Worksheet 4: Arrays and Strings

Updated: 20th August, 2019

The objective of this practical is to understand the use of arrays, strings and command-line parameters.

Pre-lab Exercises

1. Array Declarations

Explain the meaning of each of the following:

(a)
```c
int a[14];
```

(b)
```c
double b[] = {3.3, 0.0, 1.1, 2.2};
```

(c)
```c
float c[5][10];
```

(d)
```c
int *d[10];
```

(e)
```c
void **e[3][4];
```

(f)
```c
char s[] = "Hello";
```

(g)
```c
char *s = "Hello";
```

2. Arrays and Pointers

Consider the following code:

```c
int a[] = {10, 15, 20, 25};
int b[] = {50, 60, 70, 80, 90};

int* x[] = {a, b};
int* y[] = {a + 2, b + 3};

int* p;
int* q;
int** r;

p = a;
q = y[1];
r = &q;

*p = &p[3] - y[0];
r[0][1] = **r - y[0][1];
```

(a) Draw a diagram showing the pointer relationships.

(b) Show the contents of `a` and `b` at the end.

3. Array Expressions

Consider the following 2D fixed array:

```
int c[5][4] = {{ 1,  2,  3,  4},
               { 5,  6,  7,  8},
               { 9, 10, 11, 12},
               {13, 14, 15, 16},
               {17, 18, 19, 20}};
```

Given this code, explain the meaning of each of the following expressions:

(a) `c[2][1]`

(b) `c`

(c) `c[2]`

(d) `c + 2`

(e) `*(c + 2)`

(f) `c[2] + 1`

(g) `*(c[2] + 1)`

(h) `*(*(c + 1) + 2)`

(i) `c[0][6]`

(j) `*((c + 2) + 1)`

4. Malloc'd Arrays

Write suitable `malloc()` statements to dynamically allocate the following:

(a) An array of 25 `ints`.

(b) An array of 25 pointers to `floats`.

(c) An array of 25 pointers, each pointing to an array of 15 `chars`.
(Hint: use a `for` loop.)

5. Miscellaneous Questions

(a) What is the difference between a pointer to an array of `int`s ("`x`") and a pointer to the first element of the same array ("`y`")?

(b) What is the only character that cannot appear in a string, and why?

Practical Exercises

Note: The following practical exercises relate to a *single C program*. As before, you should organise your code into different .c and .h files as needed, and create a suitable makefile.

1. Pointers to Functions

Note: This question related to last weeks prac. You just just modify that code directly. The following questions is a whole new question

Back inside `order.c`, define another function called `order()`. This function should take a single `char` as a parameter, and return a function pointer. When passed "A", return a pointer to `ascending3()`. For "D", return a pointer to `descending3()` instead. If the parameter is neither "A" nor "D", return `NULL`.

To help simplify your code, write a `typedef` declaration for the function pointer type, placing it in the appropriate header file.

Modify your `main()` function again to make use of `order()`. Rather than calling `ascending3()` directly, you should use the function pointer returned by `order()`.

2. Arrays and Functions

Write the following functions. Each takes an array of `int`s and the array length. The return types differ.

You should also write a temporary `main()` function for testing purposes. Use the array initialisation notation to create test arrays. (Later we'll replace this with another `main()` that takes user input via command-line parameters.)

(a) `sum()` — adds up all the array elements and returns the sum, an `int`. (For an array containing {5, 10, 15}, `sum()` should return 30.)

(b) `max()` — returns the index of the largest element in the array. (For an array containing {10, 5, 1}, `max()` should return 0. For the array {3, 15, 6, 500, 9} it should return 3).

(c) `reverse()` — reverses the order of the array elements. It doesn't return anything. (For an array containing {1, 2, 3, 4}, `reverse()` should change it to {4, 3, 2, 1}.)

3. String Conversion

Write a function to convert an array of string-formatted integers into an array of `int`s. For instance, given the array {`"7"`, `"1"`, `"14"`, `"-5"`}, the function should produce the array {7, 1, 14, -5}.

The function should take three parameters — an array of `char*` (i.e. an array of strings, not just a single string/`char*`), an `int` array and a length.

Note: There are a few standard C functions used to convert a *single* number from a string to an `int`. Consult the lecture notes, or other resources.

4. Array Output

Write a function to output an array of `int`s on a single line. Your function should take an array and an array length, and return `void`. The output generated should resemble the following:

```
{4, 14, 5, 8, 2}
```

Consider how to get the right number of commas.

5. Command-Line Arguments

Write a program that accepts command-line arguments and does the following:

1. The program should report an error if there are less than two arguments (keeping in mind that `argv[0]` is not an argument but the name of the program.)
2. Otherwise, the program should use the conversion function from Question 2 to convert argument 2 and onwards from strings to `int`s, placing them in an `int` array.
3. Pick a maximum array size (declaring it with `#define`) and declare the array to be this size. Note that this is just the *maximum* array capacity, while the functions from Question 1 require the *actual* number of elements.

   We'll do it this way before trying malloc'd arrays later.
4. If the first argument (`argv[1]`) is the name of one of the functions from Question 1 (`"sum"`, `"max"` or `"reverse"`), your program should call that function and output the result.

For `sum()` and `max()`, this is a single value. For `reverse()`, use the array output function from Question 3.

5. If the first argument is anything else, the program should report an error.

Note: If your `main()` function is getting long, break it up into multiple functions. You should practice using multiple functions spread across multiple source files (with a makefile).

6. Malloc'd Arrays

Convert your solution to the previous question to use dynamic `malloc`'d arrays, instead of fixed-size arrays. Remember to `free` all `malloc`'d memory when you no longer need it.

Note: With `malloc`, the array capacity can be exactly equal to the number of elements. You won't need `#define` to define a maximum.

7. Strings and Characters

Write a function to convert a string to upper case (without using toupper). Your function should take a `char*` as a parameter and return nothing (modifying the existing string rather than returning a new one).

Note: A single `char` is actually an 8-bit integer, where each letter, digit and punctuation symbol is represented by a fixed integer "ASCII" value. Uppercase and lowercase letters have sequential values; e.g. `'B' == 'A' + 1` and `'b' == 'a' + 1`. Use this information to work out how to convert a single character to uppercase.

You can find the values for all characters by looking up an ASCII table. *However*, you don't actually need to know them. You can simply use `'A'`, `'B'`, etc.

Try implementing this function in several different ways:

(a) Using the `strlen()` function and retrieving each character in turn with the array indexing notation (`array[i]`).

(b) Without using `strlen()`. (Hint: how do you know where a string ends?)

(c) Without using the array indexing notation. (Hint: modify the string pointer itself.)

Use this function in your program so that the first command-line parameter is case-insensitive (i.e. so that the user can enter "sum", "SUM", "Sum", "suM", etc.).

End of Worksheet