Carleton University
Department of Systems and Computer Engineering
SYSC 2006 - Foundations of Imperative Programming - Winter 2014

Lab 2

Attendance/Demo

To receive credit for this lab, you must make a reasonable effort to complete the exercises and demonstrate the code you complete.

When you have finished all the exercises, call a TA, who will review the code you wrote. 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. Finish any exercises that you don't complete by the end of the lab on your own time. Also, you must submit your lab work to cuLearn by the end of the lab period. (Instructions are provided in the *Wrap Up* section at the end of this handout.)

Part 1 - The Pelles C Debugger

A unit test harness like the one you used last week helps you determine whether the functions/modules you're developing are correct, but they don't pinpoint the bugs that cause the code to fail the tests. Inserting output statements (e.g., `printf` statements) in your functions can help you track down bugs, but this approach can quickly become tedious.

In this part of the lab, you're going to learn how to use the debugging tool that is built into Pelles C.

Step 1

Create a new folder named Lab 2.

Step 2

Create a new project named buggy_power inside your Lab 2 folder. The project type must be Win32 Console program (EXE). After creating the project, you should have a folder named buggy_power inside your Lab 2 folder (check this).

Step 3

Download files buggy_power.c from cuLearn. Move this file into your buggy_power folder.

Step 4

You must also add buggy_power.c to your project. To do this, select Project > Add files to project... from the menu bar. In the dialogue box, select buggy_power.c, then click Open. An icon labelled buggy_power.c will appear in the Pelles C project window.

Step 5

Open buggy_power.c in an editor window. This file contains two functions: `main`, and a `power` function that is similar to the one presented in lectures, except that has a couple of bugs.

Step 6

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

Step 7

Execute the project. The `main` function calls `power` to calculate 2^5. Look at the program's output. Clearly, it's incorrect. We'll use the debugger to help us locate the problems.

Step 8

Before we can use the debugger, we need to configure the project so that debugging information is stored in the executable program when the project is built.

- From menu bar, select Project > Project options... A Project options dialog box appears. Click the Compiler tab.
- From the Debug information: drop-down menu, select Full.
- From the Optimizations: drop-down menu, select None.
- Click the Linker tab.
- From the Debug information: drop-down menu, select CodeView & COFF format.
- Click OK to close the dialogue box.
- Build the project.

Step 9

We'll now run the project under the debugger.

- From menu bar, select Project > Debug buggy_power.exe (or click the Go/Debug symbol on the toolbar). A new window, titled Debugger, will open. Also, several stacked debugging views will appear. Click the tab labelled Locals. As you execute the code using the debugger, the Locals view will show the current values of the active function's local variables and parameters.
- We're now going to execute the program, one statement at a time. From the menu bar, select Debug > Step into, or press the F11, or click the Step into symbol on the toolbar. The first executable statement in `main` will be executed. Keep doing this, until `main`'s local variables (`x` and `result`) appear in the Locals view.
- Continue to single-step until `power` is called. Notice that `main`'s local variables disappear from the Locals view. As long as statements in `power` are being executed, `power`'s parameters and local variables will be displayed in the Locals view.
- The `power` function is supposed to raise `base` to the power `n` by calculating:

  ```
  1 * base * base * base * ... * base
  ```

  A total of `n` multiplications should be performed, one multiplication each time the body of the `while` loop is executed. After each multiplication, the product is stored in variable `p`:

  ```
  p = p * base;
  ```

  Execute the loop, one statement at a time, and notice that 0 is stored in `p` every time the assignment statement is executed. Examining the function reveals that `p` is initialized to 0, not 1, before the loop is entered, so the product calculated by the loop (and returned `power`) is:

```
0 * base * base * base * ... * base
```

- From the main menu, select Debug > Stop debugging. The debugger window and views will close.

Step 10

- Edit `power` so that local variable `p` is initialized to 1 instead of 0.
- Rebuild project and execute it (don't run it under the debugger). Examine the program's output. The output has changed, but it's still incorrect. We need to continue debugging.

Step 11

- From menu bar, select Project > Debug buggy_power.exe (or click the Go/Debug symbol on the toolbar).
- Execute the program one step at a time until the first statement in `power` is executed. What value is stored in parameter `n`? To raise `base` to the power `n`, the loop body should be executed `n` times. Continue single-stepping through the code. Count the number of times that the loop body is executed. Is this the correct number of iterations? When the loop is exited, just before the `return` statement is executed, what value is stored in parameter `n`? What value is stored in local variable `p`?
- From the main menu, select Debug > Stop debugging.
- Edit `power`. Modify the loop condition (`n >= 0`) so that the loop body is executed the correct number of times. Rebuild the project and execute it. If the program's output is not correct, run the project under the debugger and continue debugging, until you fix the program.

