CS 352 (Fall 16): System Programming and UNIX

Project #9

Objects in C - `ArrFlex`

due at 9pm, Wed 16 Nov 2016

1 Overview

This project is to help you see how large libraries can implement something a lot like Object Oriented Programming in C. In C, we represent an object with a struct, and each member function is implemented as a function which takes a pointer as the first argument.

In this project, you will be implementing a small library, called `ArrFlex`. (This was originally modeled on Java's ArrayList class, but we've simplified it.) We have provided a header file named `arrFlex.h` which declares an (empty) struct and a number of functions. You will have to add the fields to this struct - but you are not allowed to change any of the function declarations.

You will write a C file (named `arrFlex.c`), which provides the implementations for all of the functions declared in the header. You will also provide a set of testcases (which are all C files!) which will test to see if the functions that you have written work properly.

2 Objects in C

You are familiar with code like this from Java:

```
public class FooBar
{
    public int    x;
    public String name;

    public int getNameLength()
    {
        return name.length();
    }
}
```

This code declares a `class;` we can create objects of that class. Each object has some data fields, and some methods associated with it.

You will remember that each data method includes a hidden parameter, known as `this` - it is a reference to the object on which the method was called. Using this, you can access the fields of the object. (In some cases, like the example above, we skip even mentioning 'this' and access the field directly - but under the covers, Java is using the 'this' pointer.)

In C, we can do the same thing with a struct. Of course, C doesn't have `public/private`, and it doesn't have the fancy syntax for calling the methods - but we can accomplish the same thing:

```
typedef struct FooBar
{
    int   x;
    char *name;
} FooBar;

int fooBar_getNameLength(FooBar *obj)
{
    return strlen(obj->name);
}
```

In C, the syntax is kind of clunky, but the same basic principles still work. So Java's "classes" are, in fact, little more than syntax sugar for things that you can already do in C. (Syntax sugar is good - I'm in favor of it. But it's not **necessary.**)

3 The Class You Will Define

In this program, you will implement `ArrFlex`. This class was inspired by Java's `ArrayList` - but it's much simplified. Instead of requiring a linked list, it's OK for you to implement each object using a single array; also, this class only stores characters (not any other type).

Each object of type `ArrFlex` represents an array of a certain size. When you create a new object, it is empty; that is, its size is zero. You can expand it using the `add()` or `insert()` functions, or make it shorter with `delete()`. You can duplicate an array with `clone()` or `subArray()`; append two arrays together with `append()`, or actually turn it into a string with `toString()`. You can get the current size with `size()`, and read or modify existing elements with `get()` and `set()`. And of course, you can allocate a new object with `new()` and free one with `free()`.

Each one of these functions (except for `new()`, of course) takes an `ArrFlex*` as the first parameter, which is, essentially, the `this` pointer for the function. It tells you what object to read or modify. Most functions also include return codes; you will find that (since C doesn't have exceptions) you will need return codes for almost all functions.

The precise definition for every function is given in the header file I've provided, `arrFlex.h`. You will find that each function is actually declared with the `arrFlex_` prefix - that is, the `new()` "method" is named `arrFlex_new()`.

NOTE: When you resize your array, generally you should double the size (not increase it by one). Expect that we will provide at least one testcase where we call `add()` thousands of times in a row - and you want your code to not time out on that testcase!

4 How It All Works

This program is going to work differently than previous programs. Instead of compiling one program and then running it against many testcases, in this case you will build many programs, and each program is a testcase.

Multiple compilation makes this surprisingly easy. You must first build the object file for the library that you've written - this will be a file named `arrFlex.o`. You will also build an object file for each testcase; for instance, a testcase named `test_arrFlex_10_foobar.c` would be built into an object file named `test_arrFlex_10_foobar.o` . You then link each testcase with the library; each testcase becomes an executable.

So how does the example executable work in this project? I have provided an example **object file** instead of an executable. So to test your code, you will also link the example object file with your testcase, to provide an example executable.

In summary, your code must:

- Compile your library into a (single) object file

  This **must** be named `arrFlex.o` - or else, our grading script won't know how to find it!

- For each testcase:

  – Build the testcase into an object file
  – Link the testcase object file to the example object file, to produce an example executable.
  – Link the testcase object file to your library object file, to produce your testcase executable.

4.1 Comparing Your Code to the Example

The library should **never** print out anything. Instead, all printing should be handled by the testcases. This means that your testcase will need to print out **a lot** - and it must print out things that allow you to find mismatches between the two libraries. Print out things like the size of the array and its elements - and print them out often. Also, make sure that your testcases check return codes.

Our grading script will compare the two libraries by comparing the outputs of the testcases in both cases - and we will write testcases which give lots of details to compare!

4.2 Testcase Writing Strategies

We have provided a single testcase for you. It doesn't run any of the functions, and it doesn't print anything - but it has code (which never runs) that calls the functions. Use this testcase first - just to make sure that you have implemented all of the functions. (If your code links, you have pretty much passed this testcase.)

