CP Lab 6: Arrays and some strings

Instructions

The purpose of this Lab is to help you get more programming experience with C. We will do some extra work with *arrays* and *iteration* (or *loops*) in this lab. We will also introduce you to programming with *strings*.

This lab sheet is a bit shorter than the previous two, to give those of you who still need it a bit more time. If you're caught up, then go back to the optional exercises from the catch-up week.

All Labs for CP will take place on the DICE machines in AT5.05 (Thursday and Tuesday) or AT6.06 (Monday). For working alone, you may use any of the DICE labs in AT, as long as they are not booked out by a class. You have 24-hour access to AT levels 3,4,5: if your card/pin does not allow you in, use the ITO contact form on our course webpage to contact the ITO.

Aims

- Get you extra experience with the DICE system, and its use for C programming.
- Get some extra practice with arrays and iteration.
- Write a program to calculate the first 100 prime numbers and store them in increasing order in an array.
- Write a 'string manipulation' program making use of the basic concepts of assignment to variables, input, output and the `if`-statement, switch-case and some predefined *string functions*.
- *if you have extra time:* Implement and test the new version of `whatday.c` which we demonstrated in Lecture 9.

Getting started

- Log into one of the DICE machines, bring up a Terminal window, move to your home directory, and create a new subdirectory called `lab6`. You may want to copy the `template.c` file from your `lab2` subdirectory into the `lab6` directory.
- If you have not completed Stage A of Lab sheet 5 yet, you may devote some of today's lab session to that. (Leave the rest of Lab 5 until you're fully caught up – Labs 6 and 7 are more important.)
- If you do not finish Lab sheet 6 today, please spend time over the week to catch up.

Stage A Reversing an array

In Lecture 9, and in lab sheet 5, we considered the task of defining a function called `Rotate` which accepted as input an integer `array` and another parameter carrying the length of that array, and performed a *circular rotation* of the array by one-position right.

In this first stage of the lab, we will consider the task of designing an function to *reverse* an array. The function you will design should have the following function prototype:

```
void Reverse (int a[], int n)
```

Your function should accept a pointer (to the starting address of an integer array) as input, as well as an integer representing the number of cells of that array, and reverse the entire array in memory.

The best starting point for this task is the earlier program `rotate.c` (containing the function `Rotate`) which we saw in lecture 9, and tested in last week's lab.

The only difference with the `Reverse` function is that for `Reverse`, the desired order we want is different. If we consider the defined array of `rotate.c`:

```
int primes[10] = 2, 3, 5, 7, 11, 13, 17, 19, 23, 29;
```

then the order we would expect after calling `Reverse(primes, 10)` *would be*

`29, 23, 19, 17, 13, 11, 7, 5, 3, 2.`

You should start your program either by copying over the usual `template.c`, or even `rotate.c` from last week's lab.

Your result for `reverse.c`

- ☐ Program should be *correct*. To test it, you can use the segment of code you developed for testing `Rotate` in last week's lab. You can then determine whether you get correct outputs.
- ☐ Make sure you rename `a.out` into a file `reverse` (or similar) before proceeding with the next stage of this lab.
- ☐ Call the lecturer or demonstrator over if you need help.
- ☐ Submit your program with
  `submit cp lab6 reverse.c`

Stage B String Manipulation Program

Now we are going to write a string manipulation program. Your program should be named *stringMan.c*. You will use the `if`-statement and/or `switch-case` construct (you have yet to meet `switch-case` in lectures, but you're welcome to look it up and use it if you wish). This program will have the following capabilities:

(i) It will implement a function called `function1` which reverses the given string. This function should match the function prototype:

```
void function1(char* str)
```

(ii) It will also implement a function called `function2` which takes as input parameters a string `str` and two characters `drop` and `sub`, and replaces occurrences of the `drop` character in `str` with `sub`. Function prototype should be:

```
void function2(char* str, char drop, char sub)
```

(iii) Your program will also implement a function `function3` which returns 1 if the given string is a palindrome, and 0 otherwise.

A string is said to be a *palindrome* if it is a word (or phrase, or number) that reads the same when reversed.

The function should match the prototype

```
int function3(char* str)
```

For each of the three functions, you will be passing it a string from `main`, and this string will already have its own storage (*as an array of* `char`) in `main`. For working with these strings ...

- You should assume some fixed maximum length for strings and always declare this amount of space (an array with this many `char` entries) for any string you work with in `main`.

  The usual way to do this (in traditional C) is to do

  ```
  #define MAXLEN 100
  ```

  and then define all your character arrays like

  ```
  /* the +1 leaves space for the terminating null character */
  char mystring[MAXLEN+1];
  ```

- Then when you are passed a string as input to a function ...

  You can assume that that pointer has the fixed number of `char` for you to work with.

  Also, because we pass with pointers, all changes you make to the passed string are being made directly to the string that `main` gave you.

- The final thing to remember is that all strings should be terminated with `'\0'` character, and that's how we recognise the end of string.

- For each of the functions, you will need to have a `for`-loop (or maybe a `while`-loop) to iterate through the string (which is after all, an array of `char`).

Apart from writing these three functions, you will also need to write a `main` function to scan in strings from the user (using `scanf`) and ask the user which function they want to test this time.

