THE UNIVERSITY OF SYDNEY

COMP2017 9017 Assignment 2

Due: 23:59 28 March 2024

This assignment is worth 10% of your final assessment

Task Description

Your task is to create create a multi-type linked list data structure and a program that interacts with it. Your assignment is broken into three tasks that must be completed in order.

- The first part is the basic command syntax of the linked lists, creation, removal, viewing etc.
- The second part is modifying the lists in place, through insertion and deletion of elements.
- The third part is allowing lists to refer to each other in a nested pattern.

It is also recommended to read through the specification carefully. You should ensure that you create test cases that cover a range of possible inputs *before beginning to code*.[1] Make sure you identify and test for edge cases.

Implementation Details

All commands are read through standard in, and all output is given through standard out.

Part 1: Basic Commands

For Part 1, your linked lists must support elements of the following types:

- `int`
- `float`
- `char`
- `string`

Your program should take in commands from `stdin` that will create and manage these multi-type linked lists. The basic commands are:

[1]Even if you do it on pencil and paper, work through some examples to be sure you understand.

- `NEW <number of elements>` - create a new list
- `VIEW <list index>` - view a specific list by its index
- `TYPE <list index>` - view a specific list by its index, printing out the types of each element.
- `VIEW ALL` - print the number of lists and each list in order of creation
- `REMOVE <list index>` - remove a list

The index for each list should be 1 higher than the last created list's index, starting from 0 for the first list, regardless of how many lists have been removed.

Basic Examples

In the following examples, "`> `" denotes the following line as input. Your program must not print it to `stdout` or read it from `stdin`. It is included in the formatting only as an indicator, to differentiate the input from the output of the program.

Command keywords are delimited by exactly one (1) space character. Be sure to replicate the formatting of these examples *exactly*.

The NEW Command

This command takes in a number as input for the initial size of the list. It then reads in an initial value for each element to initialise the list. 0 is a valid size, negative numbers are not. Lists are labelled starting at 0.

```
> NEW 5
> hello
> 1
> 2
> 3.14
> a
List 0: hello -> 1 -> 2 -> 3.14 -> a
```

The VIEW Command

This command prints out the contents of the list at the given index.

```
> VIEW 4
hello -> 1 -> 2 -> 3.14 -> a
```

The TYPE Command

This command prints out the types of each element at the given list.

```
> TYPE 4
string -> int -> int -> float -> char
```

The VIEW ALL Command

This command prints out the current set of lists in index-increasing order.

```
> VIEW ALL
Number of lists: 3
List 0
List 3
List 4
```

The REMOVE Command

This command deletes a list, and prints out the current set of lists again in index-increasing order.

```
> REMOVE 3
List 3 has been removed.

Number of lists: 2
List 0
List 4
```

Invalid Commands

If a command is invalid in some way, print `INVALID COMMAND: <command used>`. For example, when there is no `List 4`:

```
> REMOVE 4
INVALID COMMAND: REMOVE
```

If a command cannot be identified, use `INPUT`. For example:

```
> abracadabra
INVALID COMMAND: INPUT
```

It is up to you to find and prepare for edge cases.

Type Rules and Exceptions

There can be some ambiguity in certain cases for what a given input's type is. The order for type checking is as follows:

- `integer`
- `float`
- `char`
- `string`

Requirements for types are as follows:

- `int` can be negative, positive or zero (tests will also not exceed the maximum and minimum value for an int type).
- `float` is the same except it will always have a decimal point.[2]
- `float` should also be printed to 2 decimal places, though they can be read in to any precision.
- `char` is any printable character in `ascii` as long as it is singular.
- `string` covers all other cases.
- Empty lines in list creation should be considered as `string`.
- `string` can start with leading and trailing whitespace characters.
- Lines containing an `int` or `float` can have leading and trailing whitespaces. These will be interpreted as numbers. (Note that this is the default behaviour of `scanf`).
- All inputs will have a maximum total line length of `128` bytes.

```
> NEW 4
> 1.0
>
>  baguette
> 5
List 4: 1.00 ->  ->  baguette -> 5
```

Note the empty line that was interpreted as an empty string, as well as the extra space in " `baguette`". There will be no test cases that do not fit this description.

