C track: assignment 3

Goals

In this assignment you will learn about C arrays, C strings and how to make your program interact with the unix (*i.e.* Linux) command line. You will then write a simple sorting program. Finally, you will learn some useful strategies for testing code.

Language concepts covered this week

- arrays
- strings
- command-line argument processing

Other concepts covered this week

- Makefiles again.
- Using **assert** for debugging.
- Test scripts again.

Reading

Please read this page on command-line argument processing.

Arrays in C

Arrays in C are declared as follows:

```
int foo[10];  /* Declares an array called 'foo' with space for ten ints. */
```

and accessed as follows:

```
int i;

...

i = foo[4];  /* This gets the element from 'foo' at index 4. */
```

Note that arrays in C start at element 0, *not* element 1. Therefore, in this case, the last element in the array would be `foo[9]`. Note also that array elements are not initialized to be anything,

so they should be assumed to hold garbage until you assign a value to them. Also, you need to realize that if you try to access an element "off the end" of the array (*e.g.* the 100th element of the ten-element array `foo` in the code above), you will get no compiler warnings, but you'll probably get a core dump when the program runs. This is part of C's no-error-checking I-assume-you-know-what-you-want philosophy of programming.

Arrays can also be two-dimensional, three-dimensional, etc. The syntax is analogous:

```
int foo[10][5];
int i;

...

i = foo[4][2];
```

In addition, arrays can be *initialized* when they are declared:

```
int foo[5] = { 1, 2, 3, 4, 5 };

int bar[2][3] =
{
  { 1, 2, 3 },
  { 4, 5, 6 }
};
```

You will need to use array initialization in the next lab. Finally, you can pass arrays to a function like you would pass a normal variable:

```
void munge(int array[])
{
   /* code that uses the values in array[] */
}

/* more code... */

int main(void)
{
    int stuff[10];

    /* more code... */

    /* Pass the 'stuff' array to the function 'munge'. */
    munge(stuff);

    /* etc. */
}
```

However, when you do this, you should be aware that you are *not* passing a copy of the array to the function but the array itself. That means that if you modify the array in the function it will remain modified when you return from the function. This will be useful in the assignment below. The reason for this behavior is due to the fact that C arrays are actually represented as pointers; we'll cover this in later lectures and assignments.

Program to write

Write a program called `sorter` which behaves as follows:

1. If there are no command-line arguments at all when the program is run, the program should print out instructions on its use (a "usage message"; see this page). There should only be one usage message, and it must follow the standard conventions (see the previous link). Note that the optional command-line arguments (see below) must be included as part of the usage message.

2. The program will be able to accept up to 32 numbers (integers) on the command line.

3. If there are more than 32 numbers on the command line, or no numbers at all, the program should print out the usage message and exit.

4. If the optional command-line arguments "`-b`" or "`-q`" are found **anywhere** in the command line, change the behavior of the program as described below.

5. If any of the command-line arguments to the program are not integers or one of the two optional command-line arguments, your program's response is undefined -- it can do anything. (*i.e.* you shouldn't worry about having to handle anything but integer arguments or the two command-line options). The way to deal with this is as follows: for each argument, first check to see if it's one of the command-line options. If so, proceed accordingly. If not, assume it represents an integer and convert it using the `atoi()` function (see below).

6. Sort the numbers using either the minimum element sort or the bubble sort algorithm (see below). **Do not use a global array to hold the integers**; use a locally-defined array in `main` and pass the array to the sorting function. Define separate functions for both sorting algorithms. Use `assert` (see below) to check your sorting function for correctness.

7. Print out the numbers from smallest to largest, one per line.

The Sample Output

If the above doesn't make sense, this is what your program should look like (bold is what you would type, and % is the Unix prompt):

```
% sorter 5 9 -2 150 -95 23 2 5 80
-95
-2
2
5
5
9
23
80
150
% sorter
usage: sorter [-b] [-q] number1 [number2 ... ] (maximum 32 numbers)
%
```

Command-line arguments

Your program begins in the `main` function, which up until now has looked like this:

```
int main(void)
{
   /* your code here */
   return 0;
}
```

but will now look like this:

```
int main(int argc, char *argv[])
{
   /* your code here */
   return 0;
}
```

So let's take the first line apart. There are a few parts to this:

`int`

Declares that this function returns an integer.

