Lab 5: Enumerations, Structures, & Arrays

CSE/IT 113L

NMT Department of Computer Science and Engineering

One essential object is to choose that arrangement which shall tend to reduce to a minimum the time necessary for completing the calculation.

— Ada Lovelace

The best way to predict the future is to invent it.

— Alan Kay

I've known people who have not mastered their tools who are good programmers, but not a tool master who remained a mediocre programmer.

— Kent Beck

Contents

1 Introduction

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

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

2 Overview

In order to complete this lab, read the material in this section and answer the questions and problems presented in Section 3,

2.1 Reading

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

2.2 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.

```c
/* 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:

```c
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 increment 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.

```c
#include <stdio.h>

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

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

        return 0;
}
```

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

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

we can just use the above enumeration. It is equivalent.

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

What would this example print out?

```c
#include <stdio.h>

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

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

        return 0;
}
```

What does this print out?

```c
#include <stdio.h>

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

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

        return 0;
}
```

2.3 A Review of C Structures

In computer science, there are often places where you want to combine a whole group of related variables into a single package. Treating the variables as a package makes it much easier to keep track of the variables and how they are related. In C, the package of variables is called a structure; in other languages they may be referred to as a record.

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:

```c
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:

```c
/* 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:

```c
point1d.x = 10.0;

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

```c
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
```

2.4 Array of Structures

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.

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

int main(void)
{
        /* there are 10 people in this database */
        /* how many bytes does the array occupy? */
        struct person_t people[10];

        return 0;
}
```

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

```c
/*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);

int main (void)
{
        struct person_t people[2];
        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';

        print_info (people, size);

        return 0;
}

```

```c
void print_info(struct person_t p[], size_t size)
{
        int i;

        /* the for loop will iterate through each element
          in the array of structs*/
        for(i = 0; i < size ; i++) {
                printf("\nIdentification number: %d\n", p[i].id);
                printf("Age: %d\n", p[i].age);
                printf("Gender: %c\n", p[i].gender);
        }
}
```

3 Lab Specifications

The following subsections describe what you will write and submit for this lab. **Check that you have met all of the requirements for this lab before you submit it for grading, as the lab will be counted as late if you submit or resubmit it after the due date.**

3.1 Restructuring Lab 3

Exercise 1 (lab5.c, lab5.script, array.c, array.h).
Copy the **lab3.c** file you created from Lab 3 to a file named **lab5.c**. Move over any files needed to compile and run it. If you did not do Lab 3 then you should write the file with respect to this lab; don't get hung up on Lab 3 specifications, as only the material covered in this lab will be graded (however you must meet all requirements for functionality from Lab 3)

Having to pass around 13 parameters to a function is a bit of a drag to say the least. And as you have probably noticed, all the summary information pertains to one array. If you ran the program with a different array you would have a different set of values. It is difficult to keep track of all the individual variables. To correct this situation, C has the ability to bundle related data into a cohesive unit. Such things are called **structures** in C. For instance, you could define the following structure to hold the size (number of elements in the array), the min, and max value of an array.

```c
struct summary_t {
        size_t size;
        int min;
        int max;
};
```

To use the structure, which is really just a user-defined data type. You would do something like this in **lab5.c**

```c
#include <stdio.h>

struct summary_t {
        size_t size; /* use %zu for printing */
        int min;
        int max;
};

/* can pass structures to functions */
void print_summary(struct summary_t summary);

int main(void)
{
        /* this declares summary to be a variable
           of type struct summary_t */
        struct summary_t summary;

        int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

        /* to access the members of the structure (or fields)
         * you use dot notation */

        summary.size = sizeof(a)/sizeof(int);
        summary.max = find_max(a, summary.size);
        summary.min = find_min(a, summary.size);
        print_summary(summary);

        return 0;
}

void print_summary(struct summary_t summary)
{
        printf("size = %zu\n", summary.size);
        printf("max = %d\n", summary.max);
        printf("min = %d\n", summary.min);
}
```

For the new version of the lab you are to:

1. *Create a structure that holds all 13 parameters for the summary. Put this file in* ***lab5.c****.*
2. *Create a* ***get_summary*** *function in* ***lab5.c*** *that uses this structure rather than individual parameters.*
3. *Move all function calls from* ***main()*** *to* ***get_summary()*** *except for* ***print_summary()****.*
4. *Call* ***get_summary()*** *from* ***main()****.*

