CptS 122 Laboratory 2:
A Dynamic Array

Goal: In lecture we will be discussing the use of a linked list to store a list of data. In this lab we will explore the use of a dynamic array to store a list of data.

All your work is to be completed in the lab. Some tasks will just require you to read and understand the material. For such tasks, once you are comfortable with the material, you may move on to the next task. Other tasks require you to write code to accomplish the task. When you complete a task that requires programming, please contact the TA to demonstrate your code to him. Once the TA has recorded your completion of the task, you may move on to the next task.

You are strongly encouraged to help your classmates or to seek help from them. When you are helping others, please make sure you help by explaining things and are not merely providing answers.

Task 1: (20 points) Creating persistent variables: the use of `static`.

If we declare a variable as being `static`, the value of that variable persist in memory for the duration of the program execution. If a variable is a global variable, i.e., declared outside of any function, the variable can be used by all functions that appear in the same file (and appear after the declaration of the variable). Putting these two facts together provides a nice way to both hide and share data. Let's consider a very simple example of this.

In this example we will declare two variables as being `static`. We'll call these variables `debt` (which is our version of the national debt) and `limit` (which is our version of the national debt limit). These two variables will be declared in a file called `national-debt.c`. We won't make these values directly accessible to the rest of our program (i.e., the `main()` function as you will see shortly), but rather we will provide functions that serve as an interface to these values.

With that in mind, let's create two .c files and a header file. These files are:

- `national-debt.h`: header file that provides various things including prototypes of three functions we will use to manipulate our version of the "national debt."
- `national-debt.c`: C code that provides the functions that manipulate our version of the national debt.
- `debt-demo.c`: This contains the `main()` function used to demonstrate and test our national-debt functions.

You can obtain this code from the class Web site (scroll down to Lab2 under Assignments column):

`http://www.eecs.wsu.edu/~nroy/courses/cpts122/`

Here is the code for each of these files. First, here is the header file `national-debt.h`.

```
// national-debt.h
#ifndef __NATIONAL_DEBT_H__
#define __NATIONAL_DEBT_H__

#include <stdio.h>

int getDebt(void);   // Show the current debt.

int getLimit(void);  // Show the current debt limit.

void addToDebt(void);     // Increment the debt.

#endif // __NATIONAL_DEBT_H__
```

We will assume we can only increase the debt and we can only increase it by a single unit at a time. Thus, the function `addToDebt()` only increases the debt by a single step. But, importantly, we will further assume we can never exceed the debt limit. If we are at the limit and the limit is not zero, we will automatically increase it by doubling it. If the limit is currently zero, we will increase it to one. Here is the file `national-debt.c`:

```
// national-debt.c
#include "national-debt.h"

static int debt=0;   // Initialize debt to zero.
static int limit=0;  // Initialize (debt) limit to zero.

// Show the current debt.
int getDebt(void) {
  return debt;
}

// Show the current (debt) limit.
int getLimit(void) {
  return limit;
}

/* Increment the debt, but ensure we never exceed the limit.  If the
 * limit is currently 0, set it to one.  Otherwise, the limit should
 * be increased by doubling its current value. */
void addToDebt(void) {
  if (debt == limit) { // debt at limit; thus must increase limit
    if (limit == 0) {
      limit = 1;    // initial increase
    } else {
      limit *= 2;   // all subsequent increases -> doubling
    }
```

```
    printf("New limit = %u\n", limit);
  }
  debt++;
  return;
}
```

Here is code we can use to help demonstrate that our code is working properly:

```
// debt-demo.c
#include "national-debt.h"

int main() {
  printf("debt, limit: %u %u\n", getDebt(), getLimit());
  addToDebt();
  addToDebt();
  addToDebt();
  printf("debt, limit: %u %u\n", getDebt(), getLimit());
  addToDebt();
  addToDebt();
  addToDebt();
  printf("debt, limit: %u %u\n", getDebt(), getLimit());
  addToDebt();
  addToDebt();
  addToDebt();
  printf("debt, limit: %u %u\n", getDebt(), getLimit());
  return 0;
}
```

Importantly, note that the `main()` function in the file `debt-demo.c` cannot directly access or manipulate the variables `debt` and `limit` in the file `national-debt.c`. Although these values are visible to all the functions in `national-debt.c`, they are hidden from `main()`. Nevertheless, `main()` can "get" the values of these variables and increase `debt` with the appropriate function calls.

