Assignment: Linear Algebra and Arrays

Learning Outcomes

- Using arrays to store homogeneous data
- Looping over arrays
- Implementing functions with array parameters
- Implementing basic linear algebraic computations using arrays
- Providing an introduction to make and *makefile*s

Task: Linear Algebra Operations

This is a short assignment to further help your understanding of arrays, functions, and iteration [you can use any of the iteration statements provided by C]. The program will manipulate arrays in several different ways using the following linear algebraic functions:

```c
// Given an array, reverse the order of elements in array.
void reverse_array(int a[], int size);

// Add elements of first two arrays and put their sum in third array.
void add_arrays(int const a[], int const b[], int c[], int size);

// Given an array and a multiplier, multiply each element by multiplier.
void scalar_multiply(int a[], int size, int multiplier);

// Given two arrays, return the dot product.
// Dot product means sum of products, i.e., multiply each
// corresponding element of 2 arrays and sum these products.
int dot_product(int const a[], int const b[], int size);

// Given three arrays, determine the cross product of the first two.
// The cross product is another array and will be placed into third array.
// The size of all three arrays will always be at least three.
void cross_product(int const a[], int const b[], int c[]);

// Return length or magnitude of array with size element.
double length(int const a[ ], int size);
```

> ***You're NOT to create any arrays in any of the functions you write. All arrays that you need are given to you as function parameters. No credit will be given for any function that defines an array.***

Vector Addition

Given two vectors $\overrightarrow{a}$ and $\overrightarrow{b}$ of the same size, the addition of these vectors is given by a third vector $\overrightarrow{c} = \overrightarrow{a} + \overrightarrow{b}$ with the components of $\overrightarrow{c}$ obtained by adding the corresponding components of $\overrightarrow{a}$ and $\overrightarrow{b}$. The addition operation is shown in the following equation, which assumes there are n elements in vectors $\overrightarrow{a}$ and $\overrightarrow{b}$:

$$\overrightarrow{c} = \overrightarrow{a} + \overrightarrow{b} = \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ b_{n-1} \end{bmatrix} = \begin{bmatrix} a_0 + b_0 \\ a_1 + b_1 \\ a_2 + b_2 \\ \vdots \\ a_{n-1} + b_{n-1} \end{bmatrix}$$

Scaling a Vector

Scaling a vector means keeping its orientation the same but changing its length by a scale factor. It is like changing the scale of a picture; the object expands if the scale factor is greater than 1 or shrinks if the scale factor is smaller than 1, but the directions remain the same. The scale operation is shown in the following equation, which assumes there are n elements in vectors $\overrightarrow{a}$ and $\overrightarrow{b}$:

$$\overrightarrow{b} = k \cdot \overrightarrow{a} = k \cdot \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} k \cdot a_0 \\ k \cdot a_1 \\ k \cdot a_2 \\ \vdots \\ k \cdot a_{n-1} \end{bmatrix}$$

Dot Product

The dot product is a number computed from two vectors of the same size. This value is the sum of the products of the values in corresponding positions in the vectors, as shown in the summation equation, which assumes there are n elements in the vectors $\overrightarrow{a}$ and $\overrightarrow{b}$:

$$\text{Dot Product} = \overrightarrow{a} \bullet \overrightarrow{b} = \sum_{k=0}^{n-1} a_k b_k$$

To illustrate, assume $\overrightarrow{a} = \begin{bmatrix} 1 \\ -4 \\ 5 \end{bmatrix}$ and $\overrightarrow{b} = \begin{bmatrix} 3 \\ 1 \\ 2 \end{bmatrix}$. The dot product $\overrightarrow{a} \bullet \overrightarrow{b}$ is mathematically expressed like this:

$$\overrightarrow{a} \bullet \overrightarrow{b} = \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} \bullet \begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix} = a_0 \cdot b_0 + a_1 \cdot b_1 + a_2 \cdot b_2$$

