CPE-2600 FALL 2024

Systems Programming

Lab 5 Vector Calculator

Introduction

The purpose of this assignment is to design, code, and test an interactive C program that can perform vector math. It is intended to familiarize you with using structures, arrays, and console input and output C.

Work on the assignment is to be done ***individually***. You are welcome to collaborate with class members, but the project must be your own work.

Background and References

A vector is a quantity that represents both a magnitude and direction. It is used in many fields to represent different things. For example, a vector could be used to indicate a force applied to an object at a given angle.

Different mathematical transformations can be performed against vectors for various reasons. The goal of this assignment is for you to write a program that will perform different mathematical operations on one or more vectors.

There are a variety of ways to represent vectors. As mentioned, notation for vectors in a 2D space would be a magnitude and direction, or "polar" notation. It is also common to represent a vector quantity in terms of X and Y components, or "rectangular" coordinates. When moving to a 3D space, using Polar notation becomes difficult due to the handling of multiple angles, so 3D rectangular coordinates (X, Y, Z) are most often used.

For more detailed examples and explanations (in 2D) see:

- Introduction to Vectors – https://www.mathsisfun.com/algebra/vectors.html

Project Description

The goal of this project will be to create an interactive calculator for vectors. It will be reminiscent of a "mini-Matlab" program. Given our current skills, there will be come inherent limitations, but will be continue to learn techniques to overcome those limitations.

One limitation is that we will hardcode in that all vectors have three components. We will also only be able to handle a handful of vectors and implement just a few operations.

The following sample interaction will best communicate how the program will operate:

```
minimat> a = 1 2 3
a = 1   2   3
minimat> b = 4, 5, 6
b = 4   5   6
minimat> c = a + b
c = 5   7   9
minimat> d = c * 2
d = 10   14   18
minimat> a + d
ans = 11   16   21
minimat> b
b = 4   5   6
```

While this is just a sample, there are many things to note. First of all, `minimat>` for this example is just a prompt. After the prompt, we are interactively assigning values to vectors a and b. Note the , and/or space delimiters. Next, a and b are added together and the result is assigned to c. c undergoes a scalar multiplication and the result is assigned to d. Next, a and d are added together, but the result is not assigned to a specific vector but the sum is displayed regardless. Finally, b is entered with no operation and the current value is shown.
Not shown, there should be a command to quit the program. Also not shown, there should be a command (clear) to clear out the vector memory.

Consider the following mathematical operations that can be performed with vectors. Some of the descriptions below show 2D vectors, but remember that we will be doing all 3D vectors.

Add

Add two vectors together using the formula:

If A = (a_i) vector of length *n* and B = (b_i) is vector of length n, then the sum A + B is also a vector of length *n* such that A + B = $(a_i + b_i)$

The definition of vector addition indicates an element-by-element addition. The elements of vector A are added to the exact corresponding elements of vector B. To add A and B they must be the same dimensions.

For example:

```
A = (1, 4)
B = (1, 3)
A + B = (1 + 1, 4 + 3) = (2, 7)
```

Subtract

Subtract two vectors using the formula:

If A = (a_i) vector of length *n* and B = (b_i) is vector of length n, then the difference A - B is also a vector of length *n* such that A - B = (a_i - b_i)

The definition of vector subtraction indicates an element-by-element subtraction. The elements of vector A are subtracted by the exact corresponding elements of vector B. To subtract A and B they must be the same dimensions.

For example:

```
A = (1, 4)
B = (1, 3)
A - B = (1 - 1, 4 - 3) = (0, 1)
```

Vector Multiplication

Multiplying two vectors comes in two forms – dot product and cross product. This information is left here for completeness, but our program will not need to implement either of these operations right now.

Dot Product

The dot product results in a scalar value using the following formula:
If A = (a_i) vector of length *n* and B = (b_i) is vector of length n, then the dot product A · B is a scalar such that A · B = $a_1b_1 + a_2b_2 + \ldots + a_nb_n$

The definition of dot product indicates an element-by-element multiplication. The elements of vector A are multiplied by the exact corresponding elements of vector B and all results are added together.

For example:

```
A = (1, 4)
B = (1, 3)
A · B = 1 * 1 + 4 * 3 = 13
```

Cross Product

The cross-product results in a vector that is perpendicular to each of the source vectors A and B. A cross product must be performed on vectors of 3 dimensions.
If A = (a_1, a_2, a_3) and B = (b_1, b_2, b_3), then the cross product A x B is a vector such that A x B = (c_1, c_2, c_3) where:

$c_1 = a_2 * b_3 - a_3 * b_2$
$c_2 = a_3 * b_1 - a_1 * b_3$
$c_3 = a_1 * b_2 - a_2 * b_1$

NOTE: the ordering may not be inherently obvious, it doesn't really look like it follows a pattern.

For example:

```
A = (1, 2, 3)
B = (4, 5, 6)
A x B = (2 * 6 - 3 * 5, 3 * 4 - 1 * 6, 1 * 5 - 2 * 4) = (-3, 6, -3)
```

Scalar Multiplication

Simple enough, scalar multiplication is between one vector and one non-vector (aka scalar) value. Each element of the vector is multiplied by the scalar.

For example:

```
A = (1, 2, 3)
A * 2.5 = (1*2.5, 2*2.5, 3*2.5)
```

Obviously the result is a vector and can be assigned to a new vector if desired. For scalar multiplication, num * vector and vector * num should produce the same result.

Implementation Details

In summary, your task is to create a program that will perform limited vector math. The required operations are vector addition, vector subtraction, and scalar multiplication. Dot product and cross product are detailed above, but are not required.

