3: Texas
$
```

Understanding Arrays of Strings

In C you make an *array of strings* by combining the `char *str = "blah"` syntax with the `char str[] = {'b','l','a','h'}` syntax to construct a two-dimensional array. The syntax `char *states[] = {...}` on line 14 is this two-dimensional combination, each string being one element, and each character in the string being another.

Confusing? The concept of multiple dimensions is something most people never think about, so what you should do is build this array of strings on paper:

- Make a grid with the index of each *string* on the left.
- Then put the index of each *character* on the top.
- Then fill in the squares in the middle with what single character goes in each square.
- Once you have the grid, trace through the code using this grid of paper.

Another way to figure this is out is to build the same structure in a programming language you are more familiar with, like Python or Ruby.

How to Break It

- Take your favorite other language and use it to run this program, but include as many command line arguments as possible. See if you can bust it by giving it way too many arguments.
- Initialize `i` to 0 and see what that does. Do you have to adjust `argc` as well, or does it just work? Why does 0-based indexing work here?
- Set `num_states` wrong so that it's a higher value and see what it does.

Extra Credit

- Figure out what kind of code you can put into the parts of a `for-loop`.
- Look up how to use the comma character (,) to separate multiple statements in the parts of the `for-loop`, but between the semicolon characters (;).
- Read about what a `NULL` is and try to use it in one of the elements from the `states` array to see what it'll print.
- See if you can assign an element from the `states` array to the `argv` array before printing both. Try the inverse.