Note: Curly brackets `{}` are used in Part 3, and are considered invalid input if they appear in any form other than specified there. [3]

[2]The exception being scientific notation which should also be accepted, i.e. `2e-4 = 0.0002`. Note that `scanf` accepts this type of input by default.

[3]For completion of Parts 1 and 2, it is sufficient to raise an error whenever curly brackets are detected. Part 3 introduces a single exceptional use which is not an error.

```
> NEW 3
> 1.0
> {}
> baguette
INVALID COMMAND: NEW
> NEW 3
> 1.0
> {
> wordswords } words
INVALID COMMAND: NEW
```

Exiting the program

Upon `EOF`, the program should free all used dynamic memory, then exit.

Part 2: Dynamic Lists

In this part you are to implement two extra commands: `INSERT` and `DELETE`.

The INSERT Command

This command takes input of the form `INSERT <list id> <index> <value>`, and inserts the value at the given index of the given list. It should then print out the new list. For example:

```
> View 1
List 1: a -> b -> c -> d
> INSERT 1 0 Baguettes
List 1: Baguettes -> a -> b -> c -> d
```

Negative indices should insert from the end of the list. Indices outside the range are invalid:

```
> View 1
List 1: a -> b -> c -> d
> INSERT 1 -1 Baguettes
List 1: a -> b -> c -> d -> Baguettes
> INSERT 1 97 Croissants
INVALID COMMAND: INSERT
```

The DELETE COMMAND

This command takes input of the form `DELETE <list id> <index>` and removes the given index from the list. It should then print out the new list. The same conditions on indices apply. For example:

```
> View 1
List 1: a -> b -> c -> d
> DELETE 1 0
List 1: b -> c -> d
> DELETE 1 -1
List 1: b -> c
> DELETE 1 4
INVALID COMMAND: DELETE
```

Part 3: Nested Lists

For this section, you are to modify your previous code to accept a new type: other lists. This is to a maximum depth of one. This means every list is either a simple list (contains only regular types), or a nested list (contains regular types and simple lists). Nested lists cannot contain other nested lists.

Nested lists contain only references to simple list(s). Thus, changes to the simple list should also be reflected in the nested list.

To insert a simple list into a nested list, it should be specified with curly brackets. When nested lists are printed, they should be labelled as `Nested`, like so:

```
> View 1
List 1: a -> b -> c -> d
> NEW 3
> first
> {1}
> last
Nested 2: first -> {List 1} -> last
> VIEW ALL
Number of lists: 2
List 1
Nested 2
```

Any command that refers to a non-existent list, or would result in any list having depth greater than 1 should give an `INVALID COMMAND`:

```
> VIEW 0
List 0: a -> b -> c -> d
> NEW 2
> first
> last
List 1: first -> last
> INSERT 1 1 {0}
Nested 1: first -> {List 0} -> last
> NEW 1
> {1}
INVALID COMMAND: NEW
```

If all references are deleted from a nested list with `DELETE`, then it becomes a simple list. The `TYPE` command should print `reference` as the type of any references to other lists.

```
> VIEW 0
List 0: a -> b -> c -> d
> VIEW 1
Nested 1: first -> {List 0} -> last
> TYPE 1
Nested 1: string -> reference -> string
```

Removal of a simple list while it is referenced by any other list should give an
`INVALID COMMAND: REMOVE`.

The VIEW-NESTED Command

This command can print any list, but when it prints a nested list, it will also print its sub-lists, contained in curly brackets. Like so:

```
> VIEW 1
List 1: a -> b -> c -> d
> VIEW 2
Nested 2: first -> {List 1} -> last
> VIEW-NESTED 1
List 1: a -> b -> c -> d
> VIEW-NESTED 2
Nested 2: first -> {a -> b -> c -> d} -> last
```

Restrictions

To successfully complete this assignment you *must*:

- Use dynamic memory.
- Use linked list structures,[4] they must be your own implementation.[5]
- Free all dynamic memory that is used.

Any submission breaking these restrictions will receive a deduction of up to 5 marks *per breach*.

