# ENCM 339 Fall 2016 Lab 3 Complete Instructions

Steve Norman
Department of Electrical & Computer Engineering
University of Calgary

September 2016

**Attention:** This document describes all of the exercises of Lab 3.

## 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 |
| D | post-lab | 2 marks |
| E | post-lab | 8 marks |
| F | post-lab | 2 marks |
| G | post-lab | 4 marks |

Marks for post-lab exercises have not yet determined.

# 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.

## 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

**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?
}
```

**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;
}
```

**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;
}
```

`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:

```
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.

**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 |

## 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.

# 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.

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

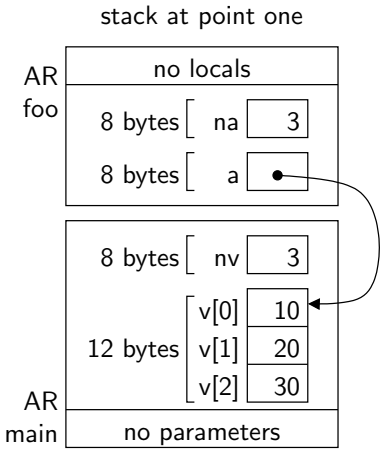
```
#include <stdio.h>

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]);
}
```




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

# 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 Exercise D: Crash!

*This is a post-lab exercise.*

## 7.1 Read This First

### 7.1.1 Null pointers

A *null pointer value* is an address that is definitely not the address of any data object. (By "data object" I mean any variable, array element, function parameter, or any other chunk of memory allocated for data storage.) One way to set up a null pointer in C is to use the `#define` macro `NULL`, which is defined in many different library header files.

### 7.1.2 Segmentation faults

On Cygwin, Linux, and Mac OS X, a *segmentation fault* is an event that normally causes a program to crash during run-time. The name is not very clear—a better name would be "attempt to access memory using invalid address".

Mistakes with pointers can cause segmentation faults when C programs run. Here are a few of the *many* ways to get a segmentation fault:

- Try to read or write memory through a null pointer.
- Try to read or write memory through an uninitialized pointer variable. Unfortunately this won't *always* cause a segmentation fault—by fluke, an uninitialized pointer may contain a valid address.
- Write an infinite loop that wanders way past the end of an array, eventually generating an invalid array element address, as in

```
for (i = 0 ; j < n; i++)
   s += a[i];
```

## 7.2 What to Do

Download the file `lab3exD.c` from D2L. Study the C code, then build an executable. You should note that the compiler does not issue any warnings or errors—everything is fine in terms of the C type system, and no variables get used before they are initialized or assigned to.

Run the executable. If you do it with Cygwin or a similar environment such as Linux or Mac OS X, you'll get a segmentation fault.

Alter the definition of `main` to fix the defect that is causing the program to crash. Copy your completed C code into your post-lab report, along with a copy of the output of the corrected program.

# 8 Exercise E: Functions to process arrays of `doubles`

*This is a post-lab exercise.*

## 8.1 Read This First

(This section repeats some ideas presented in the "Function Interface Comments" document. The ideas are important and not always easy to grasp, so the repetition may be useful.)

It's important to be able to understand and write function interface comments for functions that operate on built-in arrays. Here is a simple example:

```
double average(const int *a, size_t na);
// REQUIRES
//   na > 0.
```

```
//   Elements a[0], a[1], ..., a[na-1] exist.
// PROMISES
//   Return value is average of a[0], a[1], ..., a[na-1]
```

