Lab 4: Arrays and Structures

CSE/IT 113

CSE NMT

> I dont need to waste my time with a computer just because I am a computer scientist.
>
> — Edsger W. Dijkstra

> An algorithm must be seen to be believed.
>
> — Donald Knuth

Introduction

In this lab you will continue to write functions that operate on arrays and structures.

The concepts of the arrays of structures and enumerated type are also introduced this week.

If you don't understand any part of this lab it is important to get help with it. Next week's lab is a more complicated program using structures and enumerations.

The tutor schedule for this class is posted on Canvas. Take advantage of your resources!!

Reading

C Programming: A Modern Approach Sections 7.6, 8.1, 16.1 - 16.3

Array Problems

From the tarball, place functions for problems 1 - 8 in `vector.c` and `vector.h`. Use `vector_main.c` for your `main()` function.

Write a script called `vector.script` that uses **the same arrays as what are given in the example.**

All values should be hardcoded and the functions should run in the order that they are given in this lab. Output should be `identical` to what is shown below.

Function names are a design decision, however, function names should reflect what the function does. You will be graded on the name of your functions.

Please write your functions `in order` with the first one to be ran at the top of vector.c and downward from there. Your function prototypes should also match this order.

1. Write a function that multiplies each element of an array by a integer and stores the new value in the same array. Please print data to the screen like shown below:

   Before

```
a1[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 3}
```

   After array is multiplied array by 5

```
a1[] = {5, 10, 15, 20, 25, 30, 35, 40, 45, 15}
```

2. Write a function that adds an integer to each element of an array and stores the new value in the same array.

   Before

```
a2[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 3}
```

   After adding 3 to the array

```
a2[] = {4, 5, 6, 7, 8, 9, 10, 11, 12, 16}
```

3. Write a function that copies the contents of one array into a new array. Both arrays have to be the same size.

   Before

```
a3[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 3}
b3[] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
```

   After

```
a3[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 3}
b3[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 3}
```

4. Write a function that finds the sum of two arrays and writes the results to a third array. Since functions can't return arrays, you have to pass in three arrays to your function. All arrays must be the same size.

   Before

```
a4[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 3}
b4[] = {10, 10, 10, 10, 10, 10, 10, 10, 10, 10}
```

   After

```
a4[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 3}
b4[] = {10, 10, 10, 10, 10, 10, 10, 10, 10, 10}
c4[] = {11, 12, 13, 14, 15, 16, 17, 18, 19, 13}
```

5. Write a function that finds the product of two arrays and writes the results to a third array. Since functions can't return arrays, you have to pass in three arrays to your function. All arrays must be the same size.

   Before

```
a5[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 3}
b5[] = {10, 10, 10, 10, 10, 10, 10, 10, 10, 10}
```

   After

```
a5[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 3}
b5[] = {10, 10, 10, 10, 10, 10, 10, 10, 10, 10}
c5[] = {10, 20, 30, 40, 50, 60, 70, 60, 90, 30}
```

6. Write a function that finds the "inverted" product of two arrays and writes the results to a third array. Since functions can't return arrays, you have to pass in three arrays to your function. All arrays must be the same size. The "inverted" product is defined as

```
//a an array a[1..n]
//b an array b[1..n]
//c an array c[1..n]

c6[1] = a6[1] * b6[n]
c6[2] = a6[2] * b6[n - 1]
....
c6[n - 1] = a6[n - 1] * b6[2]
c6[n] = a6[n] * b6[1]
```

   Before

```
a6[] = {1, 2, 3, 4, 5}
b6[] = {6, 7, 8, 9, 10}
```

After

```
a6[] = {1, 2, 3, 4, 5}
b6[] = {6, 7, 8, 9, 10}
c6[] = {10, 18, 24, 28, 30}
```

7. Write a function that reverses the elements in a array. You will have to swap the array elements.

   Before

```
a7[] = {1, 2, 3, 4, 5}
```

   After

```
a7[] = {5, 4, 3, 2, 1}
```

8. Write a function that generates a random array of 10 numbers that are less than a given value (please use 50 as this value). Please output the array you generate. This should be array `a8[]`.

