

**Carleton University**  
**Department of Systems and Computer Engineering**  
**SYSC 2006 - Foundations of Imperative Programming - Winter 2017**

**Lab 4 - Memory Diagrams and More Practice with Arrays**

**Attendance/Demo**

To receive credit for this lab, you must demonstrate your solutions to the exercises. When you have finished all the exercises, call a TA, who will review your code, ask you to run the test harness provided on cuLearn, and assign a grade. For those who don't finish early, a TA will ask you to demonstrate whatever code you've completed, starting about 30 minutes before the end of the lab period. **Any unfinished exercises should be treated as "homework"; complete these on your own time, before your next lab.**

**Part 1 - Tracing Code/Memory Diagrams**

**Warm-up:** During recent lectures, you were shown how the online C Tutor helps us visualize what happens, step-by-step, as the computer executes each line of code in a C program.

Links to the two examples that were presented in class are on cuLearn. Review the operation of C Tutor by clicking on the links and tracing the execution of these programs.

**Exercise:** Fibonacci numbers are defined by the following formulas:

$$F_1 = 1$$

$$F_2 = 1$$

$$F_n = F_{n-1} + F_{n-2}, n > 2$$

Also, it is conventional to define  $F_0$  as 0. The Fibonacci sequence for  $n = 0, 1, 2, 3, 4, 5, 6, 7, 8 \dots$  is therefore 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

Here is a definition of a C function that is passed  $n$  and returns  $F_n$ , for  $n \geq 0$ .

```
int fibonacci(int n)
{
    if (n == 0)    // fib(0)
        return 0;
    if (n == 1)    // fib(1)
        return 1;

    int temp1 = 0;
    int temp2 = 1;

    int nextfib;
    for (n = n - 2; n >= 0; n = n - 1) {
        nextfib = temp1 + temp2;
        temp1 = temp2;          /* Point A. */
        temp2 = nextfib;
    }
    return nextfib;            /* Point B. */
}
```

Here is the definition of a `main` function that calls `fibonacci`:

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

int main(void)
{
    int result;
    result = fibonacci(5);
    printf("fib(%d) = %d\n", 5, result);    /* Point C. */
    return 0;
}
```

**Without using C Tutor**, draw three separate memory diagrams, one each for parts (a), (b) and (c). *Do not combine your solutions into a single diagram.* Use the same notation as the C Tutor.

- (a) Draw a memory diagram that depicts the program's activation frame(s) immediately after the statement at Point A is executed for the first time; that is, immediately after

```
temp1 = temp2;
```

is executed during the first iteration of the `for` loop.

- (b) Draw a memory diagram that depicts the program's activation frame(s) after control has left the `for` loop, but immediately before the statement at Point B is executed; that is, just before the `return` statement is executed.
- (c) Draw a memory diagram that depicts the program's activation frame(s) immediately before the statement at Point C is executed; that is, just before the `printf` call is executed.

To double-check your work:

- Download `fibonacci.c` from cuLearn and open this file in Pelles C.
- The *Labs* section on cuLearn has a link, [Open C Tutor in a new window](#). Click on this link.
- Copy/paste the program from Pelles C into the C Tutor editor.
- Use C Tutor to trace the program one statement at a time, stopping when you reach Points A, B and C. Compare your diagrams to the visualization displayed by C Tutor.

## Part 2 - Arrays

### General Requirements

You have been provided with four files:

- `exercises.c` contains incomplete definitions of three functions you have to design and code. (There are also incomplete implementations of two extra-practice exercises.)
- `exercises.h` contains the declarations (function prototypes) for the functions you'll implement. **Do not modify `exercises.h`.**
- `main.c` and `sput.h` implement a *test harness* (functions that will test your code, and a `main` function that calls these test functions). **Do not modify `main` or any of the test functions.**

For those students who already know C or C++: when writing the code for the exercises, do not use pointers or `structs`. They aren't necessary for this lab.

Your functions should not be recursive. Repeated actions must be implemented using C's `while`, `for` or `do-while` loop structures.

None of the functions you write should perform console input; i.e., contain `scanf` statements. Unless otherwise specified, none of your functions should produce console output; i.e., contain `printf` statements.

You must format your C code so that it adheres to one of two commonly-used conventions for indenting blocks of code and placing braces (K&R style or BSD/Allman style). Pelles C makes it easy to do this - instructions were provided in the handouts for previous labs.

### Getting Started

#### Step 1

Launch Pelles C and create a new project named `more_array_exercises`.

- If you're using the 64-bit edition of Pelles C, the project type should be **Win 64 Console program (EXE)**. (Although the 64-bit edition of Pelles C can build 32-bit programs, you may run into difficulties if you attempt to use the debugger to debug 32-bit programs.)
- If you're using the 32-bit edition of Pelles C, the project type should be **Win32 Console program (EXE)**.

When you finish this step, Pelles C will create a folder named `more_array_exercises`.

#### Step 2

Download files `main.c`, `exercises.c`, `exercises.h` and `sput.h` from cuLearn. Move these files into your `more_array_exercises` folder.

#### Step 3

You must also add `main.c` and `exercises.c` to your project. To do this:

- Select **Project > Add files to project...** from the menu bar.

- In the dialogue box, select `main.c`, then click **Open**. An icon labelled `main.c` will appear in the Pelles C project window.
- Repeat this step for `exercises.c`.

You don't need to add `exercises.h` and `sput.h` to the project. Pelles C will do this after you've added `main.c`.

#### Step 4

Build the project. It should build without any compilation or linking errors.

#### Step 5

Execute the project. The test harness will report several errors as it runs, which is what we'd expect, because you haven't started working on the functions the harness tests.

#### Step 6

Open `exercises.c` in the editor. Design and code the functions described in Exercises 1 through 3. If you finish before the end of the lab period, attempt the extra-practice exercises (Exercises 4 and 5); otherwise, you should do these on your own time.

**Your functions must not create temporary arrays; in other words, they must not have declarations similar to:**