It should be obvious why `na > 0` is required—you can't take the average of an empty list of numbers. But what exactly does the next part mean? `Elements a[0], a[1], ..., [na-1] exist` is really a terse way of saying: `a` points to the element at the beginning of an array, and that array is big enough to have elements with indices up to and including `na-1` (and maybe even higher). The following code fragment shows some ways to satisfy the `REQUIRES` conditions, and some ways to violate those conditions:

```
int i;
int *p;
int x[6] = { 11, 22, 33, 44, 55, 66 };
double avg1, avg2, avg3, avg4, avg5, avg6;
avg1 = average(x, 6);     // OK: Average of all elements.
avg2 = average(x, 3);     // OK: Average of first 3 elements.
avg3 = average(&x[3],3);  // OK: Average of last 3 elements.
avg4 = average(p, 6);     // BAD: First arg is the address of who-knows-what?
avg5 = average(&i, 6);    // BAD: First arg is not the address of an array element.
avg6 = average(x, 8);     // BAD: The array only has 6 elements.
```

Unfortunately, you can't expect the compiler to catch these errors—all types match in the function calls. Nor is there any way to write code to check the condition on what `a` points to—all the function gets is an address, and there is no way in C or C++ to ask, "When I use this address to access array elements, is success guaranteed?"

## 8.2 What to Do

Download the file `lab3exE.c` from D2L. Study the C code. The `main` function here is an example of a *test harness*, which, rather than solving a problem for a computer user, solves a problem for a computer programmer. Specifically, it helps the programmer check that some functions she or he has written actually do what they are supposed to do.

Build an executable and run it. You should see that the outputs from the tests of `reverse` and `append` are all incorrect. That shouldn't be surprising, since the given definitions of those two functions don't actually do anything.

**Coding Step #1.** Add code to the definition of `reverse` so that the function does what it is supposed to do. Before moving on to **Step #2**, build an executable and run it to check your work. If your output looks wrong, go back and fix your definition of `reverse` before you move on.

**Coding Step #2.** Add code to the definition of `increasing` so that the function does what it is supposed to do. Don't move on to **Step #3** until you're sure that **Step #2** has been completed correctly.

**Coding Step #3.** Now think about a function called `min_element`, which should return the value of the least element in an array of `doubles`. Edit the `.c` file to add a function prototype, a function interface comment, a function definition, and some test code in `main` for `min_element`. Do at least four tests:

- All elements are negative, least element is not element 0. (By "element 0" I mean the element that has index 0.)
- All elements are positive, least element is not element 0.

- Some elements are negative, some are positive, and the least element is element 0.
- Some elements are negative, some are positive, and the least element is the last one in the array.

When all steps are complete, copy your `.c` file and its output into your post-lab report.

# 9 Exercise F: Low level string manipulation

*This is a post-lab exercise.*

## 9.1 Read This First

In ENCM 339 we'll consider only strings that are made using character codes from the ASCII character set, which is listed as a table in Figure 6.

ASCII is reasonably good for text in English, but is obviously not at all adequate for text in most other human languages. It's possible in C to manage strings that could contain any characters from any written language, but that's an advanced topic that won't be covered in ENCM 339.

Most of Figure 6 is self-explanatory, but there are a few things about it that can use a little extra explanation ...

- The codes corresponding to decimal number from 0 up to and including 31, and also 127, are called *control characters*. Most of them have very little use in modern computing, but some of them are very important, such as 0, the null character, 9, the tab character, 10, the newline character, and a few others.
- The table shows how to represent character 92, the backslash character, as a C character constant.
- Here is how to write a single quote as a character constant: `'\''`

On a somewhat related note, here is how to insert double quotes into a string constant: `"I'd like to say \"Hello!\" to my friends!"`

## 9.2 What to Do

Download the file `lab3exF.c` from D2L. Study the C code, then build an executable and run it to see what the output is.

Modify the program so that the output is

```
The string in buffer is "(42 || 0) && -3 has a value of 1."
```

Do this by looking up decimal values for ASCII codes in Figure 6 and typing them into the `.c` file as necessary—in this exercise you're not allowed to add character constants or string constants to the C code.

Copy your modified `.c` file into your post-lab report.

