Assignment 2

Version 1.00 (last update: Sept. 22, 0:00)
Changes highlighted in yellow
Due date: Thu, Oct 1, 11:59 PM

Summary and Purpose

For this assignment, you will be writing a collection of C functions that operate like arrays. You will use these to understand the underlying properties of arrays, their strengths and weaknesses.

Deliverables

You will be submitting:

1) A file called **`array.h`** that contains your function prototypes (see below).
2) A file called **`array.c`** that contains your function definitions.
3) A **`makefile`** that contains the instructions to compile your code.

You will submit all of your work via git to the School's gitlab server. (As per instructions in the labs.)

This is an individual assignment. Any evidence of code sharing will be investigated and, if appropriate adjudicated using the University's Academic Integrity rules.

Structures for your assignment

You will be working with variables having the following structure which you must declare in your header file.

```
struct Array
{
  unsigned int width;
  unsigned int nel;
  unsigned int capacity;
  void *data;
};
```

This structure represents an array data structure (our first, and one of the simplest data structures used in this course). The elements of the Array structure are as follows: **`width`** represents the size in bytes of each element in the array, **`nel`** represents the number of elements currently in the array, **`capacity`** represents the total number of elements that can be stored in the array, and **`data`** is a pointer to the contents of the array.

Additionally, you will be using the following structure to measure the performance of your code and count the number of memory read, memory write, malloc and free operations.

```
struct Performance
{
  unsigned int reads;
  unsigned int writes;
  unsigned int mallocs;
  unsigned int frees;
};
```

Basic function prototypes and descriptions for your assignment

```
struct Performance *newPerformance();
```

This function will allocate sufficient memory for a **`Performance`** structure, set **`reads`**, **`writes,`** **`mallocs,`** and **`frees`** to zero (yes, I realize there is technically one malloc in this function) and return the address of the structure. Your function should print an error message to the standard error stream and **`exit`** if the **`malloc`** function fails.

```
struct Array *newArray( struct Performance *performance, unsigned int width,
unsigned int capacity );
```

This function will allocate sufficient memory for an **`Array`** structure, set the **`width`** and **`capacity`** attributes of the structure to the values provided, set the **`nel`** attribute to zero, and allocate **`width*capacity`** bytes of memory storing the address of that memory in **`data`**. It will increment the **`performance->mallocs`** value by one (yes, I know there are technically two mallocs performed in this function). Finally, it will return a pointer to the allocated **`Array`** structure (not to **`data`**). You may assume that width and capacity are non-negative. Your function should print an error message to the standard error stream and **`exit`** if the **`malloc`** function fails.

```
void readItem( struct Performance *performance, struct Array *array, unsigned
int index, void *dest );
```

If **`index`** is greater than or equal to **`array->nel`** this function should print an error message to the standard error stream and **`exit.`** Otherwise, this function will copy **`array->width`** bytes from the memory address **`array->data`** offset by the **`index`** (multiplied by **`array->width`**) to the memory address given by **`dest`**. In addition, it should add one to **`performance->reads`**.

```
void writeItem( struct Performance *performance, struct Array *array,
unsigned int index, void *src );
```

If **`index`** exceeds **`array->nel`** or exceeds or equals **`array->capacity`** this function should print an error message to the standard error stream and **`exit.`** Otherwise, this function will copy **`array->width`** bytes from the memory address given by **`src`** to the memory address **`array->data`** offset by the **`index`** (multiplied by **`array->width`**). If index exactly equals **`array->nel`**, `array->nel` should be incremented by one. In addition, it should add one to `performance->writes`.

```
void contract( struct Performance *performance, struct Array *array );
```

If `array->nel==0` this function should print an error message to the standard error stream and `exit.` Otherwise, it should decrement `array->nel` by one.

```
void freeArray( struct Performance *performance, struct Array *array );
```

This function will `free` both `array->data` and the `array` structure itself. And it will increment `performance->frees` by one (yes, one).

Derived function prototypes and descriptions for your assignment

The following functions must be implemented by calling the basic functions above (not by interacting with `data` of the `Array` structure directly). Error handling will be done by the basic functions.

```
void appendItem( struct Performance *performance, struct Array *array, void
*src );
```

This function will add an element to the end of the array (i.e. at position `array->nel`). It will do this by calling the `writeItem` function (above).

```
void insertItem( struct Performance *performance, struct Array *array,
unsigned int index, void *src );
```

This function will use `readItem` and `writeItem` calls to move all the elements in the `array` at the position given by `index` and higher, one position further back and then write the given data at `index` in the array.

```
void prependItem( struct Performance *performance, struct Array *array, void
*src );
```

This function will use `insertItem` to insert data at position 0.

```
void deleteItem( struct Performance *performance, struct Array *array,
unsigned int index );
```

This function will use `readItem` and `writeItem` calls to move all the elements in the `array` at the position given by `index+1` and higher, one position forward, and then use the `contract` function to remove the duplicate last entry.

