**Prelab 4** (for September 23rd)

Programming isn't just about writing a program for one-time use to solve some problem. Large pieces of software (e.g., Photoshop) are constructed by creating special-purpose functions that can be used as building blocks to create even more powerful functions.

A programming language like C provides the bare minimum set of functions, e.g., input/output (I/O) functions like `scanf` and `printf` along with standard math and string functions. It doesn't need to provide much more than that because the real power of a general-purpose programming language is that it allows programmers to expand the language by creating libraries of new functions.

As I described in lecture, it's possible to create a set of functions to improve the way arrays can be used in C. Specifically, we can create a function:

```
array = createIntArray(n);
```

which allocates an integer array of size `n` in a way that stores the size with the array so that it can be accessed when needed. This means that an array allocated in this way can be passed to functions without having to use a separate parameter to tell the function the size of the array. This is possible because we can also provide a function:

```
size = getIntArraySize(array);
```

which returns the size of an array when needed. Of course we'll also need a function to free the array:

```
freeIntArray(array);
```

As discussed in lecture, the user can't simply use `free(array)` because that requires the address that was returned by `malloc`. The address returned from `createIntArray` is ***not*** that address. (Yes, the function `createIntArray` allocates memory using `malloc`, but what else does it do?)

For the first part of this prelab you will implement functions to allocate; get the size of; and free arrays of doubles instead of int arrays. For example, if we want to allocate an array of `n` doubles a user would call your `createDoubleArray` function as:

```
array = createDoubleArray(n);
```

At first glance you might think the implementation will be almost identical to the `int` case except with `int` declarations changed to double. Close, but not quite. As I discussed in lecture, the `int` case is very simple because the number of elements in the array is an `int`, which is the same data type as all other elements of the array. That's not the case here because elements are `sizeof(double)` and we're wanting to store something that is only `sizeof(int)` before the beginning of the array.

It turns out that the basic logic is pretty much the same in that you'll allocate a chunk of memory using something like `malloc(n*sizeof(double)+sizeof(int))`, but how you interpret that chunk of memory (e.g., as an array of doubles or an array of ints) will make a big difference. Any choice can work, but one choice will make things much simpler with much less need for lots of complicated casting. The key is to recognize that a chunk of memory is just a chunk of memory. For example, because `sizeof(int)` happens to be the same as `sizeof(float)`, we could dynamically allocate an array of 1000 ints as:

```
int *array;
array = malloc(1000*sizeof(float)); // This is an int array
```

Again, it's critical to understand that `malloc` just returns the address of a chunk of memory of the size requested. You can verify that `malloc` cannot possibly know what that chunk of memory will be used for because the only value it receives as a parameter is an integer. It doesn't know whether the user calculated that parameter value as 1000*`sizeof(int)` or 1000*`sizeof(float)` or in some other completely different way that doesn't even use the `sizeof` function.

After you've completed and tested your three functions for creating; getting the size of; and freeing double arrays you'll realize that they were almost as simple as the integer-array case we went through in lecture. The last part of this prelab is to create another set of functions that will allow a user to create an array of *any* data type. This can be achieved using a prototype like the following:

```
void * createArray(int n, int dataTypeSize); // ignore return type for now
```

where the user is expected to pass the number of elements in the array *and the size of each of those elements*. We need to know both pieces of information so that we can determine the amount of memory to request from malloc. In the previous examples we could calculate the size as "*n*sizeof(int)*" or "*n*sizeof(double)*" because we already knew the data type. A general-purpose function won't know the data type in advance and so must require the user to provide the data-type size as a parameter like the following, which creates an array of 1000 doubles:

```
double *array;
array = createArray(1000, sizeof(double));
```

Note that the pointer returned by CreateArray is being assigned to a double* pointer without a need for casting. Does that mean that the return type of CreateArray must be a pointer to double? No, the return type of `void*` in the prototype represents a generic pointer that can be assigned to a pointer of any type (and the reverse is also true). This is exactly what we need to create general-purpose (generic) functions that can return pointers objects of any desired data type. Just to really nail things down, a user could allocate an array of 1000 floats like the following:

```
float *array;
array = createArray(1000, sizeof(float));
```

This is just like the previous example except that the user wants an array of floats instead of an array of doubles, so she passed sizeof(float) instead of sizeof(double). A different way to think about all of this is that instead of asking the user for a single integer giving the total size of the requested array – e.g., as malloc requires – we're asking the user to provide the number of elements and the size of each element as separate parameters. We can then multiply them together (and add sizeof(int) to the result) and pass that size to malloc.

What you'll find is that the generic createArray, getArraySize, and freeArray functions will be almost as simple as the special-purpose functions associated with createIntArray and createDoubleArray and yet can do everything that those functions do. In other words, by thinking big – i.e., about the most general case – we can design a set of general-purpose functions instead of separate sets of special-purpose functions for every possible data type. In fact, we'll see later that createArray will even work for new data types that users may define in the future.

To summarize, you'll implement six functions for this prelab. However, they'll each be only a few lines long, and the second three functions will look very much like the first three. At the end of the process you should have a strong understanding of pointers, pointer arithmetic, and memory allocation. More generally, you'll hopefully also get a feel for what that understanding can allow you to achieve.

- JKU