# 10 Exercise G: String concatenation

*This is a post-lab exercise.*

**Figure 6:** Table of the ASCII character set, copied from a Linux manual page.

| Oct | Dec | Hex | Char | Oct | Dec | Hex | Char |
|---|---|---|---|---|---|---|---|
| 000 | 0 | 00 | NUL '\0' | 100 | 64 | 40 | @ |
| 001 | 1 | 01 | SOH (start of heading) | 101 | 65 | 41 | A |
| 002 | 2 | 02 | STX (start of text) | 102 | 66 | 42 | B |
| 003 | 3 | 03 | ETX (end of text) | 103 | 67 | 43 | C |
| 004 | 4 | 04 | EOT (end of transmission) | 104 | 68 | 44 | D |
| 005 | 5 | 05 | ENQ (enquiry) | 105 | 69 | 45 | E |
| 006 | 6 | 06 | ACK (acknowledge) | 106 | 70 | 46 | F |
| 007 | 7 | 07 | BEL '\a' (bell) | 107 | 71 | 47 | G |
| 010 | 8 | 08 | BS '\b' (backspace) | 110 | 72 | 48 | H |
| 011 | 9 | 09 | HT '\t' (horizontal tab) | 111 | 73 | 49 | I |
| 012 | 10 | 0A | LF '\n' (new line) | 112 | 74 | 4A | J |
| 013 | 11 | 0B | VT '\v' (vertical tab) | 113 | 75 | 4B | K |
| 014 | 12 | 0C | FF '\f' (form feed) | 114 | 76 | 4C | L |
| 015 | 13 | 0D | CR '\r' (carriage ret) | 115 | 77 | 4D | M |
| 016 | 14 | 0E | SO (shift out) | 116 | 78 | 4E | N |
| 017 | 15 | 0F | SI (shift in) | 117 | 79 | 4F | O |
| 020 | 16 | 10 | DLE (data link escape) | 120 | 80 | 50 | P |
| 021 | 17 | 11 | DC1 (device control 1) | 121 | 81 | 51 | Q |
| 022 | 18 | 12 | DC2 (device control 2) | 122 | 82 | 52 | R |
| 023 | 19 | 13 | DC3 (device control 3) | 123 | 83 | 53 | S |
| 024 | 20 | 14 | DC4 (device control 4) | 124 | 84 | 54 | T |
| 025 | 21 | 15 | NAK (negative ack.) | 125 | 85 | 55 | U |
| 026 | 22 | 16 | SYN (synchronous idle) | 126 | 86 | 56 | V |
| 027 | 23 | 17 | ETB (end of trans. blk) | 127 | 87 | 57 | W |
| 030 | 24 | 18 | CAN (cancel) | 130 | 88 | 58 | X |
| 031 | 25 | 19 | EM (end of medium) | 131 | 89 | 59 | Y |
| 032 | 26 | 1A | SUB (substitute) | 132 | 90 | 5A | Z |
| 033 | 27 | 1B | ESC (escape) | 133 | 91 | 5B | [ |
| 034 | 28 | 1C | FS (file separator) | 134 | 92 | 5C | \ '\\' |
| 035 | 29 | 1D | GS (group separator) | 135 | 93 | 5D | ] |
| 036 | 30 | 1E | RS (record separator) | 136 | 94 | 5E | ^ |
| 037 | 31 | 1F | US (unit separator) | 137 | 95 | 5F | _ |
| 040 | 32 | 20 | SPACE | 140 | 96 | 60 | ` |
| 041 | 33 | 21 | ! | 141 | 97 | 61 | a |
| 042 | 34 | 22 | " | 142 | 98 | 62 | b |
| 043 | 35 | 23 | # | 143 | 99 | 63 | c |
| 044 | 36 | 24 | $ | 144 | 100 | 64 | d |
| 045 | 37 | 25 | % | 145 | 101 | 65 | e |
| 046 | 38 | 26 | & | 146 | 102 | 66 | f |
| 047 | 39 | 27 | ' | 147 | 103 | 67 | g |
| 050 | 40 | 28 | ( | 150 | 104 | 68 | h |
| 051 | 41 | 29 | ) | 151 | 105 | 69 | i |
| 052 | 42 | 2A | * | 152 | 106 | 6A | j |
| 053 | 43 | 2B | + | 153 | 107 | 6B | k |
| 054 | 44 | 2C | , | 154 | 108 | 6C | l |
| 055 | 45 | 2D | - | 155 | 109 | 6D | m |
| 056 | 46 | 2E | . | 156 | 110 | 6E | n |
| 057 | 47 | 2F | / | 157 | 111 | 6F | o |
| 060 | 48 | 30 | 0 | 160 | 112 | 70 | p |
| 061 | 49 | 31 | 1 | 161 | 113 | 71 | q |
| 062 | 50 | 32 | 2 | 162 | 114 | 72 | r |
| 063 | 51 | 33 | 3 | 163 | 115 | 73 | s |
| 064 | 52 | 34 | 4 | 164 | 116 | 74 | t |
| 065 | 53 | 35 | 5 | 165 | 117 | 75 | u |
| 066 | 54 | 36 | 6 | 166 | 118 | 76 | v |
| 067 | 55 | 37 | 7 | 167 | 119 | 77 | w |
| 070 | 56 | 38 | 8 | 170 | 120 | 78 | x |
| 071 | 57 | 39 | 9 | 171 | 121 | 79 | y |
| 072 | 58 | 3A | : | 172 | 122 | 7A | z |
| 073 | 59 | 3B | ; | 173 | 123 | 7B | { |
| 074 | 60 | 3C | < | 174 | 124 | 7C | \| |
| 075 | 61 | 3D | = | 175 | 125 | 7D | } |
| 076 | 62 | 3E | > | 176 | 126 | 7E | ~ |
| 077 | 63 | 3F | ? | 177 | 127 | 7F | DEL |