   The following code produces 3 random numbers:

```c
#include <stdio.h>
#include <time.h>
#include <stdlib.h>

int get_random_int(int n);

int main(void)
{
        int p, q, r;
        /* only need to call this once */
        srandom(time(NULL));

        p = get_random_int(16);
        q = get_random_int(1000);
        r = get_random_int(87);

        printf("%d, %d, %d\n", p, q, r);
        return 0;
}



/**
```

```
 * generates a random integer between [0,n)
 * @param n the upper bound
 * @return a random integer between [0, n)
 */
int get_random_int(int n)
{
        return random() % n;
}
```

The C Struct

In C, there are often places where you want to combine a whole group of variables in a particular context into a single type of variable. As the variables are related packaging them as a unit makes it much easier to keep track of your variables and how they are related. Another way to think about structures is that they are user defined data types.

For example, take points in one-, two-, and three-dimensional space. The variables you need are different for each type of point. The variables of a structure are called its *members*.

In C, you would write the three types of points as:

```
struct point1d_t
{
        double x;
};

struct point2d_t
{
        double x;
        double y;
};

struct point3d_t
{
        double x;
        double y;
        double z;
};
```

Now that we have various point structures, you can use them like this:

```
/* the statement is similar to a variable declaration,
but you are declaring a point1d_t structure and calling it point1d,
a point2d_t structure and calling it point2d, etc.
the declaration allocates space for one, two, or three doubles
(as seen in the structures above) and gives them a name */

/* point1d is of type struct point1d_t */
struct point1d_t point1d;      /* allocates space for 1 double*/

/* point2 is of type struct point2d_t */
struct point2d_t point2d;   /* allocates space for 2 doubles */

/* point3d is of type struct point3d_t */
struct point3d_t point3d;   /* allocates space for 3 doubles */
```

It is not the structure itself you want. A structure is just a way to encapsulate related variables. Rather you want the *members* of the structure. To access the members of a structure you use 'dot' notation. The name to the left of the dot is name of the declared structure and the name to the right of the dot is the member name.

For example, with the above declarations you could now make the assignments like this:

```
point1d.x = 10.0;

/* the ordered pair (5.0, 10.0) */
point2d.x = 5.0;
point2d.y = 10.0

/* the ordered triplet (1.0, 0.0, 1.0) */
point3d.x = 1.0;
point3d.y = 0.0
point3d.z = 1.0
```

Enumerated Types

Enumerated types are used when you have a finite set of possible values, and you want to use symbols to keep track of them. Enumerated types are used to create "symbolic constants." A common example is colors: there are a finite number of names for colors, and your code is easier to read when you use symbolic constants rather than numbers to keep track of these things. Enumerated types accomplish the same thing that multiple `#define` statements do, but in a more convenient manner.

```
/* RED = 0, BLUE = 1, YELLOW = 2 */
enum color
{
        RED, BLUE, YELLOW
};
```

After that code is inserted, you might use it like this if you had a `get_color()` function:

```
enum color my_color = get_color();

/*note: get_color() is a function where a user chooses a color*/

switch (my_color) {
case RED:
        /* do something with red */
        break;

case BLUE:
        /* do something with blue */
        break;
    ...
    ..
    .
}
```

Using an enumeration like that shown above gives RED the value of 0, BLUE 1, and YELLOW 2. An enumerated type default behavior is to assign the first element to 0, then increments each element by 1 after that (unless told to do otherwise).

Try these examples to see what they will print out for each value in the enumeration.

```
#include <stdio.h>

enum color
{
        RED = 1, BLUE, YELLOW, GREEN, ORANGE
};

int main()
{
        printf("RED is %d, BLUE is %d, YELLOW is %d, GREEN is %d,
                ORANGE is %d\n", RED, BLUE, YELLOW, GREEN, ORANGE);

        return 0;
}
```

What does this example print out? Notice instead of saying:

```
#define RED 1
#define BLUE 2
#define YELLOW 3
#define GREEN 4
#define ORANGE 5
```

we can just use the above enumeration. It does the same thing.

