

# **Programming Assignment #1: Smart Arrays**

**COP 3502, Summer 2016**

**Due:** Friday, June 10, *before* 11:59 PM

## **Table of Contents**

Abstract.....	2
1. Overview.....	3
2. SmartArray.h.....	4
3. Function Requirements.....	5
4. Compilation and Testing (CodeBlocks).....	8
5. Troubleshooting: File Not Found Errors with Mac OS X.....	8
6. Compilation and Testing (Linux/Mac Command Line).....	9
7. Getting Started: A Guide for the Overwhelmed.....	10
8. Deliverables.....	11
9. Grading.....	12
10. Special Restrictions.....	12

## **Abstract**

In this programming assignment, you will implement smart arrays (arrays that expand automatically whenever they get too full). This is an immensely powerful and awesome data structure, and it will ameliorate several problems we often encounter with arrays in C (see pg. 3 of this PDF).

By completing this assignment, you will gain advanced experience working with dynamic memory management and structs in C. You will also learn to manage programs that use multiple source files. In the end, you will have an awesome and useful data structure that you can reuse in the future.

## **Attachments**

SmartArray.h  
sample-main{1-5}.c  
sample-output{1-5}.c  
names.txt  
SmartArray-diagram.pdf

## **Deliverables**

SmartArray.c

(Note: Capitalization of your filename matters!)

# 1. Overview

A smart array is an array that grows to accommodate new elements whenever it gets too full. As with normal arrays in C, we have direct access to any index of the smart array at any given time. There are four main advantages to using smart arrays, though:

1. We do not need to specify the length of a smart array when it is created. Instead, it will automatically expand when it gets full. This is great when we don't know ahead of time just how much data we're going to end up holding in the array.
2. We will use `get()` and `put()` functions to access individual elements of the array, and these functions will check to make sure we aren't accessing array positions that are out of bounds. (Recall that C doesn't check whether an array index is out of bounds before accessing it during program execution. That can lead to all kinds of whacky trouble!)
3. If our arrays end up having wasted space (i.e., they aren't full), we can trim them down to size.
4. In C, if we have to pass an array to a function, we also typically find ourselves passing its length to that function as a second parameter. With smart arrays, the length will get passed automatically with the array, as they'll both be packaged together in a struct.

While some languages offer built-in support for smart arrays (such as Java's `ArrayList` class), C does not. That's where you come in. You will implement basic smart array functionality in C, including:

1. automatically expanding the smart array's capacity when it gets full;
2. adding new elements into arbitrary positions in the smart array, or at the end of the smart array;
3. providing safe access to elements at specific positions in the smart array;
4. gracefully signaling to the user (i.e., the programmer (re-)using your code) when he or she attempts to access an index in the smart array that is out of bounds (instead of just segfaulting);
5. ... and more!

In this assignment, your smart array will be designed to hold arrays of strings. A complete list of the functions you must implement, including their functional prototypes, is given below in Section 3, "Function Requirements"). You will submit a single source file, named `SmartArray.c`, that contains all required function definitions, as well as any auxiliary functions you deem necessary. In `SmartArray.c`, you should `#include` any header files necessary for your functions to work, including `SmartArray.h` (see Section 2, "SmartArray.h").

**Note that you will *not* write a `main()` function in the source file you submit!** Rather, we will compile your source file with our own `main()` function(s) in order to test your code. We have attached example source files that have `main()` functions, which you can use to test your code. We realize this is completely new territory for most of you, so don't panic. We've included instructions on compiling multiple source files into a single executable (e.g., mixing your `SmartArray.c` with our `SmartArray.h` and `sample-main1.c`) in Sections 4 and 6 ("Compilation and Testing").

Although we have included sample `main()` functions to get you started with testing the functionality of your code, we encourage you to develop your own test cases, as well. Ours are by no means comprehensive. We will use much more elaborate test cases when grading your submission.

*Good luck! Start early! Come to office hours for help if you get stuck!*

## 2. SmartArray.h

This header file contains the struct definition and functional prototypes for the smart array functions you will be implementing. You should `#include` this file from your `SmartArray.c` file, like so:

```
#include "SmartArray.h"
```

The “quotes” (as opposed to `<brackets>`), indicate to the compiler that this header file is found in the same directory as your source, not a system directory.