`main`

Declares this function's name, `main`. Recall that C programs always begin executing in the `main` function.

`int argc`

`argc` is equal to the number of elements of `argv`.

`char *argv[]`

Okay, this one's a bit trickier. The second argument, `argv`, is an array of "`char *`"s. "`char *`" is C's way of handling character strings, which are represented as arrays of characters where the last character is ASCII character 0 (often written as `'\0'`). This will make more sense when we have discussed pointers in a couple of weeks. `argv` contains one string for each of the command line arguments that your program is run with. More on this below.

To give an example of this, let's take the command line from the example above:

```
% sorter 5 9 -2 150 -95 23 2 5 80
```

This would produce the following values in `argc` and `argv`:

```
argc        10
argv[0]    "sorter"
argv[1]    "5"
argv[2]    "9"
argv[3]    "-2"
argv[4]    "150"
argv[5]    "-95"
argv[6]    "23"
argv[7]    "2"
argv[8]    "5"
```

```
argv[9]    "80"
```

Note that `argv[0]` holds the name of your program, and that the first user-supplied argument is in `argv[1]`. *Remember this!* (One thing this implies is that `argc` is 1 greater than the number of user-supplied arguments.) Notice also that a command-line argument of "5" is **not** an integer; it's a string that can be converted into an integer. Which leads us to...

Converting strings to integers

Okay, so how do we turn these argv strings into integer values? Well, there's this handy function called `atoi` ("**a**scii **to** **i**nteger"). For example, `atoi("5")` equals 5. In order to use `atoi`, you need to put the following line at the top of your program:

```
#include <stdlib.h>
```

`atoi` is a pretty dumb function; if you pass it a bogus value it'll just return 0 instead of signalling any kind of error. That means that you need to check for the optional command-line arguments "`-b`" and "`-q`" before trying to convert a command-line argument to an `int`.

The minimum element sort algorithm

Alright, so what is this "minimum element sort" algorithm? The basic idea is that the smallest element in the array will be the zeroth element in the sorted array, the second-smallest will be the first element, etc. Here's how it works:

Let `array` be the array of integers, and `num_elements` be the total number of elements in `array`.

1. Start with `start = 0` (for the index of the zeroth element).
2. Set `smallest = start` (smallest stores the index of the smallest element encountered so far).
3. Run through a loop with the variable `index` going from `start` to `num_elements`
    - If `array[index] < array[smallest]`, set `smallest = index`.
4. Once the loop ends, swap `array[start]` and `array[smallest]` (moving the smallest element found to the beginning of the array you searched).
5. Increment `start`, and if `start < num_elements`, go back to step 2. **Do not** use a `goto` statement, however; use another loop.
6. If `start >= num_elements` then you're done and the array is sorted.

Adding command-line options

You will add the ability to process two optional command-line arguments (usually called "command-line options") to your program. These options will be `-b` and `-q`. To test for these, you'll need to be able to compare strings. The function `strcmp(str1, str2)` returns `0` if `str1` and `str2` are the *same* and nonzero otherwise. So to check if the first argument is `-b`, you

would have to do something like:

```
if (strcmp(argv[1],"-b") == 0)
{
     /* put stuff here */
}
```

To use the `strcmp` function, you need to put the following line with the other `#includes` at the top of your file:

```
#include <string.h>
```

WARNING! Do **not** do this:

```
if (argv[1] == "-b")  /* WRONG! */
{
     /* put stuff here */
}
```

This doesn't work; you can't compare strings using the == operator. The reason for this will be made clear when we talk about pointers.

You should add information on any command-line options you allow to the usage information that is printed out when there are no arguments. **NOTE**: As mentioned above, now would be a really good time to read this page on how to handle command-line arguments and optional arguments. You are required to abide by these conventions for this and all future assignments; if you don't, we will make you redo the assignment.