Expanding the example

$$\begin{bmatrix} 1 \\ -4 \\ 5 \end{bmatrix} \bullet \begin{bmatrix} 3 \\ 1 \\ 2 \end{bmatrix} = (1)(3) + (-4)(1) + (5)(2) = 3 + (-4) + 10 = 9$$

Note that the vectors can be of any size, but both will be the same size. So, if both vectors had 100 elements:

$$\overrightarrow{a} \bullet \overrightarrow{b} = \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{99} \end{bmatrix} \bullet \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ b_{99} \end{bmatrix} = (a_0 \cdot b_0) + (a_1 \cdot b_1) + (a_2 \cdot b_2) + \cdots + (a_{99} \cdot b_{99})$$

Cross Product

The cross product evaluates to an array and only works on arrays of size 3. So, if we have two arrays $\overrightarrow{a} = \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix}$ and $\overrightarrow{b} = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix}$ of 3 integers each, the cross product of vectors $\overrightarrow{a}$ and $\overrightarrow{b}$ is mathematically defined as:

$$\overrightarrow{c} = \overrightarrow{a} \times \overrightarrow{b} = \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} \times \begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} a_1 b_2 - a_2 b_1 \\ -(a_0 b_2 - a_2 b_0) \\ a_0 b_1 - a_1 b_0 \end{bmatrix}$$

If $\overrightarrow{a} = \begin{bmatrix} 10 \\ 9 \\ -7 \end{bmatrix}$ and $\overrightarrow{b} = \begin{bmatrix} -2 \\ 4 \\ -5 \end{bmatrix}$, $\overrightarrow{a} \times \overrightarrow{b}$ is computed like this:

$$\overrightarrow{c} = \overrightarrow{a} \times \overrightarrow{b} = \begin{bmatrix} 10 \\ 9 \\ -7 \end{bmatrix} \times \begin{bmatrix} -2 \\ 4 \\ -5 \end{bmatrix} = \begin{bmatrix} (9)(-5) - (-7)(4) \\ -((10)(-5) - (-7)(-2)) \\ (10)(4) - (9)(-2) \end{bmatrix} = \begin{bmatrix} -17 \\ 64 \\ 58 \end{bmatrix}$$

Vector Magnitude

The magnitude or length of a vector $\overrightarrow{a}$ with n elements is a scalar value $\left\| \overrightarrow{a} \right\|$ that is defined as:

$$\left\| \overrightarrow{a} \right\| = \left\| \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix} \right\| = \sqrt{a_0^2 + a_1^2 + a_2^2 + \cdots + a_{n-1}^2}$$

Implementation Details

Open a Window command prompt, change your directory to $\mathrm{C:\backslash sandbox}$ [create the directory if it doesn't exist], create a sub-directory $\mathrm{ass06}$, and launch the Linux shell. Download driver source file $\mathrm{qdriver.c}$ and incomplete source and header files $\mathrm{q.c}$ and $\mathrm{q.h}$, respectively.

Function declarations in $\mathrm{q.h}$

Using Visual Code, open header file $\mathrm{q.h}$ and add file-level and function-level documentation blocks and declarations of the necessary functions. Do not include C standard library headers in $\mathrm{q.h}$ unless the function declarations in $\mathrm{q.h}$ rely on types declared in the C standard library. Why? Suppose you unnecessarily include header files in $\mathrm{q.h}$ and your clients in turn include $\mathrm{q.h}$ in their source files. When clients' source files are compiled, the preprocessor will include the unnecessary C standard header files into these source files by copying and pasting hundreds of lines from unused header files into the source files. This will greatly increase compile times causing great annoyance to your clients. Instead, include any C standard header files required to define the functions directly in $\mathrm{q.c}$.