[4]There are no requirements on what kind of data structure is used to keep track of all the created lists, as long as it is dynamic and the lists themselves are linked lists of any kind.

[5]This means you mustn't cite or borrow code from any other source. Design and implement the structure yourself.

Working on Your Assignment

You are encouraged to submit your assignment on Ed while you are in the process of completing it. By submitting you will obtain some feedback of your progress on the sample test cases provided.

If you have any questions about C functions, then refer to the corresponding man pages. You can and should ask questions about this assignment on Ed. As with any assignment, make sure that your work is your own, and that you do not share your code or solutions with other students.

Getting Started

The most important factor for success is your choice of data structures. We recommend:

1. Read the specification and write test cases.
2. Design your data structures.
3. Part 1 Commands
4. Part 2 Commands
5. Part 3 Commands

Writing even a few simple test cases of your own will ensure you understand the *details* of what you've been asked to do. Even on pen and paper, this is incredibly beneficial.

Data structure design will have the largest impact on the quality of your code (both in terms of style and correctness). The scaffold has a few suggestions on the function prototypes you should use, but does not cover every instance. Be considerate of how your linked-list will function.

Debugging and Avoiding Leaks

It is recommended that you use tools such as `gdb`, `valgrind` and `ASAN`
(the `-fsanitize=address,leaks` compilation flags).

These will assist in finding basic errors (`gdb`) and monitoring memory leaks (`valgrind`, `ASAN`). However, they cannot automatically prevent memory errors and leaks. You should be conscious of where leaks may occur and verify them yourself.[6]

Note: Mac users may need to use a full virtual machine (VM) to make use of these tools. We have had reports of both `valgrind` and `ASAN` being unusable on Macs. Please refer to Ed for more details.

Compilation and Testing

Your program should be compiled by the default rule, which is the first defined rule of the `Makefile`. You should name this make rule `build`. After compilation, your program should be a single binary file called `mtll` which is used to run your program.

[6]This is helped by having a plan for what data structures you will use, and how your code will manage them.

```
# compile the program
make
# alternatively
# make build

# run the program
./mtll
```

You should implement your program in multiple C source and header files. This is required for full style marks.[7] They must all compile together into one single binary when the program is built.

You should also do your own testing. If you need to compile/create your tests before running them, please implement a make rule called `tests` for this and then a separate one for running your tests. Please also store your tests in the `tests` directory provided.

After the assignment is released, a small number of test files will be made available. Correctness tests will be provided to ensure that your code can execute fundamental examples. These tests will not be the complete set of tests run against your code.

Any attempt to deceive the marking system (such as hard coding test cases) will receive a zero.

```
# compile the tests if necessary
make tests
# run the tests
./mtll run_tests
```

Submission

Submissions for this assignment will be through `git`.

The general process of writing and submitting is the same:

```
git add <files>
git commit -m "fix memory leak in function x"
git push
```

If you have any questions about git usage, feel free to check the Git Lesson, the relevant manual pages, or ask on Ed.

Do **NOT** push binaries, including executables and object files. Either add your source files manually, or create a `.gitignore` file that includes the names of all your binaries.

None of your git commits should contain a binary file. Any submission breaking this restriction will receive a deduction of 1 mark *per commit*.

[7] It is recommended to break up your code into suitable sections, i.e. the main source file handles input, a separate source file for list management functions, etc.

Marking Details

The assignment is worth 10% of your final grade. This is marked out of 20, and breaks down as follows:

Marks	Item	Notes
3/20	Code Style	Manual marking
2/20	Test Case Coverage	Manual marking
6/20	Part 1 Correctness	Automatic tests
6/20	Part 2 Correctness	Automatic tests
3/20	Part 3 Correctness	Automatic tests

For style, refer to the style guide on ed: https://edstem.org/au/courses/14786/lessons/49533/slides/334765. You will also be marked based on the modularity and organisation of your code. For full marks, code should be organised in multiple source files, and use modular, task-specific functions. Organised data structures are essential here.

The scaffold provides some suggestions on what this looks like, but more will be necessary.

