Memory management: talloc

In the last assignment, you built a linked list, and wrote code that hopefully cleaned up the list appropriately. Perhaps you have been missing the convenience of using a language with a garbage collection system that spares you from having to remember to clean individual things up. For this assignment, we're going to build an exceedingly dumb but effective garbage collector. This garbage collector is so inefficient that this may bother some of you; if so, consider improving the garbage collector to be an optional extension that you can think about when the project is complete.

This is a "pair" assignment, which means that if you are working on a team with someone else, you and your partner should do your best to engage in the pair programming model. At any point in time, one of you is "driving," i.e. actually using the keyboard and mouse. The other is actively engaged following along, preventing bugs, and providing ideas.

You should make sure that over the course of an assignment that you spend roughly the same amount of time each "driving." I will also ask you to turn in a form rating the work that your partner does. My recommendation is to take turns approximately every 15 minutes or so. Set a timer to help you remember.

1. Get started

Download this 02-talloc.zip file, and extract the zip file to your ProgrammingLanguages folder that you've been working in.

2. The idea

You'll be creating your own replacement for `malloc`, which we'll call `talloc` (for "track malloc"). For a user, `talloc` seems to work just like `malloc`, in that it allocates memory and returns a pointer to it. Inside your code for `talloc`, you'll need to call `malloc` to do exactly that. Additionally, `talloc`

should store the pointer to that memory in a linked list that we'll call the "active list" for purposes of discussion. Every time `talloc` is called, another pointer to memory gets added to that active list.

You'll then also create a function called `tfree`, which will free up all memory associated with pointers accumulated do to calls to `talloc`. Calling `tfree` at arbitrary points in your program would be a complete disaster, as it would free up memory that you may still be using. The idea is that we will be using `talloc` as a replacement for `malloc`, and then calling `tfree` at the very end of our main function. You'll then be able to program with the illusion of using a garbage collector, except that the garbage collector never actually kicks in until the program is about to end. (This was actually an option for the leJOS system for programming LEGO Mindstorms in Java; if interested, ask me for further info as to why a real system would implement such a crazy-seeming idea.)

You'll also write the function `texit`, which is a simple replacement for the built-in function `exit`. `texit` calls `exit`, but calls `tfree` first.

Finally, you'll then modify your linked list from the previous assignment. The function `cleanup` that you wrote will be eliminated, as it is no longer necessary. You should also modify `reverse` so that it no longer duplicates data between the two linked lists. When you reverse a list, that should return a new list with a new set of `CONS_TYPE` nodes, but the actual data in that list should not be copied from the old list to the new. This would be a disaster to try to clean up manually, but `tfree` will handle it easily. This change will make some later aspects of the project much easier. Your linked list code should now exclusively use `talloc`, and should not use `malloc` at all.

3. Storing the active list

One issue you'll need to think through is where the variable for the head of the active list should be. In an object-oriented language, this would likely be a private static variable in a memory management class. Oops. You can't

make the active list head a local variable in `talloc`, because `tfree` wouldn't be able to see it. We could make it a parameter to `talloc` and `tfree`, but then the programmer using `talloc` has to keep track of this, and could conceivably have multiple active lists, which sounds ugly. This is an occasion where global variable makes sense, and so you should use one. A global variable in C is declared outside of any functions. Typically, it is placed near the top of your file, underneath the include statements.

There's one bit of circular logic you've got to untangle. `talloc` needs to store a pointer (returned by `malloc`) onto a linked list. Your linked list code, in turn, uses `talloc`. Rather than trying to make this work in some complex mutually dependent structure, my recommendation is to break the circularity. In your `talloc` code, the single linked list that you use to store allocated pointers should be a linked list generated via `malloc`, instead of `talloc`. That means you'll need to duplicate some of your linked list code. Duplicated code is generally to be avoided, but avoiding this circular nightmare is worth it.

4. Some specifics

After you clone your repository, you should be able to see that you get the following starting files, among others:

- `schemeval.h`: this defines the Object structure again
- `linkedlist.h`: this is a modification form the previous assignment that removes the function `cleanup`, and also changes the documentation on `reverse` to indicate that data is not to be copied.
- `talloc.h`: this defines the functions that you'll need to write from scratch for this assignment.
- `main.c`: this is a tester function.
- `justfile`: contains instructions for the command `just`, which will compile and test your code
- `test-e` and `test-m`: usual
- `test_utilities.py`: helper utilities used by `test-e` and `test-m`

The missing files here are `linkedlist.c` and `talloc.c`. You should create `talloc.c` from scratch yourself. For `linkedlist.c`, you should copy in code from your previous assignment, and then modify it accordingly. To compile your code, issue the command `just build` at the command prompt. This will follow the instructions in `justfile` for building your project in order to produce an executable called `linkedlist`. At first, it won't build at all because your `talloc.c` and `linkedlist.c` files aren't there. To get started, copy in `linkedlist.c` (remove the cleanup function), and for now, within `talloc.c` just create every function that you need with no code inside it so that you can get everything to build. Once you have done that, you can begin implementing your functions, and testing appropriately.

The tester code creates an executable that you can run by typing `./linkedlist`. The easiest way to run the tests is to use `python3 test-m` and `python3 test-e`, as usual, which will automatically compile all of your code and run the `./linkedlist` executable for you.

Your code should have no memory errors when running on any input (correct or incorrect) using `valgrind`. The testing scripts will automatically run `valgrind` on your code, and show you if there are memory errors.

5. Preparing for celebrations of knowledge

The celebration of knowledge will ask you to write on paper C code to solve problems similar to the above. It might have you write the code from scratch; it might have you fill in some blanks in partially written code; or it might use some other method for having you demonstrate that you can write code. To prepare, you should practice writing C code on paper that can solve any of the above functions, or others like it.

If you're looking for more examples to practice: write a function that that can report a count, in bytes, of how much memory is being used in total by this talloc technique. It should include both all of the memory that the user

asked for when calling talloc, but also all of the additional overhead in Object structs that talloc creates for assembling a linked list. Here is a signature for that function. You can add it to your `talloc.h` file:

```
int tallocMemoryCount();
```

6. Grading

You will receive a grade of 3 if:

- Your code passes the M tests.
- Your program is written to work in general, and not to only work for the specific tests that we have provided.

You will receive a grade of 4 if you satisfy the above requirements for a 3, and…

- Your code passes the E tests.
- You have a comment before each function describing what the function does (its input/output behavior, not a play-by-play of "first checks… then makes a recursive call…").
- Your code is well-structured and not more complex than needed.

7. What to submit

Follow the instructions for the last assignment: submitting should be the same.