HW1: Structures as Aggregates of Heterogeneous Data

In discussing programming languages we often differentiate between two different types of data "collections" depending upon whether the data (often variables) have a single value (called **scalars)** or whether they contain multiple values (called aggregates). You are already familiar with using arrays to support multiple values of the same type, such as arrays of integers, arrays of floats, arrays of characters, etc. And arrays are quite useful. But many times the data that you wish to maintain in an aggregate is not all of one type. We call this heterogeneous data. For example, think of the data that a university stores about each student. Some of that data is a string (name) stored in C as an array of characters. Other data about a student might be a grade point average, typically stored as a float, or mailing address which itself is heterogeneous data including integers such as street number, apartment number (possibly) and zip code. But then there is the street name, the county and state names (or maybe an abbreviation). And wait a minute; even the zip typically has a dash in it. Of course, we could represent all of the address as a string, but perhaps the university would like to look for all folks who live in the 1700 block of State Street. It would be easier to do this if the address, which is part of a student record, was stored as an aggregate inside another aggregate, the student record.

The goal of this tutorial and assignment is to both motivate the use of heterogeneous aggregates (called structures in C) and introduce you to some of the many ways they can be useful in programming. So, let's start with how one declares a structure in C. The code,

```
struct studentData{
      char lastName[30];
      char firstName[30];
      char idNum[9];
      int exam1;
      int exam2;
      int exam3;
      float avg;
      char grade;
} student; // makes a variable called student which is a struct
            // with fields as defined above.
```

Each of the "fields" as they are called has a declaration like you've seen before. To access a field of a struct (student in this case) we use the '.'. So, if we wished to say that our hypothetical student received a 75 on exam 1, we could just do an assignment, such as

```
student.exam1 = 75;
```

In fact, this might be a good time for you to copy all of the files (and directories) in the tutorial directory to one of your own. I'm sure you already know how to do this having been copying files all term long, but the "magic" command is:

```
cp –r ~dmk0080/public/1040/hwk/one/tutorial .
```

Make sure that you include both the –r (for recursive) and the '.' at the end to indicate where to place the copy of that directory. If you have any questions at all about how this command works, ask someone, a TA or grader, or a peer mentor if one is available.

One of the files that you just copied is a C file, called simpleExample.c, which uses the very structure definition listed above. Look at the code to see how structure fields (of "student") are accessed in both the **scanf** and **printf** statements. At this point in the term your probably understand why some fields in the scanf include an '&' and others don't. But if not, ask someone. Go ahead and run the simpleExample.c program, but typing

```
gcc simpleExample.c
./a.out < data
```

thereby reading the input from the file "data" which is also included in the directory.

Ok, next while still on (basically) the same program I'd like to make a change. Since one's grade typically depends upon more than just exams, but also homework, labs, and programs, we'd like to add fields for those. Well, we can do that easily, BUT, let's try something different (perhaps) to make our example a little more "real-world" (and harder, of course). Instead of just adding new fields, let's make structures for exams, programs, labs, and homework. We'll make each of these

structures pretty simple, just including places to put integer scores (3 for exams, 10 for labs, 4 for programs, and 6 for homeworks) and for each type of work have a float value for average. Take 5 minutes (no more because we've lots to do) to think about how you could define these structures, possibly working with a few other students. Then we'll look at two different solutions.

Welcome "back." There are two "obvious" ways we can solve this problem. The first is just to nest additional structure definitions into the definition of the "student" structure variable. The program nestedStructure.c shows this method. (Notice that I didn't actually read in any data for the programs, labs or homework, but I DID (need to) change way that I accessed the exam data. Before it was just:

`student.exam1` …

but now, since the exam scores are in a structure of their own I need an "extra" access, namely

`student.exams.exam1` …

The other way I could have organized the "nested" structures was to define several structures outside the structure defining the student variable. Look at outsideStructure.c for an example.

Notice that now, accessing the previously defined aggregates (structs) is a little different. To declare a field within the (structure) student variable, I need to use the keyword struct AND the name that immediately follows it in the outer declaration, for example

```
struct studentRecord {
      .
      .
      .
      struct examScores exams;
```

```
        .
        .
        .
}
```

Again, if you have questions you are encouraged to ask someone and keep asking until you get your concerns resolved.

