CS 352 (Fall 17): System Programming and UNIX

Project #9

Object Orientation in C

due at 5pm, Thu 9 Nov 2017

1 Background

In Project 7, you worked with Linked Lists, and I asked you to start thinking of C structs like simplified Java classes. In fact, that is exactly what they are. C was developed in the mid 70s; ten years later, in 1985, Bjarne Stroustrup invented C++, which was C, but with some new features added to structs, to make them "classes." There were several new features, but some of the most important were:

- Member functions (what Java would later call "methods")
- Encapsulation (`public`, `private`, and others)
- Inheritance (parent and child classes)
- Virtual dispatch and polymorphism

Of course, Stroustrup didn't invent "object oriented programming" - that had been around for decades. But C++ was noteworthy because it combined object orientation with a high-performance, low-level language like C.[1] Ten years after C++ came out, Java was developed - which was basically "C++, but with some of the more confusing features simplified."

But C++ doesn't add any **new** capabilities to C; it just makes old capabilities easier to use. For instance, people have been declaring certain interfaces as "public" or "private" almost since the beginning - C++ simply added the ability to **enforce,** in code, those design decisions.

1.1 Overview of this Project

In this project, we'll be exploring how member functions are implemented. You will be implementing, in C, something which works pretty much like a "class" - you will have a whole variety of different functions, which all take the same pointer type as the first parameter. This pointer is (more or less) equivalent to the `this` pointer from C++ or Java. The only exception is a special function - `alloc()` - which is, in essence, the "constructor" for that class. Instead of taking the pointer as a parameter, it calls `malloc()` to create a new object, and returns it.

While you can do this in C, you'll find that the syntax is a little clunky. For instance, C doesn't allow you to have two functions with the same name -

[1] Some OOP purists think that this was a **terrible** idea. But whether it was good or bad - it certainly was **noteworthy.**

so we will add a long prefix (the "name of the class") to each function name. Second, you can't use the "dot" operator to call a "member function." Instead, you need to call the function, and pass the object as a parameter.

However, I hope that you will see, from this project, a rather direct connection between C++/Java member functions, and the ordinary functions that you've been learning about in C.

1.2 Two Classes

In this project, you will be implementing two "classes." (Note that nobody uses the term "class" when writing C code. But I'll do it, in this project, in order to emphasize what I'm trying to teach you.)

The first class will be a **doubly-linked list** which stores integers. (If you haven't seen doubly-linked lists, I'll review them below.) As we've done in Project 7, this class will not have a "wrapper" class - that is, we will only declare a class for the **individual nodes.** In this model, an "empty list" is represented by a `NULL` pointer. This also means that a pointer to an object of this class can be thought of in two ways: as a pointer to a single node, or as a pointer to a list. Either one might be correct, based on context.

For this first class, I've provided a header file `dblListInt.h` which declares the entire struct, and also every one of the member functions that you need to implement.

The second linked list type is a list that contains strings. This type is actually made up of two different "classes" - one that represents the entire list (a "wrapper" type), and one that represents individual nodes. With this type, you first **allocate the list** - which creates an object - but the list is empty. You then, as a **second step,** add values to the list.

I have provided a header for this type as well; it gives the names of both classes - the wrapper and the node - and also all of the member functions that you must implement. But **it does not give the internal fields of the two structs.** Instead, **you** will define the details of the struct, inside your own C file. (Read the comments in `encapsulatedListStr.h` to see how this works.)

1.3 Loosened Makefile Restrictions

For this project, I've provided a starting `Makefile` for you. It doesn't include all of the rules that you will need, but it has some of them. Study it - I'm making use of a couple of new features of `make` that you haven't seen before. Then upgrade it, to fill in the missing features.

I will be doing a lot less rigorous testing of the `Makefile` on this project. Your `Makefile` must simply include the necessary rules (along with dependencies, of course), to build the following items:

- A compiled program for each testcase, matching the testcase name.

  (You don't have to write a Makefile that automatically finds all of the testcases - just write rules for the testcases you have.)

- `dblListInt.o` and `encapsulatedListStr.o`, both compiled from the similarly-named C files (which you will write)

Each of these need rules so that we can build them individually - but you should also provide an `all` target (or something like that), so that, if we type `make`, it will build **all of these.**

NOTE: I will almost certainly have some additional testcases, which I have not released to you. I will provide a supplementary Makefile, which will compile those programs. But it will only work if **you have provided the `.o` files required above.**

1.4 New Testcase Strategy

For this project, I needed to change the testcase strategy. This project has two types of testcases: ones that produce output and are compared against an `.out` file, and various types of inputs that test a sorting program.