Breakpoints (do this after you've finished the entire lab)

Executing code one statement at a time can become tedious. To speed the debugging process, Pelles C allows allows you to set *breakpoints* in your code. After you've set one or more breakpoints, you can use the Debug > Go command. Pelles C will execute your code, halting when it reaches a breakpoint. At that point, you can single-step, executing one statement at a time, or select Debug > Go again to execute your code until another breakpoint is reached.

Information about setting breakpoints can be found in the Pelles C help facility. From the main menu, select Help > Contents. Find the section titled POIDE reference, then scroll down to the section titled Debugger.

Try setting breakpoints at various statements in `power`, then use the debugger to execute from breakpoint to breakpoint.

Part 2 - Arrays and Functions

Objective

The objective of this part of the lab is to write some C functions that process arrays.

A Brief Review of C Arrays

The C variable declaration:

```
type name[capacity];
```

allocates an array with the the specified `name`. The array's `capacity` is an integer expression, and specifies the number of elements in the array. Each element in the array stores a value of the specified `type`.

For example,

```
int samples[10];
```

declares an array named `samples` that can store 10 integer values.

Each element in an array is accessed by specifying the array name and the element's position (index), which is given by an integer expression. For example, `samples[0]` is the first element in array `samples`, `samples[1]` is the second element, and `samples[9]` is the tenth element.

An array index does not have to be a literal integer; instead, we can use any expression that yields an integer. Often, the index is specified by a variable of type `int`. Here is a loop that initializes the 10 integer elements in array `samples`:

```
// initialize samples to {0, 2, 4, 6, ..., 18}
int samples[10];
for (int i = 0; i < 10; i += 1) {
    samples[i] = 2 * i;
}
```

Here is an equivalent Python loop that creates an empty list, then initializes it by appending the same ten integers:

```
# initialize samples to [0, 2, 4, 6, ..., 18]
samples = []
for i in range(10):
    samples.append(2 * i)
```

There's an alternate way of declaring a C array that allows us to specify the initial values of the array elements by providing an *initializer list* as part of the declaration. For example, this statement:

```
int samples[] = {0, 2, 4, 6, 8, 10, 12, 14, 16, 18};
```

does the same thing as the earlier code fragment containing a `for` loop. Notice that we didn't specify the array's capacity. The C compiler calculates the array's capacity, based on the number of values in the initializer list.

C arrays can be used as function arguments. Here's a function that returns the sum of the first *n* values in an array of integers:

```
int sum_array(int a[], int n)
{
    int sum = 0;
    for (int i = 0; i < n; i += 1) {
        sum = sum + a[i];
    }
    return sum;
}
```

Notice how parameter `a` is declared. The `[]` indicates that the parameter is an array; however, we do not specify the capacity of the array. This means that the function can work with any array, regardless of its capacity, as long as each element in the array is of type `int`. It is the programmer's responsibility to ensure that the first *n* elements of the array have been initialized.

To sum all 10 integers in array samples, we call the function this way:

```
int total;
total = sum_array(samples, 10);
```

Notice that the first argument is the name of the array, `samples`, and not `samples[]`.

Of course, we can call the same function to sum just the first five elements of the array; i.e., calculate `samples[0] + samples[1] + samples[2] + samples[3] + samples[4]`:

```
int partial_sum;
partial_sum = sum_arrray(samples, 5);
```

Functions can modify their array arguments. Here's a function that initializes the first *n* elements of an array to a specified integer value:

```
void initialize_array(int a[], int n, int initial)
{
    for (int i = 0; i < n; i += 1) {
        a[i] = initial;
    }
}
```

We can call this function to zero all 10 elements in `samples`:

```
initialize_array(samples, 10, 0);
```

Aside (primarily for students who took SYSC 1005): an array can be thought of as a primitive Python list, but there are some important differences:

- When we create a Python list, we don't specify its capacity. Python lists automatically grow (increase their capacity) as objects are appended or inserted in a list. In contrast, the capacity of a C array must be specified when it is declared. The array's capacity is fixed; there is no way to increase its capacity at run-time.