Test your header file $\mathrm{q.h}$ by compiling [only] driver source file $\mathrm{qdriver.c}$ [which includes $\mathrm{q.h}$]:

```
$ gcc -std=c11 -pedantic-errors -Wstrict-prototypes -Wall -Wextra -Werror -c
qdriver.c -o qdriver.o
```

Stub functions in q.c

As explained in previous assessments, begin by implementing *stub functions* in q.c for the functions declared in q.h. A stub is a skeleton of a function that is called and immediately returns. It is syntactically correct - it takes the correct parameters and returns the proper values. The stub function for function `dot_product` would look like this:

```
// return a value of type int to ensure that the definition is syntactically
// correct and will compile although with diagnostic warning messages because
// the parameters are unused in the body of the function
int dot_product(int const a[], int const b[], int size) {
  return 0;
}
```

Although a stub is a complete function, it does nothing other than to establish and verify the linkage between the caller and itself. But this is a very important part of coding, testing, and verifying a program. After implementing all of the necessary stub functions, it is possible to compile, link, and execute your program. Compile [only] your source file q.c using the full suite of gcc options:

```
$ gcc -std=c11 -pedantic-errors -Wstrict-prototypes -Wall -Wextra -Werror -
Werror=vla -c q.c -o q.o
```

A driver source file qdriver.c [which includes q.h] is provided to test your definitions. Separately compile [only] the driver source file qdriver.c. Since you've only defined stub functions, parameters in these stub functions are unused causing the -Werror option to terminate compilation [with unused parameter error messages]. Therefore, with stub functions, you will need to temporarily drop the -Werror option to successfully compile q.c [but with warnings]:

```
$ gcc -std=c11 -pedantic-errors -Wstrict-prototypes -Wall -Wextra -Werror=vla
-c qdriver.c -o qdriver.o
```

Link both these object files plus C standard library functions [such as `printf`] into an executable file:

```
$ gcc q.1 o qdriver.o -o q.out
```

Since you've only implemented stub functions, the output from your program will not be correct. Before replacing the stub definitions with the actual definitions, let's first learn about a program called make that will make it easier to compile and link both small projects [similar to this assignment] to larger projects containing hundreds of source files.

make and *Makefiles*

The previous section has emphasized the necessity for using a variety of gcc options to ensure code is cleanly compiled without any warnings. Typing the entire set of required options each time can be cumbersome and annoying. When writing complex programs consisting of multiple [think tens or hundreds or thousands] source files, making small changes to a few files will require the many source files to be recompiled. These recompilations may occur hundreds of times every day causing substantial delays as programmers wait for the executable to be created. More importantly, programmers will have to remember dependencies between different files. For

example, if source file b.c includes header file a.h which in turn includes another header file c.h, and if c.h is updated, then b.c must be recompiled even though neither b.c nor a.h were altered.

It can be difficult to remember the entire list of source files and the dependencies required to create an executable from them. To solve this problem, a program called make is used. The version of make provided by gcc is coincidentally called make. make is a facility for automating maintenance and building executables from source files. make uses a *makefile* that specifies the dependencies between files and the commands that will bring all files up to date and build an executable from these up to date files. In short, *makefile* contains the following information:

- the name of source and header files comprising the program
- the interdependencies between these files
- the commands that are required to create the executable

A simple *makefile* consists of *rules* with each *rule* consisting of three parts: a *target*, a list of *prerequisites*, and a *command*. A typical rule has the form:

```
target : prereq-1 prereq-2 ...
  command1
  command2
  ...
```

target is the name of the file to be created or an action to be performed by make. prereq-1, prereq-2, and so on represent the files that will be used as input to create target. If any of the prerequisites have changed more recently than target, then make will create target by executing commands command1, command2, and so on. make will terminate and shutdown if any command is unsuccessful.

> ***Note that every command must be preceded by a tab and not spaces!!!***

Here's an example:

```
example.out : main.o file1.o file2.o
gcc -std=c11 -pedantic-errors -Wstrict-prototypes -Wall -Wextra -Werror main.c
file1.c file2.c -o example.out
```

Line 1 says that target example.out must be remade [or made if it doesn't exist] if any of the prerequisite files [main.o, file1.o, file2.o] have been changed more recently than the target. Before checking the times prerequisite files were changed, make will look for rules that start with each prerequisite file. If such a rule is found, make will make the target if any of its prerequisites are newer than the target. After checking that all prerequisite files are up to date and remaking any that are not, make brings example.out up to date.