```
int temp[n];
```

**Instead, your functions should modify their array arguments, as required.**

**Use the indexing (`[]`) operator to access array elements. Do not use pointers and pointer arithmetic (which have not yet been covered in lectures). This means your functions should not contain statements of the form `*ptr = ...` or `*(ptr + i) = ...`, where `ptr` is a pointer to an element in an array.**

#### Exercise 1

Write a function that prints the first  $n$  integers in an array. An incomplete implementation of this function is provided in `exercises.c`. The function prototype is:

```
void print_array(int arr[], int n)
```

The output must be formatted this way:

```
{1, 2, 3, 4, 5, 6, 7}
```

There should be a `{` before the first integer, a comma and one space after each integer except the last one, and a `}` immediately after the last integer.

Your function should assume that  $n$  is positive; i.e., it should not check if  $n$  is greater than 0.

You can use this function to help you debug the functions you'll write for Exercises 2-5. Insert calls to `print_array` at appropriate places in your function; e.g., before the `for` or `while` loop that traverses the array; inside the loop body, after an array element is modified; and after the loop.

Test suite #1 calls `print_array`, but it cannot verify that the function is correct. Instead, you have to have to review the function's output on the console.

## Exercise 2

Write a function that "rotates" the  $n$  integers in an array one position to the left. For example, the function will change the array `{6, 2, 5, 3}` to `{2, 5, 3, 6}`.

An incomplete implementation of this function is provided in `exercises.c`. The function prototype is:

```
void rotate_left(int arr[], int n);
```

Your function should assume that  $n$  is positive; i.e., it should not check if  $n$  is greater than 0.

Build the project, correcting any compilation errors, then execute the project. The test harness will run. Look at the console output and verify that your function passes all the tests in test suite #2 before you start Exercise 3.

## Exercise 3

Write a function that reverses the values in an array containing  $n$  integers.

An incomplete implementation of this function is provided in `exercises.c`. The function prototype is:

```
void reverse(int arr[], int n);
```

Your function should assume that  $n$  is positive; i.e., it should not check if  $n$  is greater than 0.

As an example, suppose the function is called this way:

```
int numbers[] = {1, 2, 3, 4, 5, 6, 7};
reverse(numbers, sizeof(numbers) / sizeof(numbers[0]));
```

When the function returns, array `numbers` will be: `{7, 6, 5, 4, 3, 2, 1}`.

Hint: your function doesn't need to consider arrays with an even number of elements and arrays with an odd number of elements as separate, distinct cases. There's no need to code something like this:

```
if (the array has an even number of elements) {
    reverse the array
} else {
    // the array has an odd number of elements
    reverse the array
}
```

Build the project, correcting any compilation errors, then execute the project. The test harness will run. Look at the console output and verify that your function passes all the tests in test suite #3 before you start Exercise 4.

**Note: the extra-practice exercises are after the wrap-up instructions.**

## Wrap-up

1. Remember to have a TA review your solutions to the exercises, assign a grade (Satisfactory, Marginal or Unsatisfactory) and have you initial the demo/sign-out sheet.
2. Remember to back up your project folder before you leave the lab; for example, copy it to a flash drive and/or a cloud-based file storage service. All files you've created on the hard disk will be deleted when you log out.

## Extra-Practice Exercises

### Exercise 4

Write a function that is passed an array of  $n$  integers. For each multiple of 10 in the given array, change all the values following it to be that multiple of 10, until encountering another multiple of 10. For example, the function will change the array:

`{2, 10, 3, 4, 20, 5}`

to:

`{2, 10, 10, 10, 20, 20}`

(The 3 and 4 after the 10 are replaced by 10, and the 5 after the 20 is replaced by 20.)

An incomplete implementation of this function is provided in `exercises.c`. The function prototype is:

```
void ten_run(int arr[], int n);
```

Your function should assume that  $n$  is positive; i.e., it should not check if  $n$  is greater than 0.

Build the project, correcting any compilation errors, then execute the project. The test harness will run. Look at the console output and verify that your function passes all the tests in test suite #4.

### Exercise 5

This exercise is more challenging than Exercises 1-4. Using only the material presented in the course to date, a correct solution will typically require between 15 and 20 lines of code. (Lines containing only a `{` or `}` are counted as one line of code, but comments are not counted as lines of code.)

Write a function that removes all the 10's from an array of  $n$  integers. The remaining elements should be shifted left towards the start of the array as required, and the "empty" spaces at the end of the array should be set to 0. For example, the function will change the array:

`{1, 10, 10, 2, 10, 3}`

to:

`{1, 2, 3, 0, 0, 0}`

An incomplete implementation of this function is provided in `exercises.c`. The function prototype is:

```
void without_tens(int arr[], int n);
```

Your function should assume that *n* is positive; i.e., it should not check if *n* is greater than 0.

Hint: the "obvious" solution uses nested loops (the outer loop searches the array for the next 10 to be removed, and the inner loop shifts all subsequent elements to the left), so you might want to start by developing that algorithm. If you want a challenge, revise that solution so that it does not use nested loops. There is at least one efficient solution that requires only two traversals of the array (one loop followed by another loop.)

Build the project, correcting any compilation errors, then execute the project. The test harness will run. Look at the console output and verify that your function passes all the tests in test suite #5.

### **Acknowledgments**

Some of these exercises were adapted from Java programming problems developed by Nick Parlante.