The first type of testcases are given as C files. Each one is named

```
test_*.c
```

and has a matching file ending in `.out`. These need to be compiled into `.o` files, and then linked with either `dblListInt.o` or `encapsulatedListStr.o` (depending on which they are going to test). The programs each create some objects, call some of the methods, and print out results; the results must exactly match the `.out` file.

The second type of testcases are text files, which are input to a sorting program (`mergeSort`). I have provided `mergeSort.c`; you must link this with your file, `encapsulatedListStr.c`. If your list is working properly, then this program will read all of the words out of an input file[2], sort them, and print them out. It uses a classic algorithm known as Merge Sort - and if your code is efficient (for instance, using $O(n)$ algorithms instead of $O(n^2)$) then it will be able to sort thousands of words in a fraction of a second.

The second type of testcases don't have any assorted `.out` files; instead, I will use the standard UNIX utility `xargs` to break the words into separate lines, and `sort` to sort them.

1.4.1 Grading Testcases

As before, each testcase will be broken into quarters. Half of the testcase score comes from matching `stdout` (and you can't gain any of the other points if you don't match this). One-quarter comes from `stderr` and the exit status; your program must print **nothing** to `stderr`, and exit with 0. One-quarter comes from a `valgrind` check. (As before, you must free **everything** when the program ends.)

The points will be broken down as follows:

[2] It uses `scanf("%255s")`, so it should be able to read almost any word.

- 40% from the C programs (each testcase is a prorgram)
- 20% for the Merge Sort testcase (each testcase is an input file to `mergeSort`)
- 40% hand grading

1.5 Turning In Your Code

Turn in your code using D2L. Turn in the following files:

```
dblListInt.c
encapsulatedListStr.c

Makefile
```

Please submit them as separate files; do not zip them up into a single zip file or tarball.

UPDATE: Due to limitations in D2L, please rename your `Makefile` to `Makefile.txt` in order to turn it in. I'll change it back when I collect the files.

2 What is a Doubly-Linked List?

A doubly-linked list is a linked list where every node includes both `next` and `prev` pointers in every node. This makes it easy to both move forwards (from the head to the tail), but also to move backwards (from the tail to the head). Sometimes, we will store pointers to both the head and tail of the list; sometimes, we only store one of them.

To insert a node into a doubly-linked list, we potentially have to update **four** pointers instead of just two. Imagine that we start with the following two nodes:

```
A - C
```

In this arrangement, we have two non-`NULL` pointers; `A->next` points at `C`, and `C->prev` points at `A`. If we want to insert a new node, `B`, between the two, like this:

```
A - B - C
```

we have to update four different pointers:

- `A->next = B`
- `C->prev = B`
- `B->prev = A`
- `B->next = C`

Likewise, if we remove a node from the list, we must update four pointers; two in the nodes which are left behind, and two in the node that is removed. (To keep things sane, we typically will set the `next,prev` to `NULL` in any node that is removed.)

Of course, when we do these modifications, we have a couple of important implementation details. First, we must change the pointers in a carefully chosen order, so that we don't lose track of any of the objects. For example, in the example below, if we have a pointer to element `A`, then probably the only way that we can find `C` is through `A->next`; thus, we should not change `A->next` until we've copied that pointer somewhere else. (We can use a temporary variable, or we can copy it into `B`; either are good options.)

Likewise, we always have to consider the possibility that the `next` and/or `prev` pointers might be `NULL` (thus indicating the ends of the list). If we aren't careful, we may segfault when (for instance) we try to update `C->prev`, but find that `C==NULL`.

Finally, it is important to make sure that the `next,prev` **pointers are always in sync!** That is, for any given node `X`, if `X->next != NULL`, then it **must** be true that `X->next->prev == X`. (And vice-versa.)

3 Merge Sort

Merge Sort is a sorting algorithm that breaks a set of values into chunks, sorts the small chunks, and then "merges" the various chunks together. In this project, I've provided the overall code, and you provide one of the key functions for it.

First, I read a bunch of strings into a list. I then run a recursive function; at each level, I divide the list in half, and then recurse into both sides. (The base case is when the list has gotten down to a single element.) Since the purpose of this function is to sort the list, when the recursive calls return, we'll have two sorted lists.

You will implement the last step of the algorithm: merging two lists. The function `encList_Str__merge()` takes two lists as parameters, which it assumes are both sorted. It will merge the two lists into one, producing a single sorted list.

It is important that your implementation for this function run in $O(n)$ time. However, this is fairly straightforward: a merge function basically runs the following loop, over and over:

- Compare the smallest element in one list, to the smallest in the other
- Move the smaller of two into the "output"

In other words, you need three lists:

- Input 1
- Input 2

- Output

(In our function, the Output list is the same object as Input 1 - but the basic idea doesn't change.) On each iteration of your loop, you will move exactly one list node from one of the inputs to the output.