Line 2 tells make how it should remake target example.out. This involves calling gcc with the usual and required options to compile and link source files main.c, file1.c, and file2.c.

A *makefile* can also contain *macro definitions* where a macro is simply a name for something. A macro definition has the form:

```
NAME = value
```

The value of macro NAME is accessed by either $(NAME) or ${NAME}. make will replace every occurrence of either $(NAME) or ${NAME} in *makefile* with value.

Here's a complete annotated example of a *makefile*:

```
# makefile for example.out
# the # symbol means the rest of the line is a comment

# this is definition of macro GCC_OPTIONS
GCC_OPTIONS = -std=c11 -pedantic-errors -Wstrict-prototypes -Wall -Wextra -
Werror
# this is definition of macro OBJS
OBJS = main.o file1.o file2.o

# this rule says that target example.out will be built if prerequisite files
# main.o file1.o file2.o file3.o have changed more recently than example.out
# the text $(OBJS) will be substituted with list of options in line 7
# the next line says to build example.out using command gcc
# the text $(GCC_OPTIONS) will be substituted with list of options in line 5
example.out : $(OBJS)
  gcc $(GCC_OPTIONS) $(OBJS) -o example.out

# the next line says main.o depends on main.c
# the line after it says to create main.o with the command gcc
main.o : main.c
  gcc $(GCC_OPTIONS) -c main.c -o main.o

# file1.o depends on both file1.c and file1.h
# and is created with command gcc $(GCC_OPTIONS) -c file1.c -o file1.o
file1.o : file1.c file1.h
  gcc $(GCC_OPTIONS) -c file1.c -o file1.o

# file2.o depends on both file2.c and file1.h
file2.o : file2.c file1.h
  gcc $(GCC_OPTIONS) -c file2.c -o file2.o

# clean is a target with no prerequisites;
# typing the command in the shell: make clean
# will only execute the command which is to delete the object files
clean :
  rm $(OBJS)
```

Assuming the *makefile* is saved as a file named makefile, you can use make to create the executable example.out like this:

```
$ make
```

Target clean on line 34 is different from the other targets; it has no prerequisites. If the following command is issued in the shell:

```
$ make clean
```

then make will execute only the command on line 35 in rule clean and then exit.

Let's conclude this section by writing a simple *makefile* for this tutorial consisting of two source files qdriver.c and q.c and a header file q.h that is included in both source files. The default name of *makefile* is either makefile or Makefile; other names can be used but make must be provided the non-default name of the *makefile*.

```
GCC_OPTIONS = -std=c11 -pedantic-errors -Wstrict-prototypes -Wall -Wextra -
Werror -Werror=vla
OBJS = qdriver.o q.o
EXEC = q.out

$(EXEC) : $(OBJS)
  gcc $(GCC_OPTIONS) $(OBJS) -lm -o $(EXEC)

qdriver.o : qdriver.c q.h
  gcc $(GCC_OPTIONS) -c qdriver.c -o qdriver.o

q.o : q.c q.h
  gcc $(GCC_OPTIONS) -c q.c -o q.o

clean:
  rm $(OBJS) $(EXEC)

```

Test makefile with source file qdriver.c [that is implemented for you] and your source file q.c that only has stub functions defined, like this:

```
$ make
```

The most common error with a *makefile* is programmers forgetting to put a horizontal tab at the beginning of a command line, and instead placing space characters there. Here's what happens if line 12 is prefixed with space characters rather than a tab:

```
Makefile:12: *** missing separator. Stop.

```

Implementation and testing

At this point, you've qdriver.c, you've defined stub functions in q.c that were declared in q.h, and you have a *makefile* called makefile. To reduce debugging time, employ a process of writing a function and thoroughly verifying the correctness of your definition before moving to the next function. To test a function, you should know what input is given to the function and the expected output from the function. Perform hand calculations to determine the output for the input provided to the function. To pass the input test(s) to your definitions, you may have to alter qdriver.c. Do so without hesitation because a copy is easily accessible on the assignment web page. Always test for boundary conditions that might cause unspecified behavior. If your definition involves a division, is there input that could generate a *division by zero* error. What happens if the input arrays contain 0 values? Record these boundary conditions and the function's response in function headers so that your clients are aware of your responses to these scenarios.