For test case coverage: This first mark is for coverage of basic command syntax. Full marks will be awarded for consideration of at least 3 edge cases. These may be for sections of the assignment you have not completed.[8]

The below table gives an indication of what is expected. Note this is not a set marking scheme, more a guideline. Refer to the above table for clearer ideas.

Expected Award	Level of Completion
Pass	Good code style, good test coverage and completion of Part 1. (Style misses one or two things: not modular, not multiple source files, etc.)
Credit	Excellent style and tests, most of Part 2. Good style and tests, completion of Part 2.
Distinction	Excellent style and tests, completion of Part 2
High Distinction	Excellent style and tests, completion of Part 3

Further Examples

All Basic Commands:

```
> NEW 3
> hello
> 2
> 1
List 0: hello -> 2 -> 1
> NEW 3
```

[8]Feel free to look for edge cases in Part 3 even if you do not manage to code it.

```
> 3.14
> world
> a
List 1: 3.14 -> world -> a
> VIEW ALL
Number of lists: 2
List 0
List 1
> VIEW 1
3.14 -> world -> a
> REMOVE 0
List 0 has been removed.

Number of lists: 1
List 1
> NEW 3
foo
bar
5
List 2: foo -> bar -> 5
> VIEW ALL
Number of lists: 2
List 1
List 2
```

Insert and Delete Commands:

```
> NEW 3
> hello
> 6
> 2e-2
List 0: hello -> 6 -> 0.02
> DELETE 0 2
List 0: hello -> 6
> INSERT 0 -2  goodbye
List 0: hello ->  goodbye -> 6
> DELETE 0 -1
List 0: hello ->  goodbye
> DELETE 0 -1
List 0: hello
> DELETE 0 -1
List 0:
> REMOVE 0
List 0 has been removed.
```

```
Number of lists: 0
```

Note the leading space in ' `goodbye`'.

Nested Lists

```
> NEW 3
> this
> is
> simple
List 0: this -> is -> simple
> NEW 4
> this
> is-nested
> {0}
> 4.0 h
Nested 1: this -> is-nested -> {List 0} -> 4.0 h
> VIEW-NESTED 1
Nested 1: this -> is-nested -> {this -> is -> simple} -> 4.0 h
> NEW 1
> other
List 2: other
> INSERT 0 2 {2}
INVALID COMMAND: INSERT
> REMOVE 0
INVALID COMMAND: REMOVE
> DELETE 1 2
List 1: this -> is-nested -> 4.0 h
> INSERT 0 2 {1}
Nested 0: this -> is -> {List 2} -> simple
> VIEW-NESTED 0
Nested 0: this -> is -> {this -> is-nested -> 4.0 h} -> simple
```

Note "`4.0 h`" is a `string`, not a `float`.

Academic Declaration

By submitting this assignment you declare the following: I declare that I have read and understood the University of Sydney Student Plagiarism: Coursework Policy and Procedure, and except where specifically acknowledged, the work contained in this assignment/project is my own work, and has not been copied from other sources or been previously submitted for award or assessment.

I understand that failure to comply with the Student Plagiarism: Coursework Policy and Procedure can lead to severe penalties as outlined under Chapter 8 of the University of Sydney By-Law 1999 (as amended). These penalties may be imposed in cases where any significant portion of my submitted work has been copied without proper acknowledgment from other sources, including published works, the Internet, existing programs, the work of other students, or work previously submitted for other awards or assessments.

I realise that I may be asked to identify those portions of the work contributed by me and required to demonstrate my knowledge of the relevant material by answering oral questions or by undertaking supplementary work, either written or in the laboratory, in order to arrive at the final assessment mark.

I acknowledge that the School of Computer Science, in assessing this assignment, may reproduce it entirely, may provide a copy to another member of faculty, and/or communicate a copy of this assignment to a plagiarism checking service or in-house computer program, and that a copy of the assignment may be maintained by the service or the School of Computer Science for the purpose of future plagiarism checking.

Changes

Any changes made to this document will be updated here.

15/03/2024-15:47 - resolve typos in Part 1 (The `TYPE` command, requirements for types), Part 3 snippet 2