For your `main`, you will need to take care of certain things:

- I expect you may want to use the `strlen` function in at least some of your functions. Also for `function3` you might want to use `strcmp`.

  If you are going to make use of either of these functions, make sure to add the following `#include` at the top of your program.

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

- You will need to declare an array of `char` of some appropriate length to store strings entered by the user, as described above (and in lecture 14).
- You will need to use `printf` to ask the user their choice of the functions (1,2 or 3) to use this time, and then use `scanf` to read in this value.
- Regardless of whether the user chose 1, 2, or 3, we will always need to read in an initial string. Use `scanf` with the `%s` option to read in the user's string into the pre-declared array of `char` in `main`.
- What happens next depends on whether the user asked for option 1, 2 or 3.

  Important: If you are in the Tuesday lab you will have already met the `switch-case` statement in this morning's lecture. In that case, use a `switch-case` statement to carry out the appropriate option depending on what option the user chose. For the people in the Thursday/Monday lab, it's up to you – use `if`, or if you want to check out `switch` yourself, go ahead.

  – For option 1, you need a call to `function1`, passing the string as the parameter to `function1`.

    After the function terminates, use a `printf` with `%s` to display the reversed string.

  – For option 2, we will use `function2`. This means we will need to take in some extra input from the user.

    * Use a `printf` to ask for the character which will be *dropped*, and then a `scanf` with `%c` to read in this `char`.

      Important: We can use `scanf("%c",...)` to read a single character from the user. *However*, the first character we read will not be the one we're looking for, but will be the 'newline' character left over from when the user hit ↵ after typing their input string.

      So we need to use the pattern `" %c"` (with a 'space" in front of the `%c`) to 'skip over' the carriage-return which is still in the input buffer. (A space in a `scanf` format string tells it to skip over any number of whitespace characters: spaces, newlines, tabs.)

    * Use a `printf` to ask for the character which will be *substituted in*, and then a `scanf` with `%c` to read in this `char`.

      Important: Same issue with `scanf` as just above.

    * After we have all our inputs, we make a call to `function2`, passing the entered string, followed by the two char values.

    * After the call terminates, we will use a `printf` with `%s` to print out the new value of the string.

  – For option 3, we will use `function3`

    * We call `function3`, passing the string as the parameter, and assigning the result of the call into an integer variable.

    * Depending on whether the value returned by `function3` is `1` or `0`, use `printf` to either print `palindrome` or `not a palindrome`.

After you have written the program, please compile it using

```
gcc -Wall stringMan.c ↲
```

Your result for `stringMan.c`

- ☐ Program should be *correct*. To test:
  - For the choice 1 and the string `university` the output should be `ytisrevinu`.
  - For the choice 2, the string `london`, and the characters `o` (to replace) and `a` (to substitute), the output should be `landan`.
  - For the choice 3, if the input is `madam` the result should be `palindrome`.
  - For the choice 3, if the input is `history` the result should be `not a palindrome`.
- ☐ Submit your program with
  `submit cp lab6 stringMan.c`

Stage C First 100 primes – a 'challenge' problem

Finally we are going to write a simple program to calculate the first 100 primes, and store them in increasing order in an array. No functions for this question, we just do everything in `main`.

Remember the definition of a prime number:

A prime number is any integer strictly greater than 1, which can only be divided by itself and 1.

So the first prime is 2, the next 3, then next 5, and so on.

You are going to write your program in a files called `primes.c`. Apart from the basic structure of `template.c`, the main thing we need from the outset, is to define an array of length 100 for the primes inside the `main` function:

```
...
int main(void) {
  int primes[100];

...
}
```

Your program should use a *main loop* or *outside loop* in which one iteration adds a new prime (into `primes[i]`, say) before updating `i`:

- A `for`-loop may be suitable for this outside loop;
- However, in finding the 'next prime' we will:
  - Possibly will need to consider more than one possibility for the 'next prime' (since some test numbers `n` will turn out to not be primes at all).
  - Definitely, in testing for primality, will need to test many potential divisors (remember we can test whether k divides n by seeing whether the expression `n%k` has value 0) of `n`

  - The two observations above mean that we will need an internal loop as well as the outside one; the internal loop does the checking to find the 'next prime'
  - Depending on how you write your code, you might want to have a third loop 'inside' the internal one.
- Think about the different variables you will need to solve this problem.

Apart from the 'problem solving' code you should also include a section at the end to ask the user for an integer at most 100, and return that prime.

- A nice thing to do during the debugging stage is just to have one `printf` to output the first five cell values of the `primes` array. This will allow you to see mistakes in the early stages of testing.

For this task, 'program planning' before coding, will be helpful. There is quite a bit of problem solving involved.

Your result for `primes.c`

☐ Program should be *correct*. To test it, you will need a segment of code at the end where you ask the user for a integer k, and tell them what the kth prime is (if `k<=100`).

The k-th prime will be in `primes[k-1]` (if you have carried out the task properly).

The answers you should get are:

  - For k=2, should return 3;
  - For k=4, should return 7;
  - For k=10, should return 29;
  - For k=100, should return 541.

☐ Make sure you rename `a.out` into a file `primes` (or similar) before proceeding with the second stage.

☐ Submit your program with
`submit cp lab6 primes.c`

Julian Bradfield after *Mary Cryan and Çiğdem Beyan, updated Nov 2016*