If the user supplies one of the command-line options, you must *not* treat it as part of the list of integers. Also, **you must *not* assume that the command-line options are going to be entered before the numbers**; it should be possible to enter them anywhere after the program name (and the test script will check for this). Finally, **it's legal to enter the command-line options more than once**, although this won't make any difference after the first time. **You don't need complicated code to achieve this!** You can process all of the command-line arguments in a single pass through the `argv` array. Many students write absurdly complicated code to handle the command-line arguments, and it's just a waste of effort. In other words, it's much easier than you probably think it is.

One thing you **do** have to make sure of is that you aren't adding values into the array of numbers at indices that are too large. This array can only be 32 elements long at most, so if you try to assign to indices 32 or greater, it's an error even if it seems to work properly. **NOTE**: It should be clear why indices greater than 32 are off-limits, but why do you think it's illegal to assign to index 32 exactly?

To keep things simple, you are only required to handle integers or the two specific command-line options `-b` and `-q`, and any command-line argument (not counting the program name) that isn't `-b` or `-q` can be assumed to be an integer.

Here are the command-line options and what they mean:

- The "`-b`" option means that you should sort using a Bubble Sort instead of a Minimal Element Sort algorithm.

Bubble Sort, like Minimal Element Sort, is not a very efficient algorithm. Much more efficient algorithms (such as Quicksort) exist, but they are best left until after you've had some experience with recursion, which is coming up next lesson :-) Also, the C standard library has a `qsort` (Quicksort) library function, but to use it you need to understand function pointers, which we won't cover in this track (although it's not that hard).

- The "`-q`" option suppresses the output (*i.e.* nothing gets printed). Why would we want to do this? See the next section.

Information on different sorting algorithms, including bubble sort, can be found all over the web, in algorithms textbooks, written on bathroom walls, scrawled on fortune cookies, etc. etc.

More on command-line options

Here are some guidelines (which we expect you to follow) for implementing the command-line option processing in this program. They will make it much easier to get a working program that passes the test script.

- First off, do not change the elements in the `argv` array at all! There is no need to. Some people try to shuffle values in the `argv` array, as if you need to pass `argv` to the sorting functions. You don't. What you need to do is to create a brand new array that can hold all the numbers in the `argv` array. Altering the `argv` array is legal, but it's poor programming style.

- Second, this new array has to be large enough to hold all the numbers on the command line, and no larger. Given what you all know now, this means that it should be 32 elements long. You may not use all of them (that's fine), but they will be there in case someone enters a command line with 32 numbers. Later, I'll show you how to create arrays that have exactly the right number of elements, based on the input. Also, you aren't allowed to do this:

  ```
  int n = 10;
  int array[n];  /* Invalid */
  ```

  This isn't legal C, so if you do this, you'll get a compiler warning, and my policy is to not allow programs that generate compiler warnings. In addition, you can't just use a humongous array, like this:

  ```
  int array[1000];
  ```

  This kind of thing is just programming to make the test script work, usually for the wrong reason. So 32 is as big an array as I'll allow.

- Third, you have to check if there are **more** than 32 numbers in the command line, which means 32 values that are neither the program name (`argv[0]`) nor `-b` nor `-q`. If there are more than 32 numbers, the program should exit with a usage message and return a non-zero value from `main()`. The test script will check for this. In addition, make sure that your program doesn't at any point add an element to the numbers array at an invalid index *i.e.* an index larger than 31. Even if the program passes the test script, this is a serious error, because you're writing into memory that isn't part of the array.

- Fourth, the `-q` or `-b` arguments can occur _anywhere_ on the command line except in

`argv[0]` (you should know why they can't occur in `argv[0]`), and there can even be more than one `-b` or `-q` arguments in the command line. Multiple `-b` or `-q` arguments don't do anything beyond what single ones do. The test script checks for this too.

- Finally, despite all these guidelines, this is actually a pretty easy problem; you can do everything in a single `for` loop. If your solution is very complicated, you are almost certainly doing something wrong.

Avoiding magic numbers

A "magic number" is a number that is just plopped into a program with no context, and especially a number like this that is repeated several times in a program. In this lab, the number "32" hard-coded into the program would be a magic number. Magic numbers are almost always bad, for two reasons:

1. They tell the reader nothing about why this number is this value.
2. If you decide to change it to a different value, you have to change it in multiple places, and it's easy to forget one, leading to hard-to-find bugs.

