Prelab 4

For this week the goal is to get more familiar with the use of structs. We'll assume the following Employee data type:

```
typedef struct
{
     int empID, jobType; float
     salary;
} Employee;
```

Now implement the following function:

```
Employee * readEmployeeArray(FILE *fp)
```

which returns an array of Employee structs similarly to what you've done before where you can assume an integer at the beginning telling you how many records follow, which tells you how big the array needs to be. (Will you need to provide getSize and freeArray functions? Seems like this must be a "Smart Array") Each record will consist of two integers and a float corresponding to the members of the Employee struct.

After readEmployeeArray() is called, the user will have an array of Employee records. The following function is useful if the user wants to retrieve the record of a particular employee from the array based on her/his employee ID:

```
Employee * getEmployeeByID(Employee *, int empID)
```

where the first parameter is the array of employees and the second parameter is the employee ID. The function returns a pointer to the record (i.e., address of an array element struct) for the employee with that ID. Note that this just requires a loop to find the employee with the specified empID. What do you do if there is no employee with that ID?

Now you need to implement *get* and *set* (also known as *getter* and *setter*) interface functions:

```
int setEmpSalary(Employee *, int empID, float salary)
int getEmpSalary(Employee *, int empID, float *salary)
int setEmpJobType(Employee *, int empID, int job)
int getEmpJobType(Employee *, int empID, int *job)
```

where the first parameter is the employee array, the second is the ID of an employee, and the return value is an error code (1 for error, 0 for success). The third parameter for the getter functions is a reference for the requested value. Why not make those return values? Well, we could do that, but then the user would have to pass an error code variable by reference to the set functions and there would be no use made of a return value.

Instead, we could choose the following prototypes:

```
void setEmpSalary(Employee *, int empID, float salary, int *ec)
float getEmpSalary(Employee *, int empID, int *ec)
void setEmpJobType(Employee *, int empID, int job, int *ec)
int getEmpJobType(Employee *, int empID, int *ec)
```

Is this better? Maybe. Both are reasonable options, so you decide which set of prototypes you prefer for your interface functions. You can use one of the example options above, or you can do something completely different. Just make sure that the conventions you use are memorable and are applied consistently – *and are documented!*

It's important to remember why interface functions are needed: They ensure that users never have to directly access any member of the Employee struct. Suppose we did allow users to access the members directly, then what would happen to their programs if we later change the member "`empID`" to "`employeeID`" ?

Note that you've already implemented functions that are relevant to this prelab. The reason for implementing useful functions is to allow you (or a user) to reuse them to make subsequent programs easier and less error-prone to implement. This "building block" approach to programming is necessary to minimize the complexity of creating large sophisticated pieces of software. (HINT: If you use a loop in any of your *get* and *set* functions then you've missed the "Big Picture" of how to think about functions as building blocks.) It would make sense to include a function getNumberOfEmployees, but if your implementation has more than one line of code then you've missed the big picture of how to use building-block functions.