Dynamic Memory

Lab 7

The labs, for this course, are designed to be completed on your own at home or in the 3rd floor Trottier labs. These labs are not graded. You do not hand in these labs. If you prefer to work on a lab in a supervised setting, check the TA Information schedule for the Lab TA period(s). You will find this schedule in our MyCourses page under Content/Course Information. The supervised labs are not teaching environments. The Lab TA will simply be present to answer questions and provide support.

This lab is about **dynamic memory** and **C structures**. Dynamic memory is a very important concept in computer science. It overcomes the limitations of fixed memory. For example, an array, `int a[10]`and a variable, `int x` are fixed structures. A variable is a structure that can store a single value. An array is a structure that can store multiple values. Both arrays and variables cannot be resized (hence we call them fixed). A variable always stores only one value it cannot be made to store more than one value. An array, after compiling, has a fixed size and cannot be resized. Dynamic memory provides greater flexibility. It allows the developer to construct new variables and arrays while the program is running. If the program suddenly needs another array, no problem, it can be created. The *heap* is part of a program's run-time environment. Special library functions, like `malloc()`, create structures in the heap. A developer can create any custom structure or primitive in the heap while the program is running. The developer does not need to know in advance the number of cells or variables the program will need. It can be created at run-time.

Variables and arrays limit what a developer can represent in an algorithm. Variables store only a single value of a single type. Arrays store many values but also using a single type. Structures address this single type value representation problem. Structures let the developer invent their own unique complex types. This flexibility opens possibilities for the developer to represent and manage complex data.

C structures can be identified by the reserved words **struct** and **union**.

In this lab we will define a complex data problem using a struct and then build a fixed and a dynamic version of that structure definition.

Part one: A C structure experiment

A data-structure is formed by nesting variables, arrays, and structures within a new structure-definition.

For example, we can programmatically define what a student is:

```
struct STUDENT_RECORD {
   char name[50];
   int age;
   float gpa;
};
```

The above structure, by convention, has an all caps type name to distinguish it as a structure. Its type name is `STUDENT_RECORD`. The type name tells us what the structure is about. The word "student" tells us the structure's purpose. The word "record" tells us that it is composed from multiple smaller pieces. It has three smaller pieces: a character array, an integer, and a floating-point number. The structure ends with a close curly bracket and a semi colon. This structure is only a definition. It is not a variable and it has not been instantiated. It does not yet exist physically.

To create physical instances of a `STRUDENT_RECORD` we do any of the following:

```
a) struct STUDENT_RECORD student;
b) struct STUDENT_RECORD students[10];
c) struct STUDENT_RECORD *s =
   (struct STUDENT_RECORD *) malloc(sizeof(struct STUDENT_RECORD));
```

In the above example, (a) builds a variable called `student` that contains the three smaller pieces defined in `STUDENT_RECORD`. Example (b) creates an array of 10 cells, where each cell is a single `STUDENT_RECORD`. Example (c) is a pointer called `s`, that receives an instance of `STUDENT_RECORD` created by the `malloc()` function.

Try to do the following tasks:

1. Create a header file called `student.h` and put in that file the `STUDENT_RECORD` struct definition, from above.
2. Create a C file called `student.c`, and include `student.h`.
3. Write a `main()` function in `student.c` that creates the three instances of `STUDENT_RECORD` described above.
4. Assign "Mary", 18, 3.8 to the variable `student`. To all the cells in the array `students`. To the malloced structure `s`. Check our class notes on how you can assign these values, but as a reminder: `student.age = 18` will work for the fixed variable `student`, `student[0].age = 18` will work for the fixed array `students`, and `s->age = 18` will work for the dynamic structure created by `malloc()`. Do not read input from the user to make these assignments.

5. Then print all the values to the screen to confirm that you were successful.
6. Make sure to make this code compile and run on mimi and feel free to experiment with it.

Part two: A dynamic programming experiment

In lecture Week 7 you were introduced to linked lists and saw the functions: `printNodes()` and `addNodes()`. In this experiment we will create a linked list of students using the following `main()` function:

```
int main() {
  struct STUDENT_RECORD *head = NULL;
  char selection='Y', aName[50];
  int anAge;
  float aGPA;

  while (toupper(selection) == 'Y') {
    // prompt the user for aName[], anAge, and aGPA

    // call addNodes() but modify the parameters to pass
    // aName[], anAge, and aGPA. Fix the function to
    // populate the node correctly.  The node, in our case
    // is STUDENT_RECORD and not NODE.

    printf("Continue? (Y/N): ");
    selection = getc(stdin);
  }

  printNodes(head);
}
```

For this experiment, try to do the following tasks:

1. Reuse the files from experiment 1. If you want, make a copy of the old files.
2. Modify the header file called `student.h`, change the `STUDENT_RECORD` struct definition by adding the field `struct STUDENT_RECORD *next`. This will effectively convert it into a linked list node structure (data fields and pointer to next).
3. Modify the C file called `student.c`, by replacing `main()` with the main function we have above.
4. Copy the two functions from the class notes, `printNodes()` and `addNodes()` into the file `student.c` above `main()`. You will need to modify these two functions to work with `STUDENT_RECORD` and not `NODE`.
5. The main method has comments placed within the loop where code should exist. Finish the main function.
6. Make sure to make this code compile and run on mimi and feel free to experiment with it. If you are successful, then all the students you entered will be printed out.