To complete this first task, simply compile, run, and make sure you understand this code! Look at the output and make sure it makes sense. If you have any questions about the code, ask the TA.

Task 2: (20 points) The `realloc()` function.

Now, we will make just a few changes to the framework we used for the national-debt code and create an array that we can use to store an arbitrary number of floats. Think of the "debt" as the number of meaningful elements we have stored in our array and think of the "limit" as the total capacity of the array. When the array is full, we can use the function `realloc()` to increase the memory associated with the array, i.e., increase the capacity of the array.

Keep in mind when we say "array" we really mean a block of memory we reference with a pointer, i.e., we use the pointer as if it were an array. The memory associated with this pointer is all

allocated dynamically.

The prototype for `realloc()` is

```
void *realloc(void *old_ptr, size_t new_size);
```

`realloc()` returns the address of memory that is `new_size` bytes. This size may be larger or smaller than the amount of memory previously associated with the first argument, i.e., `old_ptr`. If the reallocation was unsuccessful, `realloc()` returns `NULL`. `realloc()` uses the function `free()` to free the memory associated with `old_ptr` (so we don't have to worry about doing that).

Let's consider a simple example where we use `realloc()` to increase the size of a `float` array from 10 to 100 elements. In this case, if the reallocation failed, we will use the standard library function `exit()` to terminate execution of the program.

```
float* my_array;
float* tmp;

// initialize my_array to have 10 elements
my_array = (float*)malloc(10 * sizeof(float));

         //  .
         //  .  (missing code not shown)
         //  .

tmp = (float*)relloc(my_array, 100 * sizeof(float));
if (tmp == NULL) {
  printf("Memory allocation error.  Terminating...\n");
  exit(-1);
}
my_array = tmp;
```

Note: To use either `realloc()` or `exit()` we must include the header file `stdlib.h`. Also note that the variable `tmp` is not strictly necessary. A slightly cleaner version of this is as follows:

```
float* my_array;

// initialize my_array to have 10 elements
my_array = (float*)malloc(10 * sizeof(float));

         //  .
         //  .  (missing code not shown)
         //  .

my_array = (float*)realloc(my_array, 100 * sizeof(float));
```

```
  if (my_array == NULL) {
    printf("Memory allocation error.  Terminating...\n");
    exit(-1);
  }
```

Now, let's again create three files, but where we want to implement and demonstrate the use of a dynamic array. The files are as follows:

- `dynamic-floats.h`: header file.
- `dynamic-floats.c`: C code that provides functions to use the dynamic float array.
- `floats-demo.c`: This contains the `main()` function used to demonstrate and test the dynamic float array.

This code is again available from the class Web site.

In this case we will define a function to add (or append) a float to the end of the array, a function to get the number of elements in the array, and a function to get the float value at a specified location in the array.

Here is the header file `dynamic-floats.h`:

```
// dynamic-floats.h
#ifndef __DYNAMIC_FLOATS_H__
#define __DYNAMIC_FLOATS_H__

#include <stdio.h>
#include <stdlib.h>

// Show number of "meaningful" elements in the array.
int getNumElements(void);

// Show float at specified location.
float getFloat(int location);

// Add an element to the end of the array.
void addToArray(float x);

#endif // __DYNAMIC_FLOATS_H__
```