You should not modify `SmartArray.h` in any way, and you should not send `SmartArray.h` when you submit your assignment. We will use our own unmodified copy of `SmartArray.h` when compiling your program.

If you write auxiliary functions in `SmartArray.c` (which is strongly encouraged!), you should **not** add those functional prototypes to `SmartArray.h`. Just put those functional prototypes at the top of your `SmartArray.c`.

**Think of `SmartArray.h` as a public interface to the `SmartArray` data structure.** It contains *only* the functions that the end user (i.e., the programmer (re-)using your code) should call in order to create and use a `SmartArray`. You do not want the end user to call your auxiliary functions directly, so you do not put those functional prototypes in `SmartArray.h`. That way, the end user doesn’t need to worry about all your auxiliary functions in order to use an `SmartArray`; everything just works. (And *you* don’t have to worry about the end user mucking everything up by accidentally calling auxiliary functions that he or she shouldn’t be messing around with!)

The basic struct you will use to implement the smart arrays (defined in `SmartArray.h`) is as follows:

```
typedef struct SmartArray
{
    char **array;    // pointer to array of strings
    int size;        // number of elements in array
    int capacity;    // length of array (maximum capacity)
} SmartArray;
```

The `SmartArray` struct contains a double char pointer that can be used to set up a 2D char array (which is just an array of char arrays, otherwise known as an array of strings). `array` will have to be allocated dynamically at runtime. It will probably be the bane of your existence for the next week or so.

The struct also has `size` and `capacity` variables, which store the number of elements in the array (initially zero) and the current length (i.e., maximum capacity) of the array, respectively.

### 3. Function Requirements

In the source file you submit, `SmartArray.c`, you must implement the following functions. You may implement any auxiliary functions you need to make these work, as well. Please be sure the spelling, capitalization, and return types of your functions match these prototypes exactly. In this section, I often refer to `malloc()`, but you're welcome to use `calloc()` or `realloc()` instead, as you see fit.

```
SmartArray *createSmartArray(int length);
```

**Description:** Dynamically allocate space for a new `SmartArray`. Initialize its internal array to be of length `length` or `DEFAULT_INIT_LEN`, whichever is greater. (`DEFAULT_INIT_LEN` is defined in `SmartArray.h`.) Properly initialize pointers in the array to `NULL`, and set the `size` and `capacity` members of the struct to the appropriate values.

**Output:** “-> Created new `SmartArray` of size `<N>`.” (Output should not include the quotes. Terminate the line with a newline character, ‘\n’. `<N>` should of course be the length of the new array, without the angled brackets.)

**Returns:** A pointer to the new `SmartArray`, or `NULL` if any calls to `malloc()` failed.

```
SmartArray *destroySmartArray(SmartArray *smarty);
```

**Description:** Free any dynamically allocated memory associated with the `SmartArray` struct and return `NULL`.

**Returns:** `NULL` pointer.

```
SmartArray *expandSmartArray(SmartArray *smarty, int length);
```

**Description:** Dynamically allocate a new array of length `length`. Copy the contents of `smarty`'s old array into the new array. Free any memory associated with the old `smarty->array` that is no longer in use, then set `smarty->array` to point to the newly created array. Be sure all pointers are properly initialized. Update the `size` and `capacity` of the `SmartArray` (if applicable).

Note: If `length` is less than or equal to `smarty`'s current array capacity, or if the `smarty` pointer is `NULL`, you should NOT modify the `SmartArray` at all. In that case, just return from the function right away without producing any output.

**Output:** “-> Expanded `SmartArray` to size `<N>`.” (Output should not include the quotes. Terminate the line with a newline character, ‘\n’. `<N>` should be the new length of the array, without the angled brackets. Do NOT produce any output if you the array is not expanded.)

**Returns:** A pointer to the `SmartArray`, or `NULL` if any calls to `malloc()` failed.

```
SmartArray *trimSmartArray(SmartArray *smarty);
```

**Description:** If smarty's capacity is greater than its current size, trim the length of the array to the current size. You will probably want to `malloc()` a new array to achieve this. If so, avoid memory leaks as you get rid of the old array. Update any members of smarty that need to be updated as a result of this action.