It's extremely easy to avoid using magic numbers; just use the `#define` preprocessor instruction to define a meaningful symbolic name for the magic number, and **only** use the symbolic name in the program. That way, if you want to change the number's value, you only have to do it in one place (by changing the `#define` statement).

We will take marks off for using magic numbers in this lab and in all subsequent labs, so make sure you don't use them. **Specifically, the number "32" should only occur <u>once</u> in your entire program, inside a `#define` statement!**

Commenting

Don't forget to write good comments! From now on, we'll expect your commenting to be top-notch, unless (as in some later labs) we supply you with a code template that already has the comments (in which case we'll expect that you've read and understood the comments, and that your code is consistent with them).

Specifically, we want to see:

- A comment at the top of the source code file explaining what the program does.

- A comment before each function (except for `main`) which explains what the function does, what its arguments mean, and what its return value represents (if it has a return value).

- Comments inside functions explaining how the code works, unless it's totally obvious. For this lab, for instance, you'll want to explain how your sorting functions work. It's better to put this inside the function than before the function, because knowledge of how it works isn't usually important for using the function (*i.e.* it's not part of the function's interface, just its implementation).

You may think this is too much work; it isn't. The amount of time it takes you to write good comments is usually only a fraction of the time it takes you to write good code. In fact, many

programmers advocate writing comments **before** writing code; the comments then serve as a kind of "scaffolding" around which you build your program. Writing the comments first forces you to think clearly about what you want to achieve, instead of just randomly writing code and stopping when it seems to work.

Testing your program

Many programmers can write working code, but few programmers test their code systematically to see if it actually does work. It is important to learn different strategies for doing this, because it can not only improve the quality of your code, but also the quality of your life. If you don't believe me, that's because you haven't spent enough four-hour+ sessions trying to track down a single bug caused by a typo somewhere in your program. Don't laugh; it happens all the time, and it happens more with C than with any other computer language (except maybe C++). What we want to do here is see how we can make it happen less often.

The joy of `assert`

Using `assert` is probably the single easiest way to improve the quality of your code, and one of the least used. The idea behind `assert` is to make your program *self-checking*. Let's say you've just coded up an incredibly hairy algorithm which will balance the budget, cure cancer, send mankind to the stars and do other wonderful things. How do you know that your code doesn't have a bug? And if it does have a bug, how do you find it?

Well, there are probably points in your program where you expect certain things to be true. For instance, in the sorting program described above, after you've completed the sort, you expect that the array of integers is, in fact, sorted. If you could check this right after you did the sort, then if your sorting algorithm ever fails on any input, you will know about it right away. `assert` makes this easy. Using `assert` looks like this:

```
int i;

/* your algorithm goes here... */

assert(i == 10);  /* assert that the integer variable i is equal to 10. */
```

In the above case, if the program reaches the `assert` statement and `i` is equal to 10, the program continues on executing; the `assert` statement has no effect. However, if `i` is not equal to 10, the program aborts (exits), telling you the exact line in the file where the error occurred. "Whoa!", you're probably saying, "that's a bit extreme, isn't it?" Well, the point is to use `assert` for situations where you *know* something has to be true assuming that your algorithm is correct. If it isn't true, your program has failed, and all bets are off. (In actual fact, it's fairly easy to write custom versions of `assert` that don't abort but just print a nasty message and continue.) The `assert` statement is referred to as an *assertion*, logically enough.

You might also be asking: why don't I just write this:

```
int i;

/* your algorithm goes here... */
```

```
if (i != 10)
{
    abort();  /* or exit(), or some similar function */
}
```

instead of using that `assert` function? Well, aside from the fact that `assert` is more concise, and that `assert` prints out line numbers where the assertion failed, the big advantage of assertions is that they can be *switched off*. Even though assertions don't usually slow your code down appreciably, there is some cost, which goes up the more assertions you use. Let's say you've used assertions to debug your code, and you reach a point where you are highly confident that your algorithm is correct. At this point you may want your algorithm to run as fast as possible, and not waste time checking assertions that you are sure will always be true. In other words, you want to leave the assertions out in the compiled code, but you don't want to remove the lines with the `assert` statements in them in case you modify the code later and need to debug it again.