Here is the file `dynamic-floats.c`:

```
// dynamic-floats.c
#include "dynamic-floats.h"

// Initialize my_array (pointer) to NULL.
```

```
static float* my_array=NULL;
// Initialize number of elements and number of allocated elements to zero.
static int num_elements=0;   // current number of meaningful elements
static int num_allocated=0;  // total possible number of elements

// Show the current number of elements.
int getNumElements(void) {
  return num_elements;
}

// Show the float at the specified location.
float getFloat(int location) {
  return my_array[location];
}

/* Add the specified value x to the end of the array.  If needed,
 * increase the size of the array using realloc(). */
void addToArray(float x) {
  if (num_elements == num_allocated) { // need to increase size
    if (my_array == NULL) {  // Initialize my_array if first element.
      my_array = (float*)malloc(sizeof(float));
      num_allocated = 1;
    } else {              // Double size of my_array otherwise.
      num_allocated *= 2;
      my_array = (float*)realloc(my_array, num_allocated * sizeof(float));
      if (my_array == NULL) { // realloc() failed.  Terminate.
        printf("Reallocation failed.  Terminating.\n");
        exit(-1);
      }
    }
  }
  my_array[num_elements] = x; // assign float to end of my_array
  num_elements++;
  return;
}
```

And here is the file `floats-demo.c` where the `main()` function is used in an attempt to demonstrate that our code is working properly.

```
// floats-demo.c
#include "dynamic-floats.h"

int main() {
  int i;

  addToArray(3.0);
  addToArray(7.0);
  addToArray(10.0);
```

```
  addToArray(13.0);
  addToArray(17.0);

  for (i=0; i<getNumElements(); i++)
    printf("array[%d] = %f\n", i, getFloat(i));

  return 0;
}
```

Compile, execute, and run this code. Make sure you understand it and are comfortable with what `realloc()` does.

Now, to complete this task, you have to write a function that has the following prototype:

```
void printFloats(int location);
```

This function should print all the elements from the location given as the argument (i.e., `location`) until the last element. The elements should be printed one per line. If `location` is greater than the number of elements, an error message should be printed showing the number of elements. This function should be in the file `dynamic-floats.c`. Modify the header file accordingly. Once you are confident your function is behaving properly, demonstrate your code to the TA. Then move on to the next two tasks which build on this code.

Task 3: (30 points) Removing an element from the array.

Write a function to *remove* an element from the array at the specified location. This function has the following prototype:

```
void removeFloat(int location);
```

Ensure that the location is a valid one (i.e., within the range of valid element indices). Print an error message if it is not.

Now, what do we mean by removing an element? If the element is anywhere other than the last element, move all the subsequent elements "up" one location in the list, i.e., in general all elements after the removed element will have their index changed from n to $n-1$. After doing this, be sure to decrement the number of elements.

If the element is the last element in the array, you don't have to do anything other than decrement the number of elements (since that value effectively indicates the end of the list).

Modify your test/demo function, i.e., `main()`, to demonstrate that your `removeFloat()` function works. Demonstrate your code to the TA. When he is satisfied your code is correct, you are free to move on to the next task.

Optional: You are not required to shrink the total allocated space for the array. However, if you want to implement that, that would be ideal! You should only do this shrinkage if the number of elements is now half of the overall allocated space. In this case cut the space in half (i.e., since we increased the memory by a factor of two when needed, let's decrease the memory by the same factor when shrinking it).

Task 4: (30 points) Adding an element at a specified location.

We already have a function to add an element to the end of the array. Now let's write a function to add an element to the array but at a specified location. Print an error message if the location is not a valid one.

Regarding this point, if there are currently N elements in the array, a valid location is anywhere from 0 to N. Thus, importantly, N may beyond the end of the currently allocation array size. For example, let's say there are currently 4 elements in the array and the array size (`num_allocated`) is also 4. We can say we want the float added at location "4" but this would be the fifth element in the array (don't forget the start of the array has an index of zero!).

This function should have the following prototype:

```
void addToArrayAt(float x, int location);
```

This prototype should appear in `dynamic-floats.h` and the associated code should be in `dynamic-floats.c`. Modify your demonstration/testing function to demonstrate that this function works properly. When your function/program is working properly, show your code to the TA.

Submitting Labs:
You are not required to submit your lab solutions. However, you should keep them in a folder that you may continue to access throughout the semester. You should not store your solutions to the local C: drive on the Sloan 353 machines. These files are erased on a daily basis.

Grading Guidelines:
This lab is worth 100 points. Your lab grade is assigned based on completeness and effort. To receive credit for the lab you must show up on time and continue to work on the problems and complete all the problems until the TA has dismissed you.