CS102: Structures

A **structure** is a defined data type that allows a programmer to store different pieces of information, possibly of different types, in a single variable. For example, let's say you want to write a program that allows a user to enter data points on a Cartesian plane. Each point will have an x coordinate and a y coordinate. It would be annoying to store all the x coordinates and y coordinates in separate variables. We want to define a data type used for storing points such that each single variable will store both the x coordinate and the y coordinate of a point. You can set up such a data type using structures as follows:

```
struct point
{
  int x;
  int y;
};
```

This defines a new data type called "point". New variables can be declared with this type. Let's say you want to declare two variables, "point1" and "point2", of type "point". You would use:

```
struct point point1;
struct point point2;
```

or, you could declare both variables with a single declaration as follows:

```
struct point point1, point2;
```

The individual elements of a structure are often referred to as **fields** or **members** of the structure. You access a specific field of a structure variable by specifying the name of the variable followed by a period (known as the **member operator**) followed by the name of the field. For example, the following are all valid statements if "point1" and "point2" are structure variables of type "point", and "point" is a structure as defined above:

```
point1.x = 5;
point1.x = point2.y * 7;
printf("The coordinates of point1 are (%d, %d).\n",
point1.x, point1.y);
scanf("%d", &point2.y);
```

If the definition of a structure type is defined at the top of a function, it is local to the function, and only that function can declare structure variables of that type. If the structure type is defined outside all functions at the top of your program, it is a global type, and all functions can use structure variables of that type.

One concept that is often used in conjunction with structures is **type definitions**. Using a type definition really means that you are giving it a second name to an existing type. The existing type can be either a standard type that C provides (e.g., "int", "float", or "char")

or a type that you have already defined (e.g., a structure). It has become standard practice to name all type definitions using all capital letters, but this is not required. For instance, let's say you don't like the abbreviation "int" for integer, and would prefer to use "INTEGER". You can create the following type definition using the "typedef" construct:

```
typedef int INTEGER;
```

Now, in your program, you can declare integers as follows:

```
INTEGER x, y, z;
```

This is equivalent to:

```
int x, y, z;
```

It is more common to use type definitions when defining structures because it avoids the need of repeating the "struct" keyword for every variable declaration. For instance, after defining the "point" structure as above, you can create a type definition for type "POINT" as follows:

```
typedef struct point POINT;
```

Then you can declare "point1" and "point2" as follows:

```
POINT point1, point2;
```

The type FILE is actually a structure containing information about a particular file. It is defined in "stdio.h".

It is also possible to combine the definition of a struct with the typedef that gives it a name. For instance:

```
typedef struct point
{
     int x;
     int y;
} POINT;
```

When you do this, if you know that all variables will be declared using the type definition, you may actually leave out the first structure name. For instance, it is valid to say:

```
typedef struct
{
     int x;
     int y;
} POINT;
```

It is possible to assign the value of one struct variable to another structure variable of the same type. For example, let's say that "point1" and "point2" are both structures of type "point", "point1" has already had its fields filled in, and you want to assign the value of "point1" to "point2". It is valid to say:

```
point2 = point1;
```

This is the same as assigning the values of each individual field of "point1" to the fields of "point2". If you do this with structures containing arrays, each element of each array is copied (it is technically a byte-by-byte copy taking place), but if you do this with structures containing pointers, only the pointers are copied (i.e., the original and the copy point to the same memory).

It is also possible for one field of a struct to be another struct. For example, let's say you have already defined the "point" struct as above (without using "typedef"), and now want to define a structure to contain the top left and bottom right points of a rectangle. You could do this as follows:

```
struct rectangle
{
     struct point top_left;
     struct point bottom_right;
};
```

You could then declare "rectangle" variables as follows:

```
struct rectangle rectangle1, rectangle2;
```

Assuming "point1" and "point2" have still been declared as above, some valid statements would now be:

```
rectangle1.top_left.x = point1.x;
point2.y = rectangle2.bottom_right.y;
rectangle1.top_left = point1;
```

You can initialize the fields of a structure variable one at a time after the variable has been declared, or you can initialize all the fields at once when you declare the structure variable. Similar to when you initialize an array variable with its declaration, you specify the values of all the fields, in order, within curly braces. Nested structures must have nested sets of curly braces. For instances, the following are valid declarations with initializations assuming the structures defined above:

```
struct point point1 = {2, 5};
struct rectangle rectangle1 = {{2, 5}, {7, 1}};
```

If one of the fields happens to be a string, you can indicate its initial value by specifying the string in quotes.