It turns out that by adding the magic word **`-DNDEBUG`** as an argument to `gcc` when you compile the C code, the assertions will be removed from the code before compilation, so your code will run at maximum speed.

Using `assert` in your program

To use `assert`, make sure you add this to the top of your file:

```
#include <assert.h>
```

In your sort function(s), write this right at the end of the function:

```
/* Check that the array is sorted correctly. */

for (i = 1; i < num_elements; i++)
{
    assert(array[i] >= array[i-1]);
}
```

where `num_elements` is the number of elements in the array and `array` is the name of the array. Notice that here we're putting the `assert` statement in a loop. This bit of code is technically called a *postcondition*; a postcondition is what is required to be true after a function has completed. Some computer languages, such as Eiffel, have very sophisticated systems for checking postconditions (as well as *preconditions* and various kinds of *invariants*); these languages make it easier to write correct code than C does. Here, the postcondition will cause an assertion failure if the array isn't sorted at the end of the sorting function, which is what we want.

Notice how the postcondition is just three lines of trivial code, as compared to the sort routines themselves, which will probably be considerably longer. This illustrates the general principle that it's much easier to check if something is right than to make it right in the first place. Assertions are easy to write; use them!

NOTE: It may seem wasteful to write the same assertion code at the end of both sorting functions, but that's where I want you to write it, and not *e.g.* in the `main()` function. That's because you will eventually be writing larger programs which span many files, and functions you write will be used outside of the file they are written in. It's important for functions to be

self-contained as much as possible, and if the assertion checking is built in to a sorting function, then anyone using that function will automatically get the benefit of the assertion checking.

Running the test script

Now you get to know why you added the `-q` option to your program. The reason for the `-q` option is to make the program only print output if the assertion fails. We can use this to write a test script to check the program.

We are supplying you with a Makefile and a test script for your program. The test script is written in a computer language called python. Python is a scripting language which is excellent for many tasks, one of them being writing test scripts. Python is available on most Linux systems, including on all of the CS cluster machines. What you should do is to download the test script into the same directory that your lab 3 C code is located in. Use the "save page as" feature of your browser; don't try to cut and paste the code into a text editor, because the odds are 100-to-1 that it won't work properly. When the file is in the correct location, make it executable by typing:

```
% chmod +x run_test
```

where `%` is the terminal prompt.

What the test script does is to generate a series of random inputs to the program and run the program with the `-q` option. The test script is invoked by typing

```
% make test
```

at the terminal prompt. The test script will tell you what it's doing as it proceeds. If your sort routines ever fail, you will see the output of `assert`, telling you what line the failure occurred at. The test script will run the program using the Minimal Insertion Sort as well as the Bubble Sort. If something else goes wrong (*e.g.* your command-line processing is faulty, or a core dump occurs), the test script will report an error, along with the program invocation that caused it. It will not tell you exactly what error occurred, but by re-running the erroneous invocation (by cutting and pasting into your terminal window) you should be able to figure it out yourself. If all goes well, the test script will report success.

Note that if there is a bug in your assertion code itself, all bets are off. However, as mentioned above, writing correct assertions is pretty trivial (especially since we've done it for you :-)).

Supporting files

- The Makefile.

  Once again, be sure that all the command lines in the Makefile start with tabs, or they will not work.

- The test script.

  Once again, in order to get this to work, you have to do

```
% chmod +x run_test
```

after downloading this file, in order to make it executable. From now on, we'll assume you'll remember this (and the Makefile tab rule above) for future assignments.

- The style checker. To invoke the style checker you can just type

  ```
  % make check
  ```

  at the prompt (assuming that the Makefile is in the same directory).

Again, remember to read this page for more information on how to handle command-line arguments effectively.

To hand in

The `sorter.c` file. We will check to see that it compiles, passes the style checker and passes the test script.

References

- Darnell and Margolis, chapter 7.
- K&R, chapters 1 and 5.
- Any algorithms textbook *e.g.* Sedgewick, Algorithms in C.