To give you a bit of practice, choose either the nestedStructure.c or outsideStructure.c to "complete". Make a copy of that .c file and of data. Of course you'll call them something other than the original names of those files. In your new data file (**data2** or maybe **bigData**), add enough data for the programs, labs and homeworks for each of the 7 "students" listed. Add in any integer scores that seem appropriate. Then go to your copied "C" file and add to the scanf and printf statements to read and write the new fields we've just added. It should be relatively simple as you'll have a "template" to work with in regards to the limited fields of nestedStructure.c or outsideStructure.c, whichever you chose to copy. Compile and run your program as before (using the new larger data file you added to, of course.)

Now, let's move on. So far, we have been dealing with structures themselves, but more often in programming, you'll be working with structures "indirectly" using pointers to structures. Please look at the example program v4.h and v4.c. Again, we have much the same program as simpleExample.c but instead of using structure definitions we use pointers to structures. This will give us a distinct advantage in many cases, as I'll show in a much larger example that we'll be using for the next few labs, namely a "database" program. But more on that later.

First, look at v4.h. You've seen .h (or header) files before and it is quite common for larger C programs to have several .h files. The main thing we have used the .h file for here is to declare the student structure as a type (notice the keyword "**typedef**") so that instead of declaring variables in the program (v4.c) as

```
struct something someVariableName
```

We can make a new name for **struct something** and just call it **something**, thereby allowing declarations like

```
something someVariableName
```

Moving on to v4.c, notice that student is not defined to be a **studentRecord**, but rather to be a **pointer** to a **studentRecord.** This means that we have to do a bit more work (dynamically allocate the actual structure) but it has one GREAT advantage in "real programming" (whatever that might be.) That is because C does not allow aggregates to be passed as parameters to functions or returned from functions. Instead, function parameters and return values "representing" aggregates need to be passed as **POINTERS** to aggregates. ("Wait" I hear you cry, we pass arrays to functions all the time. Actually, you don't. It is just that arrays in C are defined to have a "value" that is the address of the start of the array, which is exactly what a pointer is --- an address. But C doesn't have such a rule for structures).

Ok, the only new wrinkles to the code in v4 are that, since we're using pointers to structures rather than structures themselves, we need to allocate space (using **malloc**) for the actual structure itself and the way we access the fields is different. Look closely at **v4.c** to notice both the dynamic allocation (**malloc**) statement and the change from using "**.**" to "**->**" to access the fields. Then compile and run v4.c by:

```
gcc v4.c
./a.out < data
```

Now, let me repeat myself. The main reason we use pointers a great deal with structures is that in bigger programs that use structures we usually want to pass those "structures" around among functions that read them, sometimes change them and often return them. And you can only pass and return aggregates (other than arrays) as pointers to the aggregate rather than the aggregate itself.

One last thing to talk about for this lesson is the often need to have arrays of pointers. To motivate this, I'm going to "point" you (sorry about that) to the database program that you now have as a directory in your code. This database application (and calling it a database is really exaggerating but it does do some of the things that a database program would do) maintains heterogeneous data for a number of fictitious students. The specific fields aren't terribly important except that like all "good" databases we wish to be able to locate all the records that match some characteristics, like "first-year students with grade point averages below

2.0" or "all students whose expected graduation date is before 2016". To answer these kinds of questions, we need to sort the records based upon many different fields of the structure. Compared to using multiple arrays of structures (one for each field of interest) and moving the structures around during sorting, it is MUCH more efficient (and less error prone) to have one copy of each student record and then maintain several arrays of pointers to the data (one array of pointers for expected graduation year, one for class in school, one for age, one for grade-point, etc.) and then "sort" the data by moving the pointers around in the arrays to represent ordering among records based upon that field. In the database directory that you copied, look at the file **db.c** and look for the function **buildDataBase()** in **db.c**. In that function the variable **theDataBase,** a global variable by the way (shame on me!), is used to create the "entire" database that is eight arrays of pointers to structures. Look specifically at lines 63 (a comment) through 111 to see how the database is allocated (**malloc** of course), how individual **DBRecord** structures are allocated (again **malloc**), how the input data is read into the record just allocated (**scanf**) and (in lines 109 and 110) the 8 arrays of pointers are initially all set to point to the **num DBRecords** of the database. Later on, sorting occurs so that each of the 8 arrays of pointers to **DBRecord structs** lists the pointers in order of the **DBRecord** field associated with that array.

Declaring arrays of pointers to structures is not hard. If our structure definition is called "student", then

```
student *classOrder[36000];
student *gpaOrder[36000];
student *expectedGradYear[36000];
```

declares three different arrays each of which can be ordered based upon a different field of the student records in the database.

no need to submit your tutorial files

I hope that at this point, you're ready for your homework. Good Luck!