**Output:** "-> Trimmed SmartArray to size <N>." (Output should not include the quotes. Terminate the line with a newline character, '\n'. <N> should be the new length of the array, without the angled brackets. Do NOT produce any output if the length of the array is not reduced by this function.)

**Returns:** A pointer to the SmartArray, or NULL if `malloc()` failed or if smarty was NULL.

```
char *put(SmartArray *smarty, char *str);
```

**Description:** Insert a *copy* of str into the next unused cell of the array. If the array is already full, call `expandSmartArray()` to grow the array to length (`capacity * 2 + 1`) before inserting the new element. When copying str into the array, only allocate the minimum amount of space necessary to store the string.

**Returns:** A pointer to the *copy* of the new string that was inserted into the array, or NULL if the string could not be added to the array (e.g., `malloc()` failed, or smarty or str was NULL).

```
char *get(SmartArray *smarty, int index);
```

**Description:** Attempts to return the element at the specified index. This is where you protect the user from going out-of-bounds with the array.

**Returns:** A pointer to the string at position index of the array, or NULL if index was out of bounds or the smarty pointer was NULL.

```
char *set(SmartArray *smarty, int index, char *str);
```

**Description:** If the array already has a valid string at position index, replace it with a *copy* of str. Otherwise, the operation fails and we simply return NULL. Ensure that no more space is used to store the new copy of str than is absolutely necessary (so, you might have to use `malloc()` and `free()` here).

**Returns:** A pointer to the copy of the string placed in the SmartArray, or NULL if the operation failed for any reason (e.g., invalid index, or smarty or str was NULL).

```
char *insertElement(SmartArray *smarty, int index, char *str);
```

**Description:** Insert a *copy* of str at the specified index in the array. Any elements to the right of index are shifted one space to the right. If the specified index is greater than the array's size, the element being inserted should be placed in the first empty position in the array.

(Continued from previous page...) As with the `put()` function, if the `SmartArray` is already full, call `expandSmartArray()` to grow the array to length  $(\text{capacity} * 2 + 1)$  before inserting the new element. When copying `str` into the array, only allocate the minimum amount of space necessary to store the string.

**Returns:** A pointer to the copy of the string inserted into the array, or `NULL` if insertion fails for any reason (e.g., `malloc()` failed, or `smarty` or `str` was `NULL`).

```
int removeElement(SmartArray *smarty, int index);
```

**Description:** Remove the string at the specified index in the array. Strings to the right of `index` are shifted one space to the left, so as not to leave a gap in the array. The `SmartArray`'s `size` member should be updated accordingly. If `index` exceeds the `SmartArray`'s `size`, nothing is removed from the array.

**Returns:** 1 if an element was successfully removed from the array, 0 otherwise (including the case where the `smarty` pointer is `NULL`).

```
int getSize(SmartArray *smarty);
```

**Description:** This function returns the number of elements currently in the array. We provide this function to discourage the programmer from accessing `smarty->size` directly. That way, if we decide to change the name or meaning of the `size` variable in our `SmartArray` struct, the programmers who download the latest version of our code can get it working right out of the box; they don't have to go through their own code and change all instances of `smarty->size` to something else, as long as we provide them with a `getSize()` function that works.

**Returns:** Number of elements currently in the array, or -1 if the `smarty` pointer is `NULL`.

```
void printSmartArray(SmartArray *smarty);
```

**Description:** Print all strings currently in the array.

**Output:** Print all strings currently in the array. Print a newline character, `'\n'`, after each string. If the `SmartArray` pointer is `NULL`, or if the array is empty, simply print `"(empty array)"` (without quotes), followed by a newline character, `'\n'`.

```
double difficultyRating(void);
```

**Returns:** A double indicating how difficult you found this assignment on a scale of 1.0 (ridiculously easy) through 5.0 (insanely difficult).

```
double hoursSpent(void);
```

**Returns:** An estimate (greater than zero) of the number of hours you spent on this assignment.

## 4. Compilation and Testing (CodeBlocks)

The key to getting a multiple files to compile into a single program in CodeBlocks (or any IDE) is to create a project. Here are the step-by-step instructions for creating a project in CodeBlocks, importing `SmartArray.h`, `sample-main1.c`, and the `SmartArray.c` file you've created (even if it's just an empty file so far).

