ENCM 339 Fall 2016 Lab 3 In-Lab Exercises

Steve Norman
Department of Electrical & Computer Engineering
University of Calgary

September 2016

Attention: This document describes only the in-lab exercises of Lab 3. Over the weekend of October 1–2, a complete set of instructions, including post-lab exercises, will be posted to D2L.

1 Administrative details

1.1 Due dates

In-lab exercises must be handed in on paper by the end of your scheduled lab period, in the hand-in box *for your lab section*. The hand-in boxes are on the second floor of the ICT building, in the hallway on the west side of the building.

When you hand in your in-lab exercises, make sure that "ENCM 339" and *your name and lab section* are written in a clear and easy-to-spot way on the front page—it is a waste of time for both you and your TAs to deal with an assignment that doesn't have a name on it. Also make sure that your pages are *stapled together securely*—pages held together with paperclips or folds of paper tend to fall apart.

Post-lab exercises must be submitted electronically using the D2L Dropbox feature. All of your work should be in a *single PDF file* that is easy to read for your TA to read and mark. See the D2L module "How to make a post-lab PDF file for ENCM 339 in Fall 2016" for more information. Due dates and times for post-lab exercises are as follows:

Lab Section	Due Before
B01	Tue Oct 11, 9:00am
B02	Tue Oct 11, 1:30pm
B03	Thu Oct 13, 9:00am
B04	Thu Oct 13, 1:30pm

For both in-lab and post-lab exercises: 20% marks will be deducted from assignments handed in late, but no later than 24 hours after a due date. That means if your mark is X out of Y, it will be recorded as (0.8 times X) out of Y. There will be no credit for assignments turned in later than 24 hours after due dates; they will be returned unmarked.

1.2 Marking Scheme

Exercise	Type	Marks
A	in-lab	4 marks
B	in-lab	8 marks
C	unmarked	n/a

Marks for post-lab exercises have not yet determined.

Figure 1: Sketch of a program with a function to find the average element value of an array of `int`s. The function can be given only one argument, and the function is supposed to work correctly for whatever number of elements the array has. This problem is **impossible** to solve in C. (It's easy to solve, though, if a second argument is allowed.)

```
#include <stdio.h>

double average(int arr[ ]);

int main(void)
{
  int test1[ ] = { 10, 20, 20 };
  int test2[ ] = { 20, 21, 22, 24 };
  double avg1, avg2;
  avg1 = average(test1);
  avg2 = average(test2);
  printf("The averages are %f and %f.\n", avg1, avg2);
  return 0;
}

double average(int arr[ ])
{
  // How to code this?  How do we find out how many
  // array elements there are?
}
```

2 Function interface comments, continued

For Lab 2, you were asked to read the document called "Function Interface Comments", in the "Help Documents for Lab Assignments" module on D2L.

For this lab, Lab 3, please take a close look at Section 4.3 of that document—this section talks about pointer arguments used to supply addresses of arrays, which will show up in almost every lab from now until the end of the course.

3 Two frequently-made mistakes

The material in this section is really important to know! If you don't know it you will have a very hard time working with any real-world C code that uses arrays in any way.

In addition to potentially important career-related benefits, a good understanding of this material will save you a lot of time working on lab exercises, and a possibly a lot of marks on the midterm test.

3.1 A problem that is impossible to solve

Figure 1 is a nearly complete program to test a function called `average`; that function is supposed to find the average value of the elements of an array of `int`s. All that's left to do is to write a few lines of code to complete the function definition. The tricky part is that the function will need to find out how many elements are in the array. For example, with the given `main` function, the first call to `average` should add up 3 elements and do a division by 3.0, but the second call should add up 4 elements and do a division by 4.0.

Figure 2: Use of `sizeof` produces a wrong solution to the impossible problem.

```
double average(int arr[ ])
{
  // This approach is WRONG.  In fact, it's not just WRONG,
  // it's WRONG in a way that CAN'T BE FIXED!

  int n;  // We'll try to put the number of elements here.

  // Let's try to divide the size of the array in bytes
  // by the size of one int in bytes.  That should give us
  // the number of elements, no?
  n = sizeof(arr) / sizeof(int);

  double sum = 0.0;
  int i;
  for (i = 0; i < n; i++)
    sum += arr[i];
  return sum / n;
}
```

3.2 First mistaken approach: Trying to use `sizeof`

I've seen many students try to write something like the code in Figure 2. (Of course the students didn't write `WRONG` in all-caps in comments in their code.)

Here is the basic flaw: The parameter declaration `int arr [ ]`, no matter how much it looks like a declaration that `arr` is an array, really says that the type of `arr` is pointer-to-`int`. It would have been equivalent—and *much clearer* to a C beginner—to declare the parameter as `int *arr`. So the calculation that tries to find the number of elements in the array fails completely; it just divides the size of a pointer by the size of an `int`.

3.3 Second mistaken approach: Belief that all arrays are like strings

Another **incorrect** way to try to find the number of elements in an array of `int`s (or an array of `double`s, or an array of one of most other element types) is to hunt for the `'\0'` character that lies just beyond the last valid element in the array. This leads to code like what is shown in Figure 3.

The essential flaw is this: In C, strings are terminated with `'\0'`, but pretty much all other data stored in arrays is NOT terminated with `'\0'`, and in fact is not terminated with any kind of special element value at all.