The following code fragment illustrates the expected behavior of the functions - note the inputs to each function and their corresponding outputs. Every function provides the array's size as a parameter except function `cross_product`. If there is more than one array parameter, you can assume all arrays have the same size. Linear algebra defines cross products only for three-dimensional vectors and therefore when defining this function you can assume array parameters have sizes of at least 3.

```
#define ARRAY_SIZE 5

int a[] = {1, 2, 3, 4, 5, 6, 7, 8};
reverse_array(a, sizeof(a)/sizeof(a[0]));
// after reversing, a will be: 8 7 6 5 4 3 2 1

// here, the author of add_arrays is assuming that all three arrays have
// size of at least ARRAY_SIZE - some may have size more than ARRAY_SIZE,
// but there is no array with size less than ARRAY_SIZE
int b[ARRAY_SIZE] = {3, 4, 7, 2, 1}, c[ARRAY_SIZE];
add_arrays(a, b, c, ARRAY_SIZE);
// now, first 5 elements of c will contain: 11 11 13 7 5

scalar_multiply(a, ARRAY_SIZE, 8);
// now, a will contain these values: 64 56 48 40 32 3 2 1

int dp = dot_product(a, b, 3);
// dot product of first three elements of a(64, 56, 48) and
// first three elements of b(3, 4, 7) gives a dot product of 752

// so far, b's first 3 elements have values 3 4 7
// while c's first 3 elements have values 11 11 13
cross_product(b, c, a);
// the result will write values -25 38 -11 into first 3 elements of a

// magnitude of b(3,4,7,2,1) when printed with f format specifier is
8.888194
double mag = length(b, 5);
```

After implementing the functions, it is possible that you may have altered qdriver.c by adding diagnostic `printf` statements. Remove [or comment out] such diagnostic statements. Also, make sure to turn on option -Werror if it was previously removed during the debugging phase. Build an executable like this:

```
$ make
```

When executing program q.out, redirect the program's output to a text file your-output.txt:

```
$ ./q.out > your-output.txt
```

You're given text file output.txt representing the correct output generated by the driver qdriver.c. Your implementation's output must *exactly* match output.txt. Test your output using the diff command, like this:

```
$ diff -y --strip-trailing-cr --suppress-common-lines your-output.txt
output.txt
```

Options -y, --strip-trailing-cr, and --suppress-common-lines are described here. If diff is not silent, then one or more of your function definitions is incorrect and will require further work.

File-level and Function-level documentation

Every source and header file you submit must contain file-level documentation blocks whose purpose is to provide human readers [yourself and other programmers] useful information about the purpose of this source file at some later point of time.

Every function that you declare in a header file [and define in a corresponding source file] must contain a function-level documentation block.

> ***Don't copy and paste documentation blocks from previous assignments. Annoyed graders will definitely subtract grades to the full extent specified in the rubrics below when they detect such copy-and-paste scenarios.***

Submission and automatic evaluation

1. In the course web page, click on the submission page to submit required files.
2. Read the following rubrics to maximize your grade. Your submission will receive:
3. F grade if your submission doesn't compile with the full suite of gcc options [shown above].
4. F grade if your submission doesn't link to create an executable.
5. $A+$ grade if the submission's output matches the correct output. Otherwise, a proportional grade is assigned based on how many incorrect results were generated by your submission.
6. A deduction of one letter grade for each missing documentation block. Every submitted file must have one file-level documentation block. Every function that you declare in a header file must provide a function-level documentation block. A teaching assistant will physically read submitted source files to ensure that these documentation blocks are authored correctly. Each missing block will result in a deduction of a letter grade. For example, if the automatic grader gave your submission an $A+$ grade and three documentation blocks are missing, your grade will be later reduced from to $A+$ to $B+$.
   Another example: if the automatic grade gave your submission a C grade and three documentation blocks are missing, your grade will be later reduced from C to F.