A cool thing about enumerations is that you can define one element, all elements, or no elements.

What would this example print out?

```
#include <stdio.h>

enum color
{
        RED = 1, BLUE = 7, YELLOW = 2, GREEN, ORANGE
};

int main()
{
        printf("RED is %d, BLUE is %d, YELLOW is %d, GREEN is %d,
                ORANGE is %d\n", RED, BLUE, YELLOW, GREEN, ORANGE);

        return 0;
}
```

What does this print out?

```
#include <stdio.h>

enum color
{
        RED = 1, BLUE = 7, YELLOW, GREEN, ORANGE
};

int main()
{
        printf("RED is %d, BLUE is %d, YELLOW is %d, GREEN is %d,
                ORANGE is %d\n", RED, BLUE, YELLOW, GREEN, ORANGE);

        return 0;
}
```

Questions

Answer the following questions.

1. Write two to three **paragraphs** stating why would you use a structure rather than an array to store data and vice versa. Give an example of each. Save this as `lab4_q1.txt`

2. Write a program that uses an enumeration:

   Use enumeration to declare the following types

   - RED = 18
   - ORANGE = 19
   - YELLOW = 5
   - GREEN = 6
   - BLUE = 7
   - INDIGO = 14
   - VIOLET = 15

   The catch to this assignment is that you can **only declare 3 elements** of the enumeration, and you cannot change the order of the elements. The enumeration looks like this:

```
1 enum color
2 {
3         RED, ORANGE, YELLOW, GREEN, BLUE, INDIGO, VIOLET
4 };
```

   Your job is to correctly declare only three of the types and let the compiler do the rest of the work.

   Save this file as `lab4_q2.c` and make a script called `lab4_q2.script`. Your program should print the color and its corresponding number.

3. Write a program that calculates the Euclidean distance and the Manhattan or taxicab distance between two two-dimensional points. Make sure you use a point structure to store the points (see above), functions to calculate the two distances, and ask the user for the point data. The Euclidean distance is calculated using the Pythagorean Theorem for (points (x_1, y_1) and (x_2, y_2)):

   $euclidean = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$

   And the Manhattan distance is calculated as:

   $manhattan = |x_1 - x_2| + |y_1 - y_2|$

   Save this as `lab4_q3.c` with a script `lab4_q3.script`.

4. Write a program that takes in a date from the user and stores it in `struct date_t`, that has 3 elements `unsigned short month, unsigned short day, unsigned short year`. `unsigned` means the integer is either positive or zero. Use the token `%hu` for `unsigned shorts` in your format strings.

   Then calculate the zodiac sign for that date. For example, if the date entered is 8/28/12, you should print out `August 28, Virgo`. For zodiac dates use the Tropical Zodiac columns found in the table at `http://en.wikipedia.org/wiki/Zodiac`

   Don't forgot about leap years, but how you handle them is a design decision. You can determine if a year is a leap year by using the following:

```
if (year % 4 == 0 && year % 100 != 0) || year % 400 == 0)
       /* leap year */
else
       /* not a leap year */
```

   You have to use the enumeration:

```
enum month
{
         JAN = 1, FEB, MAR, APR, MAY, JUNE,
         JUL, AUG, SEPT, OCT, NOV, DEC
};
```

   in your code.

   Add a menu system asking if the user wants to keep entering a date or if they want to exit from the program. See below for an example of a simple menu system.

   Save this as `lab4_q4.c` and make a script called `lab4_q4.script`

A simple menu system

```
#include <stdio.h>
#include <stdlib.h>

#define ENTER_INTEGER 1
#define QUIT 2

int main(void)
{

         int menu;
         int m;

         while (1) {
                  printf("1. Enter an integer\n");
```

```
15                  printf("2. Quit\n");
16                  printf("Enter selection: ");
17                  scanf("%d", &menu);
18
19                  switch(menu) {
20                  case ENTER_INTEGER:
21                          printf("\nEnter an integer: ");
22                          scanf("%d", &m);
23                          printf("You entered %d\n\n", m);
24                          break;
25                  case QUIT:
26                          printf("Goodbye...\n");
27                          exit(EXIT_SUCCESS);
28                          break;
29                  default:
30                          printf("You entered something
31                                  I don't understand\n\n");
32                          break;
33                  }
34          }
35          return 0;
36  }
```