5. *Modify the **print_summary()** function to also use this structure.*
6. *Call **print_summary()** from **main()**.*
7. *You will have to pass the summary structure, the array, and the size of the array to both **get_summary()** and **print_summary()**.*
8. *Capture the output of your program using the large array in a script file named **lab5.script**.*

3.1.1 Running lab5

Like in Lab 3, comment out the large array and test the changes with simple arrays. Once you are satisfied the changes work with small data sets, uncomment the large array and run your program. This is the data you will use when you run your script.

3.2 Array of Structures Problems

Exercise 2 (array_struct.c, array_struct.h, array_struct.script).
Answer the following problems referring to the information in Section 2.4.

`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 init_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`

3.3 Questions

Exercise 3 (lab5_q1.txt, lab5_q2.c, lab5_q3.c, lab5_q4.c, lab5_q2.script, lab5_q3.script, lab5_q4.script).
Answer the following questions referring to the information in Section 2.

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 `lab5_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:

   ```
   enum color
   {
           RED, ORANGE, YELLOW, GREEN, BLUE, INDIGO, VIOLET
   };
   ```

   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 `lab5_q2.c` and make a script called `lab5_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 1) use a point structure to store the points (see above), 2) use functions to calculate the two distances and 3) 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 `lab5_q3.c` with a script `lab5_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 `lab5_q4.c` and make a script called `lab5_q4.script`

A simple menu system

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

#define ENTER_INTEGER 1
#define QUIT 2
#define LEN 32

int main(void)
{
        char buf[LEN];
        int menu;
        int m;

        while (1) {
                printf("1. Enter an integer\n");
                printf("2. Quit\n");
                printf("Enter selection: ");
                fgets(buf, LEN, stdin);
                menu = atoi(buf);

                switch(menu) {
                case ENTER_INTEGER:
                        printf("\nEnter an integer: ");
                        fgets(buf, LEN, stdin);
                        m = atoi(buf);
                        printf("You entered %d\n\n", m);
                        break;
                case QUIT:
                        printf("Goodbye...\n");
                        exit(EXIT_SUCCESS);
                        break;
                default:
                        /* the \ is the line continuation character in C
                         * and is only needed to continue "string literals".
                         * You don't have to use it for other C constructs
                         * as the C compiler skips over white space.
                         *
                         * For example, to place function parameters on
                         * separate lines, you do the following:
                         *
                         * foo(a,
                         *     b,
                         *     c);
                         *
                         * NB: no line continuation */

                         printf("You entered something \
                                I don't understand\n\n");
                         break;
                }
        }
        return 0;
}
```

4 README

Exercise 4 (README).
Every lab you turn in must include a README file. The name of the file is README with no file extension.The README should include the following sections Purpose, Pseudo-Code (optional), and Conclusion:

- **Purpose:** describes what the program does (what problem it solves). Keep this brief.
- **Pseudo-code:** contains the pseudo-code you wrote for the lab. This depends on the lab. Some require pseudo-code; some do not.

- **Conclusion:**
  - What you learned. What new aspect of programming did you learn in this lab? This is the portion of the lab where you want to be analytical about what you learned.
  - Did Pair Programming help you in solving the problem and completing the prelab? Did you have problems working with your buddy?
  - Did you work with your buddy on the lab? What sections did you discuss? Did you and your buddy carry out a review session with each others code?
  - Did you encounter any problems? How did you fix those problems?
  - What improvements could you make?

The conclusion does not have to be lengthy, but it should be thorough.

5 Getting Help

If you need help with this Lab, please go to tutoring offered weekdays and weekends in Cramer. Please also ask questions in class to the TA's or instructor. These concepts are important for future labs and exams, so if you don't understand them now it is important to get help!
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 `https://www.cs.nmt.edu/tutoring`. Take advantage of your resources!!

Submitting

You should submit your code as a tarball file that contains all the exercise files for this lab. The submitted file should be named (**note the lowercase**)

`cse113_firstname_lastname_lab5.tar.gz`

Upload your `.tar.gz` file to Canvas.

List of Files to Submit

Exercises start on page 7.