It is also possible to use arrays within structures, or to have arrays of structures. Rather than give individual examples of this, we are going to look at entire small application that relies on many of the topics we have covered. This defines a structure called "student". Associated with each student is a first and last name, scores for four homework assignments, a score on a test, and a final weighted average. This application will read student information from a specified text file, compute the final average for each student, and output the information in sorted order based on names to another specified text file. A few new concepts are also introduced in this program.

```
/*
 * This program reads student information from a specified text file.
 * Each student has an associated first and last name, four homework
 * scores, and one test score. The program computes the weighted
 * average score for each student, and sorts the data based on names.
 * The updated information is written to a specified text file.
 * Written by: Carl Sable
 */

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

#define NUM_HW 4

struct student
{
  char lastName[30];  /* last name */
  char firstName[30]; /* first name */
  int homeworks[NUM_HW];  /* scores on homeworks */
  int test;  /* score on test */
  float final;  /* final average */
};

typedef struct student STUDENT;

STUDENT *input_students(int *);
void compute_averages(STUDENT [], int);
void sort_students(STUDENT [], int);
void swap_students(STUDENT *, STUDENT *);
void display_students(STUDENT [], int);

int main(void)
{
  STUDENT *pStudents;
  int numStudents;

  pStudents = input_students(&numStudents);
  compute_averages(pStudents, numStudents);
  sort_students(pStudents, numStudents);
  display_students(pStudents, numStudents);
```

```
  free(pStudents);

  return 0;
}

/* Read all student data from a specified text file */
STUDENT *input_students(int *pNumStudents)
{
  int x, y;
  char filenameInput[30];
  FILE *fpInput;
  STUDENT *pStudents;

  printf("Enter name of input file: ");
  scanf("%s", filenameInput);

  fpInput = fopen(filenameInput, "r");
  if (fpInput == NULL)
  {
    fprintf(stderr, "Error: Could not open input file!\n");
    exit(1);
  }

  /* First line of input file contains number of students */
  fscanf(fpInput, "%d", pNumStudents);

  /* Allocate necessary memory to store student info */
  pStudents = (STUDENT *)malloc(sizeof(STUDENT) * *pNumStudents);
  if (pStudents == NULL)
  {
    fprintf(stderr, "Error: Out of memory!\n");
    exit(1);
  }

  /* Read information about students */
  for (x = 0; x < *pNumStudents; x++)
  {
    /* Input file lists names as first name followed by last name */
    fscanf(fpInput, "%s %s",
         pStudents[x].firstName,
         pStudents[x].lastName);
    for (y = 0; y < NUM_HW; y++) /* Read homework scores */
      fscanf(fpInput, "%d", &pStudents[x].homeworks[y]);
    fscanf(fpInput, "%d", &pStudents[x].test); /* Read test score */
  }

  fclose(fpInput);

  return pStudents;
}

/*
 * Computes final average for each student.
 * Assumes homework is worth 60% (in total) and test is worth 40%.
 */
void compute_averages(STUDENT arrS[], int numStudents)
{
```

```
  int x, y;
  int h_tot;

  for (x = 0; x < numStudents; x++)
  {
    h_tot = 0;
    for (y = 0; y < NUM_HW; y++)
      h_tot = h_tot + arrS[x].homeworks[y];

    arrS[x].final = (0.6 * h_tot / NUM_HW + 0.4 * arrS[x].test);
  }

  return;
}

/* Use selection sort to sort students according to names. */
void sort_students(STUDENT arrS[], int numStudents)
{
  int index1, index2, indexS;

  /* Outder loop loops through all slots */
  for (index1 = 0; index1 < numStudents - 1; index1++)
  {
    indexS = index1;
    /* Inner loop finds the smallest element starting at index1 */
    for (index2 = index1 + 1; index2 < numStudents; index2++)
      if ((strcmp(arrS[indexS].lastName, arrS[index2].lastName) > 0) ||
        ((strcmp(arrS[indexS].lastName, arrS[index2].lastName) == 0) &&
         (strcmp(arrS[indexS].firstName, arrS[index2].firstName) > 0)))
      indexS = index2;

    /* Swap the smallest element with the one at index1 if necessary */
    if (indexS != index1)
      swap_students(&arrS[index1], &arrS[indexS]);
  }

  return;
}

/* Exchanges the values of two students. */
void swap_students(STUDENT *pstud1, STUDENT *pstud2)
{
  STUDENT tmp;

  tmp = *pstud1;
  *pstud1 = *pstud2;
  *pstud2 = tmp;

  return;
}

/* Displays name and final average of each student. */
void display_students(STUDENT arrS[], int numStudents)
{
  int x;
  char filenameOutput[30];
  FILE *fpOutput;
```

```
  STUDENT *pStudents;

  printf("Enter name of output file: ");
  scanf("%s", filenameOutput);

  fpOutput = fopen(filenameOutput, "w");
  if (fpOutput == NULL)
  {
    fprintf(stderr, "Error: Could not open output file!\n");
    exit(1);
  }

  /* Loop through all students */
  for (x = 0; x < numStudents; x++)
    fprintf(fpOutput, "%s, %s %.1f\n",
          arrS[x].lastName, /* In output, dispay last name first */
          arrS[x].firstName,
          arrS[x].final);

  fclose(fpOutput);

  return;
}
```