- We can determine the length of a Python list (that is, the number of objects stored in the list) by passing the list to Python's built-in `len` function. In contrast, C does not keep track of how many array elements have been initialized, and there is no function we can call to determine this. It is the programmer's responsibility to do this, usually by using an auxiliary variable.

- Python generates a run-time error if you specify an invalid list index, but C does not check for out-of-bounds array indices. For example, a C expression such as `samples[10]` will compile without error. At run-time, this expression accesses memory outside the array. Similarly, while `samples[-1]` is a perfectly valid Python expression, when used in a C program, this expression accesses memory outside the array.

- Python provides functions, methods and operators that perform several common operations on lists; for example, append an object to the end of a list, insert an item in a list, delete an item from a specified position in a list, remove a specified object from a list, determine if a specified object is in a list, find the largest and smallest objects in a list, etc. In contrast, the only array operation C provides is the `[]` operator to retrieve or set the value at a specified index.

General Requirements

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

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

You have been provided with file main.c. This file contains incomplete implementations of five functions you have to design and code. It also contains 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.**

Instructions

Step 1

Create a new project named arrays inside your Lab 2 folder. The project type must be Win32 Console program (EXE). **Do not create this project inside the buggy_power folder you created in Part 1.** After creating the project, you should have a folder named arrays inside your Lab 2 folder, in addition to the buggy_power folder (check this).

Step 2

Download files main.c and sput.h from cuLearn. Move these files into your arrays folder.

Step 3

You must also add main.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.

You don't need to add 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.

The console output will be similar to this:

```
== Entering suite #1, "Exercise 1: max()" ==

[1:1]  test_max:#1  "max({1.0, 2.0, 3.0, 4.0}) ==> 4.0"  FAIL
!     Type:      fail-unless
!     Condition: fabs(max(data1, 4) - 4.0) < 0.001
!     Line:      93
[1:2]  test_max:#2  "max({1.0, 2.0, 4.0, 3.0}) ==> 4.0"  FAIL
!     Type:      fail-unless
!     Condition: fabs(max(data2, 4) - 4.0) < 0.001
!     Line:      95
[1:3]  test_max:#3  "max({4.0, 3.0, 2.0, 1.0}) ==> 4.0"  FAIL
!     Type:      fail-unless
!     Condition: fabs(max(data3, 4) - 4.0) < 0.001
!     Line:      97
[1:4]  test_max:#4  "max({5.0}) ==> 5.0"  FAIL
!     Type:      fail-unless
!     Condition: fabs(max(data4, 1) - 5.0) < 0.001
!     Line:      99
[1:5]  test_max:#5  "max({2.0, 2.0}) ==> 2.0"  FAIL
!     Type:      fail-unless
!     Condition: fabs(max(data5, 2) - 2.0) < 0.001
!     Line:      101

--> 5 check(s), 0 ok, 5 failed (100.00%)

== Entering suite #2, "Exercise 2: min()" ==

...

==> 13 check(s) in 5 suite(s) finished after 0.00 second(s),
     0 succeeded, 13 failed (100.00%)

[FAILURE]
*** Process returned 1 ***
```

In Exercise 1, you'll complete the implementation of a function named max. The first test suite is named "Exercise 1: max()". This test suite has one *test function*, named test_max. This function calls max five times. Each time, the value returned by max is compared to the value we expect a correct implementation of the function to return.

For example, the first test performed by test_max checks if max correctly returns the largest value in the array of doubles `{1.0, 2.0, 3.0, 4.0}`:

```
[1:1]  test_max:#1  "max({1.0, 2.0, 3.0, 4.0}) ==> 4.0"  FAIL
!     Type:      fail-unless
!     Condition: fabs(max(data1, 4) - 4.0) < 0.001
```

The condition may appear a bit strange. Because of how real numbers are represented in a computer, we should never use the `==` operator to compare two real numbers for equality. Instead, two real numbers are considered to be equal if they differ from each other by a small amount. So, we subtract 4.0 (the expected result) from the value returned by `max`, and call `fabs` to obtain the absolute value of this difference. If this value is small (less than 0.001), we consider the value returned by `max` to be equal to 4.0.

Step 6

Open `main.c` in the editor. Design and code the functions described in Exercises 1 through 5.

Exercise 1

Write a function that returns the maximum value in an array of doubles containing *n* elements. The function prototype is:

```
double max(double x[], int n);
```

Note: your function should assume that `n` is positive; i.e., it should not check whether `n` is passed a positive or negative value. Your function **cannot** assume that all elements in the array will be greater than any particular value; in other words, it **cannot** assume that all elements will be, for example, greater than 0 or greater than -999.0.