1. Start CodeBlocks.
2. Create a New Project (*File -> New -> Project*).
3. Choose "Empty Project" and click "Go."
4. In the Project Wizard that opens, click "Next."
5. Input a title for your project (e.g., "SmartArray").
6. Choose a folder (e.g., Desktop) where CodeBlocks can create a subdirectory for the project.
7. Click "Finish."

Now you need to import your files. You have two options:

1. Drag your source and header files into CodeBlocks. Then right click the tab for **each** file and choose "Add file to active project."
- or –
2. Go to *Project -> Add Files....* Browse to the directory with the source and header files you want to import. Select the files from the list (using CTRL-click to select multiple files). Click "Open." In the dialog box that pops up, click "OK."

You should now be good to go. Try to build and run the project (F9).

Note that if you import both `sample-main1.c` and `sample-main2.c`, the compiler will complain that you have multiple definitions for `main()`. You can only have one of those in there at a time. You'll have to swap them out as you test your code.

Even if you develop your code with CodeBlocks on Windows, you ultimately have to transfer it to the Eustis server to compile and test it there. See the following page (Section 6, "Compilation and Testing (Linux/Mac Command Line)") for instructions on command line compilation in Linux.

## 5. Troubleshooting: File Not Found Errors with Mac OS X

If you encounter a file-not-found error on your Mac, you might need to put your input files in an unusual directory. Use my `pwd()` ("print working directory") function to print the directory where your IDE wants those input files. [pwd\(\) is defined in file-read.c, posted May 24 in Webcourses.](#)



## 6. Compilation and Testing (Linux/Mac Command Line)

To compile multiple source files (.c files) at the command line:

```
gcc SmartArray.c sample-main1.c
```

By default, this will produce an executable file called a.out that you can run by typing:

```
./a.out
```

If you want to name the executable something else, use:

```
gcc SmartArray.c sample-main1.c -o SmartArray.exe
```

...and then run the program using:

```
./SmartArray.exe
```

Running the program could potentially dump a lot of output to the screen. If you want to redirect your output to a text file in Linux, it's easy. Just run the program using the following:

```
./SmartArray.exe > whatever.txt
```

This will create a file called whatever.txt that contains the output from your program.

Linux has a helpful command called diff for comparing the contents of two files, which is really helpful here since we've provided a sample output file. You can see whether your output matches ours exactly by typing, e.g.:

```
diff whatever.txt sample-output1.txt
```

If the contents of whatever.txt and sample-output1.txt are exactly the same, diff won't have any output. It will just look like this:

```
seansz@eustis:~$ diff whatever.txt solution.txt
seansz@eustis:~$ _
```

If the files differ, it will spit out some information about the lines that aren't the same. For example:

```
seansz@eustis:~$ diff whatever.txt solution.txt
6c6
< Size of array: 0
---
> Size of array: -1
seansz@eustis:~$ _
```

## 7. Getting Started: A Guide for the Overwhelmed

Okay, so, this might all be overwhelming, and you might be thinking, “Where do I even start with this assignment?! I’m in way over my head!”

Don’t panic! We’re here to help in office hours, and here’s my general advice on starting the assignment (as well as a few hints and spoilers):

1. First and foremost, start working on this assignment early. Nothing will be more frustrating than running into unexpected errors or not being able to figure out what the assignment is asking you to do on the day that it is due.
2. Start by creating a skeleton `SmartArray.c` file. Add a header comment, add some standard `#include` directives, and be sure to include `SmartArray.h` from your source file. Then copy and paste each functional prototype from `SmartArray.h` into `SmartArray.c`, and set up all those functions return dummy values (zero, `NULL`, etc.).
3. Test that your `SmartArray.c` source file compiles. If you’re at the command line on a Mac or in Linux, your source file will need to be in the same directory as `SmartArray.h`, and you can test compilation like so:

```
gcc -c SmartArray.c
```

Alternatively, you can try compiling it with one of my sample-main source files, like so:

```
gcc SmartArray.c sample-main1.c
```

For more details, see Section 6, “Compilation and Testing (Linux/Mac Command Line).”