The definition of the "student" structure contains five fields. Two are arrays of characters used to store strings representing a student's first and last name. (We are assuming that each first and last name will be at most 29 characters long, leaving room for the null character.) Another field is an array of integers storing the scores of the student on four homework assignments. The next field stores the score of the student on a test. The final field is not present in the input file but computed by the program; it represents the student's final average for the course. The "typedef" line defines "STUDENT" to be the same type as "struct student".

In "main", we declare a pointer to STUDENT that will soon point to dynamically allocated memory; you can call this a dynamically allocated array. Each slot in this dynamically allocated array represents one student. There is also a declaration of an integer ("numStudents") that will store the number of students, but this information comes from a text file that is read in another function. Our "main" calls four separate routines to read input from a specified text file, to compute averages for students, to sort the student records (structures) according to names, and to display information about the students to a specified output file.

The routine "input_students" is responsible for reading data from an input file specified by the user. After prompting the user for the name of the input text file, the file is opened, and it is assumed to have a very specific format. The first row of the input file contains the number of students whose information follows. This value is assigned to the variable pointed to by "pNumStudents", a parameter. Looking back at "main", we see that this parameter points to (i.e., stores the address of) the "numStudents" local variable in "main". The remainder of the input file lists student information, one student per line. Each remaining line lists a student's first name, last name, four homework scores, and one test score (separated by whitespace). Every line in the file ends with a newline character.

After the first line of the input file is read, the function dynamically allocates enough memory to store all of the student information. Note that the "sizeof" operator can be applied to a structure type to determine the number of bytes necessary to store each structure. We multiply this by the number of students (pointed to by pNumStudents).

We loop from 0 through one less than the number of students and read the information for each student. Note that the "scanf" statement used to read names does not use '&' since it is passing a string (an array of characters, but remember that the name of an array is a pointer). The other two "scanf" statements do use '&' since each is being used to read the value of an integer. The precedence of '&', although high, is less than the precedence of '[]' or '.' so in each case, the expressions to the right of the '&' sign in this function is evaluated first, and then the address of the integer to be filled is passed to "scanf".

The routine "compute_averages" is passed the array of students (really a pointer to the dynamically allocate array). The function loops through the "STUDENT" records, and for each "STUDENT" structure, it computes the student's final average based on his or her other scores and fills in the final field of the structure.

The routine "sort_students" uses **selection sort** to sort the student records by name. Note that **sorting** is an extremely important topic in computer science. Selection sort is not an efficient sorting algorithm by any means. I have used it here only because it is conceptually simple to understand and simple to code. This is a quadratic routine (meaning that it takes time proportional to the square of the number of items to sort). This is OK if you or sorting hundreds, or even thousands, of items; but if you are sorting hundreds of thousands or millions of items, it would be much too slow. Efficient sorting techniques include quicksort and merge sort. If you take a class on data structures and algorithms, you will learn about several efficient sorting techniques in great detail, including which sorts are the best to apply in various situations. In any case, selection sort relies on nested loops, as you can see. The outer loop loops through the positions of the array (from 0 through one less than the number of students). For each position, the inner loop finds the index of the smallest element, starting at the index determined by the outer loop. What we mean by "smallest" hear is the structure with the name that comes first according to ASCII order. The last names are compared first, and if they are equal (in which case "strcmp" returns 0), the first names are then compared to break ties. In the past, we have only used strcmp to check if two strings were equal; here, we use it to see if one is greater than the other. The "if" statement checks to see if the name at the position indicated by "indexS" is greater than the name at the position indicated by "index2". (The function "strcmp" will return some positive number if and only if the first string argument is greater than the second string argument.) If so, we have found the smallest element so far, starting at position "index1", up through "index2", and in this case we update "indexS" to remember this index. After the inner loop ends, "indexS" stores the index of the structure with the smallest name, and this is swapped with the structure at "index1" (unless it was already in the right place, in which case the two indexes are the same). In this way, selection sort determines the smallest item in the list fist, then the second smallest, then the third smallest, etc.