It's unpleasant but true that the defective code of Figure 3 could, by accident, produce a correct result! Depending on what is sitting in memory just beyond the last element that ought to be added in to `sum`, it's possible that the comparison `arr[n] != '\0'` could be false at exactly the right time to deceive a programmer into believing that his or her code is correct.

3.4 From impossible to easy

There is **no way** to write a correct definition for `average` if the function prototype is this:

Figure 3: Hunting for `'\0'` produces a wrong solution to the impossible problem.

```
double average(int arr[ ])
{
  // This approach is WRONG!  It's based on an incorrect
  // generalization about C string handling.

  int n = 0; // We'll try to put the number of elements here.
  double sum = 0.0;
  while (arr[n] != '\0') {
    sum += arr[n];
    n++;
  }
  return sum / n;
}
```

```
double average(int arr[ ]);
```

In C, at a minimum, you'll need a second parameter to tell `average` how many array elements to look at:

```
double average(int arr[ ], int n);
```

However, your course instructors think that's poor style. You know that `arr` will be a pointer, so just say so—try hard not to confuse any reader of your code about the type of `arr`. Further, the function shouldn't modify any of the array element values accessed via `arr`, so let's enforce that with `const`. The function prototype should be:

```
double average(const int *arr, int n);
```

With that in place, and assuming that the value of `n` is greater than 0, the function definition is easy to write.

4 Exercise A: Practice with null-terminated strings

This is an in-lab exercise.

4.1 Read This First

As explained in lectures, a string in C is a sequence of character codes stored in an array of `char` elements, with a `'\0'` marking the end of the string.

It was also explained in lectures that—in most contexts in C code—a string constant such as `"YZ"` will generate a null-terminated string in the "static storage" region of memory.

4.2 What to Do

Download the file `lab3exA.c` from D2L. Study the C code.

The program will reach `point one` twice. Make memory diagrams for these two moments in time. In your diagrams, clearly label the stack and static storage regions.

Figure 4: Sizes of some frequently used types in Cygwin64.

`sizeof(char)`	1
`sizeof(short)`	2
`sizeof(int)`	4
`sizeof(long)`	8
`sizeof(size_t)`	8
`sizeof(float)`	4
`sizeof(double)`	8
`sizeof(char *)`	8
`sizeof(int *)`	8
`sizeof(double *)`	8

5 Exercise B: Two bad array functions and one good one

This is an in-lab exercise.

5.1 Read This First

5.1.1 `size_t`

The type of the value produced by a `sizeof` expression is called `size_t`, which is some kind of unsigned integer type. With 64-bit Cygwin (and also 64-bit Linux and Mac OS X) `size_t` is the same as `unsigned long`, which is a 64-bit unsigned integer type.

To print the value of something that has type `size_t` in a call to `printf`, it's best to use `%zu`—that will work on any C system that complies with the C99 standard.

5.1.2 Sizes of some common types in Cygwin64

Always remember this: Sizes of many types are not universal across all platforms that support C!

For example, a `long` is 8 bytes on Cygwin64, 64-bit Mac OS X and 64-bit Linux, but is only 4 bytes with Microsoft Visual C++, 32-bit Linux, and many C compilers for embedded systems.

So the information in Figure 4 is not true about all C code everywhere, but is true for the platform we use in the lab.

5.2 What to Do

Download the file `lab3exB.c` from D2L. Study the C code, then build an executable and run it.

Using the diagram in Figure 5 as a model, make AR diagrams for points one, two, three, and four, with annotations to show the sizes of all the local variables and function parameters. Indicate the sizes that these objects would have on Cygwin64.

(Don't show the string constants used in calls to `printf` in your diagrams—that would be tedious.)

Figure 5: Example program, with AR diagram for `point one`, enhanced to show the sizes of variables and parameters in bytes.

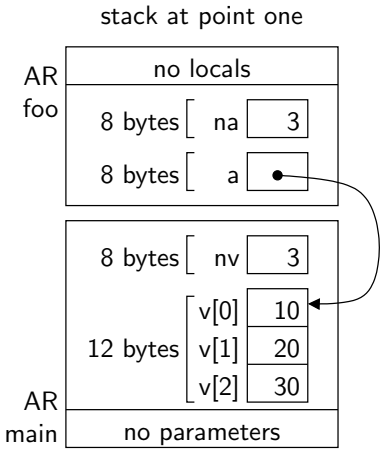
```
#include <stdio.h>

void foo(const int *a, size_t na);

int main(void)
{
  int v[ ] = { 10, 20, 30 };
  size_t nv;
  nv = sizeof(v) / sizeof(v[0]);
  printf("The array v has %zu elements.\n", nv);
  foo(v, nv);
  return 0;
}

void foo(const int *a, size_t na)
{
  // point one

  printf("The value of the last element "
         "in the array is %d.\n", a[na-1]);
}
```



6 Exercise C: More practice with arrays and pointers

This exercise won't be marked.

6.1 Read This First

Doing this unmarked exercise is highly recommended, as it may help you prepare for the midterm test, and may give you some hints about how to do Exercise E in this lab.

6.2 What to Do

Download the file `lab3exC.c` from D2L. Study the C code, then build an executable and run it.

Make AR diagrams for points one and two. You can check your work against solutions that will be posted on D2L sometime before 9:00am on Tue Oct 4.

7 Post-Lab Exercises

Check D2L later for complete Lab 3 instructions, including all the post-lab exercises.