If you’re using an IDE (i.e., you’re coding with something other than a plain text editor and the command line), open up your IDE and start a project using the instructions above in Section 4, “Compilation and Testing (CodeBlocks)”. Import `SmartArray.h`, `sample-main1.c`, and your new `SmartArray.c` source file, and get the program compiling and running before you move forward. (Note that CodeBlocks is the only IDE we officially support in this class.)

4. Once you have your project compiling, go back to the list of required functions (Section 3, “Function Requirements”), and try to implement one function at a time. Always stop to compile and test your code before moving on to another function!
5. You’ll *probably* want to start with the `createSmartArray()` function. (Alternatively, `printSmartArray()` might be a good starting point, as well.)

As you work on `createSmartArray()`, write your own `main()` function that calls `createSmartArray()` and then checks the results. For example, you’ll want to ensure that `createSmartArray()` is returning a non-`NULL` pointer to begin with, and that the fields inside the `SmartArray` struct that it created are properly initialized when you examine them back in `main()`. If you’re uncertain about how to call certain functions, read through my sample main files for examples.

6. After writing `createSmartArray()`, I would probably work on the `printSmartArray()` function, because it will be immensely useful in debugging your code as you work. Here's how I'd test these functions at first: In your own `main()` function, call `createSmartArray()`. Then, back in `main()`, manually insert one or two strings into the smart array before passing it to the `printSmartArray()` function. Make sure everything works as intended and the output is as expected. If not, trace carefully through your code to see what went wrong.
7. If you get stuck, draw diagrams. Make boxes for all the variables in your program. If you're dynamically allocating memory, diagram them out and make up addresses for all your variables. Trace through your code carefully using these diagrams.
8. With so many pointers, you're bound to encounter errors in your code at some point. Use `printf()` statements liberally to verify that your code is producing the results you think it should be producing (rather than making assumptions that certain components are working as intended). You should get in the habit of being immensely skeptical of your own code and using `printf()` to provide yourself with evidence that your code does what you think it does.
9. When looking for a segmentation fault, you should always be able to use `printf()` and `fflush()` to track down the *exact* line you're crashing on.
10. You'll need to examine a lot of debugging output. You might want to set up a function that prints debugging strings only when you `#define` a `DEBUG` value to be something other than zero, so you can easily flip debugging output on and off. (Just be sure to remove your debugging statements before you submit your assignment, so your code is nice and clean and easy for us to read.)

## 8. Deliverables

Submit a single source file, named `SmartArray.c`, via Webcourses. The source file should contain definitions for all the required functions (listed above), as well as any auxiliary functions you need to make them work.

Your source file **must not** contain a `main()` function. Do not submit additional source files, and do not submit a modified `SmartArray.h` header file. Your source file (without a `main()` function) should compile at the command line using the following command:

```
gcc -c SmartArray.c
```

Be sure to include your name and NID as a comment at the top of your source file.

*Continued on the following page...*

## 9. Grading

The *expected* scoring breakdown for this programming assignment is:

50%	correct output for test cases
25%	implementation details (manual inspection of your code)
5%	difficultyRating() returns a double in the expected range
5%	hoursSpent() returns a double in the expected range
5%	source file is named correctly (SmartArray.c); capitalization counts
10%	adequate comments and whitespace; source includes student name and NID

**Your program must be submitted via Webcourses, and it must compile and run on Eustis to receive credit. Programs that do not compile will receive an automatic zero.**

Your grade will be based largely on your program's ability to compile and produce the *exact* output expected. Even minor deviations (such as capitalization or punctuation errors) in your output will cause your program's output to be marked as incorrect, resulting in severe point deductions. The same is true of how you name your functions and their parameters. Please be sure to follow all requirements carefully and test your program thoroughly.

Additional points will be awarded for style (proper commenting and whitespace) and adherence to implementation requirements. For example, the graders might inspect your `destroySmartArray()` function to see that it is actually freeing up memory properly.

## 10. Special Restrictions

Please carefully adhere to the following restrictions:

1. It's very important that you do not include a `main()` function in your `SmartArray.c` file. If your source file has a `main()` function, it will fail to compile during testing, and you will not receive credit for the assignment.
2. Please do not use global variables.
3. Please avoid mid-function variable declarations. Within each function, all the variables you use should be declared at the top of that function.
4. Please do not make system calls; eliminate `system("pause")` from your vocabulary.