The Last 20%

The above, constitutes 80% of the assignment. If you complete it, you can get a grade up to 80% (Good). The rest of the assignment is more challenging and will allow you to get a grade of 80-90% (Excellent) or 90-100% (Outstanding). Make sure you complete the first part well, before proceeding to the following additional part.

Write the following functions:

```
int findItem( struct Performance *performance, struct Array *array, int
(*compar)(const void *, const void *), void *target );
```

This function will retrieve elements from **`array`** using **`readItem`** (above) starting with the first element in the array and proceeding incrementally. For each element it will apply the **`compar`** function to **`target`** and the retrieved element. If the **`compar`** function returns 0 (indicating a match), this function should return the index of the matching element. If they compar function returns a non-zero value (indicating a mismatch) it should proceed with the next element. If they function processes the entire array without finding a match, it should return a value of -1.

```
int searchItem( struct Performance *performance, struct Array *array, int
(*compar)(const void *, const void *), void *target );
```

This function will retrieve elements from **`array`** using **`readItem`** (above) starting with the middle element in the array rounded down (i.e. if there are 10 elements indexed 0 to 9, it will start with element 4). For each element it will apply the **`compar`** function to **`target`** and the retrieved element. If the **`compar`** function returns 0 (indicating a match), this function should return the index of the matching element. If the **`compar`** function returns a value of less than zero (indicating the retrieved element precedes the target) it should repeat the search on all higher indexed elements. If the **`compar`** function returns a value of greater than zero (indicating the retrieved element comes after the target) it should repeat the search on all lower indexed elements. If they function processes the final element without finding a match, it should return a value of -1. (This is a binary search.)

You can write additional helper functions as necessary to make sure your code is modular, readable, and easy to modify.

Header File

Use the #ifndef…#define…#endif construct (Lecture 02) in your header file to prevent problems if you header file is included multiple times.

Testing

You are responsible for testing your code to make sure that it works as required. The CourseLink web-site will contain some test programs to get you started. However, we will use a different set of test programs to grade your code, so you need to make sure that your code performs according to the instructions above by writing more test code.

Your assignment will be tested on the standard SoCS Virtualbox VM (http://socs.uoguelph.ca/SoCSVM.zip) which will be run using the Oracle Virtualbox software (https://www.virtualbox.org/wiki/Downloads). If you are developing in a different environment, you will need to allow yourself enough time to test and debug your code on the target machine. We will NOT test your code on YOUR machine/environment.

Full instructions for using the SoCS Virtualbox VM can be found at: https://wiki.socs.uoguelph.ca/students/socsvm.

Your program must free all the memory that it allocates and not allocate an excess of memory.

Makefile

You will create a makefile that supports the following targets:

`all:` this target should generate array.o.

All programs and .o files must be compiled with the **`-c99 -Wall -pedantic`** options and compile without any errors or warning.

`clean:` this target should delete all **`.o`** files.

`array.o:` this target should create the object file, **`array.o`**, by compiling the **`array.c`** file.

All compilations and linking must be done with the **`-Wall -pedantic -std=c99`** flags and compile and link **without any warnings or errors**.

Git

You must submit your .c, .h and makefile using git to the School's git server. Only code submitted to the server will be graded. Do ***not*** e-mail your assignment to the instructor. We will only grade one submission; we will only grade the last submission that you make to the server and apply any late penalty based on the last submission. So once your code is complete and you have tested it and you are ready to have it graded make sure to commit and push all of your changes to the server, and then do not make any more changes to the A2 files on the server.

Academic Integrity

Throughout the entire time that you are working on this assignment. You must not look at another student's code, now allow your code to be accessible to any other student. You can share additional test cases (beyond those supplied by the instructor) or discuss what the correct outputs of the test programs should be, but do not share ANY code with your classmates.

Also, do your own work, do not hire someone to do the work for you.

Grading Rubric

newPerformance	1
newArray	1
readItem	2
writeItem	2
contract	2
freeArray	1
appendItem	2
insertItem	2
prependItem	2
deleteItem	2
style	2
makefile	2
findItem	2
searchItem	2
Total	25

Ask Questions

The instructions above are intended to be as complete and clear as possible. However, it is YOUR responsibility to resolve any ambiguities or confusion about the instructions by asking questions in class, via the discussion forums, or by e-mailing the course e-mail.