The trick with this function will be handling edge cases. Remember to carefully consider what should happen in all these interesting cases:

- You've been moving nodes into the output list - and at last, one of the lists has become empty. (This actually happens **every** time, except in very special cases.)
- One (or both) of the input lists are empty
- The first element from input 1 comes before the first element of input 2 (and vice-versa)

4 Class 1 - Doubly-Linked Int List

You must implement the "class" `DblList_Int`. I have provided the header file for this class, which includes the struct definition, the necessary typedefs, and and prototypes for all of the "methods" of this class.

Each method has extensive documentation - read the spec in the header. Write your implementation for this class in a file named `dblListInt.c`.

Do **not** modify the header. (We'll replace it with the standard version if you do.)

4.1 Testing

I have provided some basic testcases, but **they do not test all of the functions** - also, they don't check for all possible errors. Write some additional testcases of your own. While you won't be graded on the quality of your testcases, you definitely **will** be checked, when we grade your program, with some extra testcases that I've written.

And, as before, I **encourage you to share you testcases on Piazza.**

5 Class 2 - Encapsulated String List

You must implement two "classes" which make up the encapsulated list of strings. By "encapsulated," I mean that the **implementation is hidden from the users.** So, the header file `encapsulatedListStr.h` provides the **names** of the two classes (one for the wrapper class, one for the nodes), and typedefs for each - but only your code, in `encapsulatedListStr.c` , will know how many fields there are, and what they are.

5.1 Testcases

The primary way of testing this class is using the `mergeSort` program. This will test a lot of the functionality - but it doesn't test **all** of it. I **encourage you to write some string list testcases of your own.**

5.2 Duplicated Strings

Your list must store pointers to strings, so that it can store any length of string. But any time that we have a pointer to a string, we have a question of ownership: who is responsible for freeing the memory for that string? Is the string a part of the list - or does the list just hold the pointer, and somebody else will free it? Similarly, can the list assume that the string will always be available, and unchanging? Or might the buffer that holds the string change, or get freed?

For this reason, the `addHead()` and `addTail()` functions have a `dup` parameter. If `dup==0`, then your code should simply store the pointer into your list. Assume that the string will never change, and that there is some other code, somewhere, which will free it **after** your list is cleaned up. However, if `dup==1`, then your code **must** `malloc()` a new buffer, copy the string into it, and save that.

This means, of course, that in addition to the pointer itself, you also need a field in your list node which indicates whether the string should be freed when the list node is freed. Make this a **per-node** field, since it's legal for us to create a single list which mixed both duplicated, and non-duplicated, strings.

NOTE: The `mergeSort` program will only test the `dup==1` case. This means that, in order to test the `dup==0` case, you will **have** to write some testcases of your own!

5.3 What Type of List?

What type of list should you implement? That's up to you. I'm guessing that many of you will use a doubly-linked list (just like the integer list), but you're welcome to choose something else if you want.

Just make sure it's a linked list. No arrays here!

6 Standard Requirements

- Your C code should adhere to the coding standards for this class: `http://lecturer-russ.appspot.com/classes/cs352/fall17/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. (Sometimes the error is serious - meaning "the

operation cannot be performed." Sometimes, it's more minor, such as "two files are different" or "minor issues happened.")

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).

7 `malloc()` Failures

As always, check the return code on `malloc()`, and have some sort of error handling routine for it. However, as we've discussed, it won't be practical to test this code.

8 `free()` all `malloc()`s

Starting with Project 5, you must now `free()` every buffer that you `malloc()`, before your program terminates. If you don't, then `valgrind` will report errors - and you will lose partial credit on your testcase.

This is, of course, a good habit to have! Always keep track of all memory that you have allocated with `malloc()` - and always be responsible to free it.

However, in the Real World, people often ignore the need to free memory, if the program is about to die. As I've mentioned in class, when your process dies, the OS will automatically free **all** of your memory - so it doesn't really matter whether you `free()`d everything or not.

But since we want you to practice `free()`ing your memory, we will require that you `free()` every buffer that you `malloc()`ed - even if you don't free it until right at the **end** of your program. (This is, probably, exactly what you'll do in Project 5!)

9 Special Note: `scanf()` and `%s`

Several programs in this class will require you to read strings from `stdin` using `scanf()`. **This can be dangerous, if you read more data than your buffer allows** - since it is possible to read right off the end of the array, and overwrite other memory.

To solve this, `scanf()` allows you to limit the number of characters that you read with the `%s` specifier. **You must always use this feature.** Remember that `scanf()` will also write out a null terminator - so this number **must** be less than the size of your buffer, like this:

```
char buf[128];
int rc = scanf("%127s", buf);
```