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 of the tests in the first test suite before you start Exercise 2.

Exercise 2

Write a function that returns the minimum value in an array of doubles containing *n* elements. The function prototype is:

```
double min(double x[], int n);
```

Note: your function should assume that `n` is positive; i.e., it should not check whether `n` is passed a positive or negative value. Your function **cannot** assume that all elements in the array will be smaller than any particular value; in other words, it **cannot** assume that all elements will be, for example, less than 0 or less than 999.0.

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 of the tests in the test suite before you start Exercise 3.

Exercise 3

There are several different ways to *normalize* a list of data. One common technique scales the values so that the minimum value in the list becomes 0, the maximum value in the list becomes 1, and the other values are scaled in proportion. For example, consider the values in this unnormalized list:

[-2.0, -1.0, 2.0, 0.0]

The normalization technique described above changes the list to:

[0.0, 0.25, 1.0, 0.5]

The formula for calculating the normalized value of the k^{th} value in a list, x_k, is:

$$normalized\ value\ of\ x_k = (x_k - min_x)\ /\ (max_x - min_x)$$

where min_x and max_x represent the minimum and maximum values in the list, respectively. If you substitute min_x for x_k in this formula, the dividend becomes 0, so the normalized value of min_x is 0.0. If you substitute max_x for x_k in this formula, the dividend and divisor have the same value, so the normalized value of max_x is 1.0.

Write a function named `normalize` that is passed an array containing *n* real numbers. This function will normalize the array using the technique described above. Your function should assume that the array will contain at least two different numbers. Your function must call the `max` and `min` functions you wrote for Exercises 1 and 2.

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 of the tests in the test suite before you start Exercise 4.

Exercise 4

A sound (for example; a note played on a guitar or a spoken word) is recorded by using a microphone to convert the acoustical signal into an electrical signal. The electrical signal can be converted into a list of numbers that represent the amplitudes of *samples* of the electrical signal measured at equal time intervals. If we have *n* samples, we refer to the samples as $x_0, x_1, x_2, \ldots, x_{n-1}$.

The *average magnitude*, or average absolute value, of a signal is given by the formula:

$$\text{average magnitude} = (|x_0| + |x_1| + |x_2| + \ldots + |x_{n-1}|)\ /\ n \quad = \quad \sum |x_k|\ /\ n; \quad \mathrm{k} = 0, 1, 2, \ldots, n-1$$

Write a function that returns the average magnitude of the signal represented by an array of doubles containing *n* elements. The function prototype is:

```
double avg_magnitude(double x[], int n);
```

Your function should assume that `n` is positive; i.e., it should not check whether `n` is passed a positive or negative value.

C`s math library (`math.h`) contains a function that calculate the absolute values of real numbers. The function prototype is:

```
// Return the absolute value of x.
double fabs(double x);
```

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 of the tests in the test suite before you start Exercise 5.

Exercise 5

The *average power* of a signal is the average squared value, which is given by the formula:

$$\text{average power} = (x_0^2 + x_1^2 + x_2^2 + \ldots + x_{n-1}^2)\ /\ n \quad = \quad \sum x_k^2 / n; \quad \mathrm{k} = 0, 1, 2, \ldots, n-1$$

Write a function that returns the average power of the signal represented by an array of doubles containing *n* elements. The function prototype is:

```
double avg_power(double x[], int n);
```

Your function should assume that n is positive; i.e., it should not check whether n is passed a positive or negative value.

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 of the tests in the test suite.

Wrap-up

1. Remember to have a TA review and grade your solutions to the exercises before you leave the lab.

2. The next thing you'll do is package the project in a ZIP file (compressed folder). From the menu bar, select Project > ZIP Files... A Save As dialog box will appear. Click Save. Pelles C will create a compressed (zipped) folder named arrays.zip, which will contain copies of the the source code and several other files associated with the project. (The original files will not be removed). The compressed folder will be stored in your project folder (i.e., folder arrays).

3. Log in to cuLearn, click the Submit Lab 2 link and submit arrays.zip. After you click the Add submission button, drag the file to the File submissions box. After the icon for the file appears in the box, click the Save changes button. At this point, the submission status for your file is "Draft (not submitted)". You can resubmit the file by clicking the Edit my submission button. After you've finished uploading your file, remember to click the Submit assignment button. This will change the submission status to "Submitted for grading".