## 10.1 Read This First

You should know by now that a C string is stored within an array of `char`s, that the end of a string is marked by the null character, and that the C character constant for the null character is `'\0'`.

The library header file `<string.h>` contains prototypes for common string-related operations, such as copying a string, finding the length of a string, finding the location of a particular character within a string, and so on.

The "cat" in `strcat` stands for *concatenation*, which means appending one string to the end of another. Here is a very brief program that uses `strcat`:

```
#include <string.h>
int main(void)
{
  char s[8];
  strcpy(s, "foo");
  strcat(s, "bar");
  return 0;
}
```

This is what the array `s` would look like before and after the call to `strcat`:

before call to `strcat`

| 'f' | 'o' | 'o' | '\0' | ?? | ?? | ?? | ?? |
|---|---|---|---|---|---|---|---|
| s[0] | | | | | | | s[7] |

after call to `strcat`

| 'f' | 'o' | 'o' | 'b' | 'a' | 'r' | '\0' | ?? |
|---|---|---|---|---|---|---|---|
| s[0] | | | | | | | s[7] |

Although they are both part of the standard C library, neither `strcpy` nor `strcat` are safe! For example, if the `strcat` call in the above example were

```
strcat(s, "abcdefghijklmnopqrstuvwyz");
```

then `strcat` would try to write many characters outside of the array `s`, with an unpredictable but probably destructive and confusing effect. You should look for and use safer alternatives to `strcpy` and `strcat` if you're ever writing C code in which absolute security and reliabililty are needed. Nevertheless, `strcpy` and `strcat` are important parts of the history of C, and it is important for you to understand how they work.

## 10.2 What to Do

Download the file `lab3exG.c` from D2L. Study the C code, then build an executable and run it to see what the output is.

Modify the definition for `safecat` so that it does what the function interface comment says it does. The test code in `main` should help you check whether your modifications have worked, but always keep in mind that tests usually can't *prove* that code is correct!

Copy your final version of `lab3exG.c` into your post-lab report.