Assignment 2

Version 2.00 (last update: Sept. 27, 14:00)
Changes highlighted in yellow
Due date: Thu, Sept 30, 11:59 PM

The memsys data structure described in the assignment is now passed to all of the array functions. Due to the late change in the assignment, no late penalties will be applied to assignments submitted before Tuesday, October 5, 11:59 PM.

Summary and Purpose

For this assignment, you will be writing a collection of C functions that operate like arrays. You will build your functions on a set of functions provided by the instructor. You will use this 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.

memsys

The instructor has provided a library of constants, structures and functions which allocate and free memory and access locations in memory for reading or writing. This library computes 4 statistics about the operations performed on memory and will be used to evaluate your assignment's performance. You ***must*** use this library to implement your assignment. The library provides the following functions:

```
struct memsys *init( int capacity, int max_mallocs );
```

This function initializes the **`memsys`** stem. It takes two parameters: **`capacity`**, which is the total number of bytes of memory that will be made available by the **`memsys`** system; **`max_mallocs`**, which is the total number of blocks of memory that can be allocated. The function returns a pointer to the **`memsys`** structure which is used in all other functions in the **`memsys`** library.

void shutdown(struct memsys *memsys_str);

This function frees the **memsys** structure and its associated data and is used at the end of a program which uses **memsys**.

int memmalloc(struct memsys *memsys, int bytes);

This function allocates memory within the **memsys** system. It works similarly to the malloc function in C. The parameter **bytes** indicates the number of bytes of memory requested. The function returns an integer (instead of a pointer) which can be used by the **getval** and **setval** functions (below). If the **memsys** system has no more memory to allocate the function returns the constant MEMNULL.

void memfree(struct memsys *memsys, int addr);

This function frees memory within the **memsys** system. It works similar to the **free** function in C. The parameter **addr** is an integer value previously returned by **memmalloc**.

void setval(struct memsys *memsys, void *val, size_t size, int address);

This function sets the value of some memory in the **memsys** system. The location where the data will be stored is given by the integer **address** (which will be based on a previous call to **memmalloc**), the data to be stored is taken from the memory address given by **val**, and the size of the data should be equal to **size**.

void getval(struct memsys *memsys, void *val, size_t size, int address);

This function gets the value of some memory in the **memsys** system. The location where the data will be retrieved from is given by the integer **address** (which will be based on previous call to **memmalloc**), the data will be transferred to the address given by **val**, and the size of the data should be equal to **size**.

void print(struct memsys *memsys);

This function prints out the memory in the **memsys** system, the allocation blocks and the statistics that are tracked.

demo1.c

This is a file containing a main function that exercises and demonstrates the **memsys** system.

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;
  int 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 an integer that indicates the location of array in the **memsys** system.

Basic function prototypes and descriptions for your assignment

All of the functions below, ***must*** be implemented using the **memsys** system. You must not use **malloc**, **free**, or use any variables to store or access the entire array.

```
struct Array *newArray( struct memsys *memsys, 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 using **memsys**, storing the location of that memory in **data**. 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 **memmalloc** function fails.

```
void readItem( struct memsys *memsys, 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 use **getval** to copy **array->width** bytes from the memory location **array->data** offset by the **index** (multiplied by **array->width**) to the memory address given by **dest**.

```
void writeItem( struct memsys *memsys, 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 use **setval** to copy **array->width** bytes from the memory address given by **src** to the memory location **array->data** offset by the **index** (multiplied by **array->width**). If index exactly equals **array->nel**, **array->nel** should be incremented by one.

```
void contract( struct memsys *memsys, 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 memsys *memsys, struct Array *array );
```

This function will **memfree array->data** and **free** the **array** structure itself.

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 memsys *memsys, 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 memsys *memsys, 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 memsys *memsys, struct Array *array, void *src );
```

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

```
void deleteItem( struct memsys *memsys, 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 memsys *memsys, 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 the **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 memsys *memsys, 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. Put your helper functions in your array.c and array.h files.

Header File

Use the #ifndef…#define…#endif construct in your header file to prevent problems if your 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 **`-std=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

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	3
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.