Your implementation must represent vectors internally in a `struct`. That struct must have a member for the vector's name and member(s) for it's three magnitudes. The name could be a reasonably size character array to hold a C-style string. Alternatively, you could limit the name to a single character. For the magnitudes, you should be using floating point math, so you should use type `double` - either three members or an array would work for this.
Your implementation must be able to store up to 10 individual vectors at the same time, but the names and values must be able to be added at runtime as per the sample interaction above. I suggest you create an array of your vector structs. As you create new vectors, you locate (hint - iterate the array) the first empty location and put the vector there. Once this array is filled, if a new array is attempted to be created, the program should refuse and print a memory full message to the user. If the user issues the `clear` command, the storage array will be cleared and new vectors can be made.
I recommend roughly three layers for your implementation.

The inner-most layer will perform the vector math operations. Recall that structs pass by value (unless you choose to pass the pointer to a struct), so these functions may be relatively simple.

```
// vect is a typedef'ed struct
vect add(vect a, vect b)
{
    vect returnval;
    returnval.x = a.x + b.x;
    ...

    return returnval;
}
```

The next layer would manage the storage array that holds the vector structs. Perhaps it implements some container-like functionality like:

- addvect(vect new) - add new vector to storage array. If same name exists, replace at that location, otherwise add to empty location. If array full, do nothing and return error code.
- findvect(char * name) - search array for name vector and return (a copy) to caller
- clear() - empty the storage array
- ?? maybe a few more...

Since you have several functions accessing the same array, it will be reasonable to make the array visible to all functions in this layer. We do not like global variables, so, what might work well?

And finally, the user interface layer. This layer would be responsible for parsing the commands from the user, calling the storage array functions to store and retrieve vectors, calling the operations layer to add, subtract, and multiply vectors, and display results to user and/or error messages. All input and output from and to the console should be in this layer.

Main should not really have a lot of code. We have a single command line option (see below). Excepting that, main should really just hand off control to your user interface.

As a whole, this sounds like a huge project, but it really is not. If you work from the simplest functions out to the user interface you will find much of the program is trivial. Be sure to test each function and layer as you go. Think unit testing.

Summary Requirements

1. In many ways, the user interaction will be similar to Matlab...
2. The program may be invoke with no arguments and will present a prompt to the user. The program will accept a command/expression from the user and return to this prompt after executing the command (except quit).
3. The program may be invoked with the command line argument `-h` for which it will display help text with a summary of usage.
4. Exit Command: `quit` - this command will cause the program to exit gracefully.
5. Assignment: `varname = VALx, VALy, VALz` - this expression will create or replace the vector varname in storage. varname can be a string or single character per your design. VALx, VALy, and VALz may be floating point values and may be separated by a space or comma. Other whitespace should be ignored. **UPDATE - You may "require" spaces before and after the =. This should make the expression easier to parse. See note below.** OPTIONAL - if VALz is omitted that components should be set to 0.0. All of these should be valid - `a = 1,2,3`, `b = 1 2 3`, `c = 1.2 2.4, 5.6`. If an invalid assignment expression is entered, the user should be informed and the program will return to the command prompt. As noted above, the number of vectors your program must support is to be hardcoded to support 10 vectors.
6. Display: `varname` - typing just a variable name should display the current value of that vector to the console. If varname does not yet exist a message to that effect should be displayed.

7. Addition: `var1 + var2` - this expression will display the sum of the two supplied vectors. The vectors must exist or an error message should be displayed. **UPDATE - You may "require" spaces before and after the +. This should make the expression easier to parse. See note below.**
8. Subtraction: `var1 - var2` - this expression will display the difference of the two supplied vectors. The vectors must exist or an error message should be displayed. **UPDATE - You may "require" spaces before and after the -. This should make the expression easier to parse. See note below.**
9. Scalar Multiplication: `var1 * num` or `num * var1` - this expression will display the vector scalar multiplied by num which may be any valid value afforded by floating point numbers (double). The vector must exist or an error message should be displayed. **UPDATE - You may "require" spaces before and after the operator. This should make the expression easier to parse. See note below.**
10. Operation plus assignment: `result = var1 + var2` - performs the operation indicated (addition, subtraction, or scalar multiplication) and assigns the result to indicated result vector. If the result vector does not exist it will be created and if there is no room a message will be displayed. If the result vector exists, it will be replaced with the new values. In any case, the result of the operation should also be displayed. **UPDATE - You may "require" spaces before and after the = and the operator. This should make the expression easier to parse. See note below.**
11. Command: `clear` - empties all of the stored vectors.
12. Command: `list` - lists all of the stored vectors and their values.
13. Display format - in all cases, vectors should be displayed with readable formatting. The display should include the vector's name followed by the three values. You may consider limiting the precision of the floating point values.
14. OPTIONAL - implement the dot and cross products.

NOTE ON PARSING: Parsing user input is hard. Without the spaces, an expression like `c=a+b` might be challenging to parse. At this point in your skills, I am afraid you spending an inordinate amount of time on the problem, hence the update to allow your implementation to expect spaces. One suggestion made by a student would be to pre-format the string entered by the user to add the spaces if they were not present where expected, then use strtok or sscanf to parse individual elements of

the expression. I do like that idea a lot, but again, doing something like this is optional and not required at this point. Parsing user input is hard.

ANOTHER NOTE ON PARSING: Be sure to consult the "string library" (that is, standard library functions in string.h, stdlib.h, etc). While not rich by any means, there are basic functions to locate characters in strings, etc. You should not be duplicating anything that could be accomplished with standard library functions. The arguments and/or return values to some of the functions may be confusing. We have learned everything you will need to know to use these functions effectively, so study the documentation, and try them out.

Deliverables

This assignment is tentatively due by the end of Week 6. You will submit a "Pretty Printed" pdf to the Canvas assignment as well as demo your implementation during lab in Week 6.