Array of structures

Preliminaries

Just like you can have an array of integers, you can have an array of structures. Now each element of the array is a structure. For example, the following would allocate space for 10 **struct** person_t structures.

```
1  /*A structure to hold info about a person*/
2  struct person_t {
3          int ID;
4          int age;
5          char gender;
6  };
7
8  int main(void)
9  {
10         /* there are 10 people in this database */
11         struct person_t people[10];
12
13         return 0;
```

```
}
```

To access the 10 different structures you combine array syntax with structure dot notation.

```
/*A struct to hold info about a person*/
struct person_t {
        int ID;
        int age;
        char gender;
};

int main(void)
{
        struct person_t people[10];
        size_t size;

        /*size is the size of the array of structs/size of one struct*/
        size = sizeof(people)/ sizeof(struct person_t);

        /*Set values using dot notation. Use [] to show the array position*/
        people[0].ID = 900372847;
        people[0].age = 20;
        people[0].gender = 'F';

        people[1].ID = 900193847;
        people[1].age = 37;
        people[1].gender = 'M';

        /* etc */

        return 0;
}
```

You can pass array of structures to functions and just like other arrays you have to pass the size of the array to the function. Try out this sample code!

```
#include <stdio.h>

/*A struct to hold info about a person*/
struct person_t {
        int ID:
        int age;
        char gender;
};

void print_info(struct person_t p[], size_t size);
```

```
11
12  int main (void)
13  {
14          struct person_t people[2];
15          size_t size ;
16
17          /*size is the size of the array of structs/size of one ↙
                ↳ struct*/
18          size = sizeof(info)/ sizeof(struct person);
19
20          /*Set values using dot notation. Use [] to show
21            the array position*/
22          people[0].ID_num = 900372847;
23          people[0].age = 20;
24          people[0].gender = 'F';
25
26          people[1].ID_num = 900193847;
27          people[1].age = 37;
28          people[1].gender = 'M';
29
30          print_info (people, size);
31
32          return 0;
33  }
34
35  void print_info(struct person_t p[], size_t size)
36  {
37          int i;
38
39          /* the for loop will iterate through each element
40            in the array of structs*/
41          for(i = 0; i < size ; i++) {
42                  printf("\nIdentification number: %d\n", ↙
                        ↳ p[i].ID_num);
43                  printf("Age: %d\n", p[i].age);
44                  printf("Gender: %c\n", p[i].gender);
45          }
46  }
```

Array of Structures Problems

`array_stuct.c` contains the basis for writing a program to determine the min, max, and average of ages and heights using an array of structures.

1. Write the body of the function `fill_array()`. The prototype is given in `array_struct.h`.
2. Write a function that finds the min of the ages. Return the index.
3. Write a function that finds the min of the heights. Return the index.
4. Write a function that finds the max of the ages. Return the index.
5. Write a function that finds the max of the heights. Return the index.
6. Write a function to find the average age of the test subjects. Return the average age.
7. Write a function to find the average height of the test subjects. Return the average height.
8. Write a function to print an individual structure. Returns nothing.
9. Print each structure item that corresponds to the minimum age, the maximum age, the minimum height, and the maximum height.
10. Print the average age and the average height of the test subjects.
11. Run with the provided data in `array_struct.c` and save it as `array_struct.script`

Submission

Create a tarball named `cse113_firstname_lastname_lab4.tar.gz`.

In your tarball, please include

1. vector.c
2. vector.h
3. vector_main.c
4. vector.script
5. lab4_q1.txt
6. lab4_q2.c

7. lab4_q2.script
8. lab4_q3.c
9. lab4_q3.script
10. lab4_q4.c
11. lab4_q4.script
12. arrray_struct.c
13. array_struct.h
14. array_struct.script
15. README - note, no pseudo code is needed this week.

Submit your tarball to Canvas by the beginning of your next lab section.