The "swap_students" routine that "sort_students" relies on is very similar to the routine we have seen when we covered pointers to swap two integers. Remember that swapping two values (or in this case, structures) requires three lines of code, so it is often considered more readable to make this a separate function. By passing pointers to the two "STUDENT" records, we can permanently change their values. (Since "swap_students" is only used in one location, an argument can be made to move this code into "sort_students", avoiding the extra function calls.)

Finally, "display_students" loops through the "STUDENT" records (already sorted) and prints the name and final average for each student into a specified text file. Unlike the input file, here we display last names first, first names last.

Although we have used pointers to structures in this program (in "swap_students"), we did not access any specific fields of the structures that the pointers point to. We only manipulated the structures as a whole. Let's say that "pStud" is a pointer to a variable of type "STUDENT" as defined in the above program. One way to access its fields is by first using the indirection operator to get to the structure itself, than accessing its fields in a typical fashion. For instance, the following are valid statements:

```
(*pStud).homeworks[1] = 87;
scanf("%s", (*pStud).lastName);
```

Note the parentheses around the instances of "*pStud". These parentheses are necessary because the precedence of '.' is higher than the precedence of '*'. We must do the indirection of the pointer first to get to the structure, and then access the structure's field.

To avoid the confusion and sloppiness of this type of code, another operator, known as the **selection operator**, exists to be used along with pointers to structures. The selection operator dereferences a pointer to a structure and accesses a specified field of the structure. The selection operator has the same precedence as the member operator. To use the selection operator, you first specify the name of the pointer, then a '-' followed by a '>' followed by the name of the field. So the selection operator is specified as "->" which looks sort of like a right arrow. For example, the two lines above could be rewritten as the equivalent:

```
pStud->homeworks[1] = 87;
scanf("%s", pStud->lastName);
```

This notation is much more common than the first notation.

Without structures, each piece of information about any object or entity would need to be stored in a separate variable. Apart from the extra bookkeeping that this would entail just to keep track of all the variables, certain procedures, such as the sort in the above program, would become quite annoying. For example, if each field for all students were kept in a separate array, and we wanted to sort all student information according to names, the sort would have to be based on the array of names, but slots of all the other

arrays would need to be swapped also so that information on individual students remains coordinated. It would be very easy to make errors that lead to information about different students getting mixed up.

For those of you who are familiar with object oriented programming, structures also represent a step toward **classes**. There are no member functions (a.k.a. methods), but there is encapsulation in the sense that all of the data associated with a concept is stored together in one place. (You do not need to worry about this terminology if you are new to programming.)

Finally, structures are a key component to many important data structures, including **linked lists**. In a linked list, each structure contains one field that can point to another structure of the same type. Consider an application in which you not only do not know how much data there will be at compiler time, but there is also no indication at any single time how much more data there will be in the future. (This is unlike the case with the program we just looked at, in which the start of the input file indicates the number of student records to follow.) For instance, consider a word processor, a spreadsheet, or a database; a user using any of these applications can keep entering data as long as they want to. It would not be reasonable to ask the user at the start how big his or her document or database is going to be ahead of time! Therefore, the program needs to occasionally allocate additional dynamic memory to store data (whenever the memory previously allocated has been used up). A linked list allows each node (a structure in a linked list) to store a piece of data and also contain a pointer to the next node (with more data). Every time more data is entered, a new node can be allocated and inserted into the appropriate location in the linked list. (That is, the previous node can point to the newly allocated node, and this node can point to the next node.) You do not need to worry about the details of this now. This is another topic that you will learn about in much detail if you take a course in data structures and algorithms.