But take note: do **not** try to implement all of the code before you do this! Instead, implement "stub functions" - that is, functions which have the right parameters, but don't do anything but return a dummy return code. In that way, you should be able to pass the first testcase before you've written any code.

After you pass the first testcase, start writing small testcases. Write testcases that are as small as possible - just test one or two functions. (Your first testcase will probably just test `new()` and `free()`, since those will be needed in all the rest.)

Eventually, you will probably want to write a large, unified testcase, which tests everything - but make it your last testcase, after you have passed all of the rest of the testcases.

5 Testcase Format

Every testcase is a C file. The name of the file must be `test_arrayFlex_*.c` Unlike Projects 7/8, you will **not** be turning in testcase directories; just turn in single files.

Every testcase must include `arrFlex.h`, and - of course - any standard library headers that are required.

6 Limitations

Remember the following rules about your code:

- You **must change** the definition of `struct ArrFlex` in `arrFlex.h` .
- You **must not change** any of the function prototypes in `arrFlex.h` . (If you do, our build script will almost certainly not be able to build your code.)
- Even if you haven't implemented it, we **strongly encourage** you to implement a (dummy) version of every function. (If you don't do this, then you will not be able to link with many of the testcases.)

7 Makefile and `gcov`

You must turn in a Makefile. It must automatically build the following items:

- `arrFlex.o`

  Make sure that you include the options for `gcov`, since the testcases built by this Makefile will be used for `gcov` testing.

- An object file for every testcase.

- Each testcase executable.

  For a testcase named `test_arrFlex_foobar.c`, the executable must be named `test_arrFlex_foobar` .

It does **not** need to build any of the example executables - our grading script will do that.

7.1 `gcov`

Your Makefile must build your library - and each testcase executable - with the options for `gcov`. Our grading script will run the testcases that you have provided - which is why your executables must have names that match the testcase C files.

You are only required to cover the code **in your library.** (It's perfectly OK - even normal - to not cover your testcases very well!) The TAs will be checking the file `arrFlex.c.gcov` only.

8 What You Must Turn In

Turn in the following directory structure using the assignment name `cs352_f16_proj09`:

```
arrayFlex/
    Makefile
    arrFlex.c
    arrFlex.h
    test_arrFlex_*.c
```

8.1 Grading Scheme

This project has some grade items which apply to the entire project (not to individual testcases), so our grading scheme has gotten more complex:

- 10% of your grade will come from your Makefile
- 10% of your grade will come from `gcov` coverage
- 80% of your grade will come from testing your code, comparing it to the example program.

Of course, just as before, we also have a set of penalties which may be applied - these subtract from your overall score:

- -20% - Changing any of the function prototypes in `arrFlex.h`
- -10% - Poor style, indentation, variable names
- -10% - No file header, or missing header comments for any function (other than `main()`)
- -10% - Missing checks of returns codes from standard library functions (such as, but not limited to, `malloc()`)
- -20% - Any use of `scanf()` family `%s` specifier without limiting the number of characters read.

8.2 Testcase Grading

We have decided to subdivide the score from each testcase:

- You must exactly match `stdout` to get **ANY** credit for a testcase.
- You will lose one-quarter of the credit for the testcase if the exit status or `stderr` do not match the example executable.
- You will lose one-quarter of the credit for the testcase if `valgrind` reports any errors. (This includes any memory leaks.)

As always, you lose half of your testcase points if your code does not compile cleanly (that is, without warnings using `-Wall`).

8.3 Testcase Selection

As in Project 7, we will use the testcases that you submit to perform `gcov` testing on your program - and a selection of testcases from the students (plus a few from the instructors) for checking to see if your program works **correctly.**

9 Standard Requirements

- Your C code should adhere to the coding standards for this class:
  `http://www.cs.arizona.edu/classes/cs352/fall16/DOCS/coding-standards.html`
- Your programs should indicate whether or not they executed without any problems via their **exit status** - that is, the value returned by `main()`. If you return 0, that means "Normal, no problems." Any other value means that an error occurred.

  In this class, we will always use the value 1 to indicate error; however, outside this class, many programs use many different exit status values - to indicate differnt types of errors.

  In `bash`, you can check the exit status of any command (including your programs) by typing `echo $?` immediately after the program runs (don't run **any** commands in-between).

10 Grade Preview

48 hours before the project is due, we will run the automated grading script on whatever code has been turned in at that time; we'll email you the result. This will give you a chance to see if there are any issues that you've overlooked so far, which will cost you points.

Of course, this grading script will not include the full set of student testcases, which we will use in the grading at the end, but it will give you a reasonable idea of what sort of score you might earn later on.

We will only do this once for this project. If you want to take advantage of this opportunity, then make sure that you have working code (which compiles!) turned in by 48 hours before the due date.