COMP26120
Academic Session: 2019-20

Lab Exercise 4: Pointers and Linked Lists

Duration: 1 lab session

For this lab exercise you should do all your work on your *ex4* branch.

The initial version of this program should be a copy of your program from the "Arrays and Memory Management" lab exercise. Copy the starting files (*arrays.c* and *makefile*) from your *ex3* branch. You should rename *arrays.c* as *lists.c* and replace `#include 'arrays.h'` with `#include 'data.h'` in *lists.c*. To copy a file you can run something like

```
git checkout ex3
cp arrays.c tmp_file
git checkout ex4
mv tmp_file lists.c
git add lists.c
```

Learning Objectives

At the end of this lab you should be able to:

- Use structs, unions, pointers, malloc and free in C;
- Explain the use of pointers and memory management in C;
- Use a simple Recursive Data Type (RDT) - lists - in C;
- Use valgrind and clang sanitiser to help debug C programs.

Introduction

Remember, you can see The GNU C Programming Tutorial for more information.

union

Union is a special data type in C which stores different variables under a single name. It is declared and accessed in similar manner as *structs*, except for the fact that all of the members in *union* are stored in the same memory location. Only one member can be in a *union* at one time. These are different from structs and are used to be more memory efficient.

```c
typedef struct node
{
  char name[20];      // holds the actual node content, could be any data
  struct node *next; // points to the next list node
} node;
```

Figure 1: Basic form of a node in a linked list.

Linked list

Here is a short video on introduction to linked list.

A linked list is a linear collection of structures chained together by pointers, while their order is **not** relevant to their physical memory addresses.

Linked lists are easy to be sorted by redirecting pointers, and can be updated in length dynamically. Figure **??** gives the basic form of a node in a linked list. What value should go in next at the end of the list? What would happen if we did node−>next=node?

Doubly-linked list

Above is an example of a structure type in a **single linked list**, while there is also doubly-linked list in which each node contains pointers to the **previous** and to the **next** node. This allows easy traversal in both directions in linked lists.

Part 1

In this part you will implement a singly-linked list with different methods for insertion. You should produce a single C program *lists.c* for all steps of this part. **You should start with the appropriately renamed arrays.c from exercise 3. See instructions at the start of this document.**

Step A: Insert at start of list

Edit the program to replace the array *people* by a list:

- Remove all uses of *nextinsert* (as a variable or as a parameter) in your program. Modify the declaration of your *struct* to include a *next* pointer (i.e. to the same *struct* - this is why a list is a *Recursive* Data Structure).
- Change the declaration of *people* to be a pointer to your *struct* instead of an array, and initialise it to be empty (*NULL*). Your program should always keep this pointing to the start of your list. What happens if you forget to store where the start of the list is?
- Change *insert* so that:

  Its first parameter is a list (i.e. a pointer to your *struct*) instead of an array.
  It adds the *malloc*ed new person at the start of the list.
  It returns a pointer to the start of the new list as its result.

  Figure **??** gives further pseudocode for this.

- Change the call to *insert* so that the return result is put back into the pointer to the start of the list (i.e. *people*).
- Change the code that *free*s memory. Make sure you call *free* **after** you storing where the next element in the list is. Why might this matter?

```
create a new space for the new person
(check it succeeded)
set the data for the new person
set the new person's *next* link to point to the (start of the) current
    list
return the (start of the) new list i.e. a pointer to the new person
```

Figure 2: Pseudocode for insert

At this point you could try submitting your code and you should observe that some of the tests, but not all of them, will pass (as you haven't added all of the functionality yet). Make sure this is the case before continuing. Importantly, use *valgrind* to get ride of any invalid reads or memory leaks.

Step B: Insert at end of list

Rename your *insert* function to be *insert_ start*. Make an extra copy of it named *insert_ end*. Modify your program to call *insert_ start* and check that is still works as before.

Modify *insert_ end* to insert the new struct at the end of the list.
It still needs to return a pointer to the start of the list. This is only the same as a pointer to the struct that has just been inserted if the list was initially empty.
Use a loop to run down the list from the start to the end each time the function is called. What is the complexity of this function? Figure **??** gives pseudocode for *insert_ end*.

Note: To improve efficiency, we could keep a new pointer to the end of the list in *main* and pass it to *insert_ end*, so that each time there's no need to loop from head to tail. This improvement is optional and does not affect marking.

Update your program so that it takes an optional single command-line argument such that "insert_start" runs *insert_ start* and "insert_end" runs *insert_ end*. If no arguments are provided then the program should run *insert_ start*.

Your code should return a non-zero return code (and a suitable message to `stderr`) if unrecognised/unexpected command-line options are given (this applies from now on).

Submitting now should make more tests pass. Again, check tests and *valgrind* before proceeding.

Step C: Insert into sorted list

Create a new copy of *insert_ end*, called *insert_ sorted*, and extend your program to take a command-line argument "insert_sorted" to call this. Now modify *insert_ sorted* to put people into the list in (ascending) name order:

- You could just use *strcmp*, but instead write a similar function *compare_ people* which, given pointers to two *struct*s describing people, simply gets their name strings and returns the result of calling *strcmp* on those strings.

```
create a new space for the new person
(check it succeeded)
set the data for the new person
if the current list is empty (i.e. NULL)
  do the same as insert_start
otherwise
  use a loop to find the last item in the list (where *next* is NULL)
  set the "next" link of this item to point to the new node
  return the (start of the) list
```

Figure 3: Pseudocode for insert_end

```
create a new space for the new person
(check it succeeded)
set the data for the new person
if the list is empty or the new person is less than the first item
  do the same as insert_start
otherwise
  use a loop to find the last item that is smaller than the new person
  set the new person's *next* link to point to the next item in the list
  set the *next* link of this item to point to the new person
  return the (start of the) list
```

Figure 4: The algorithm for insert_sorted

- Modify *insert_sorted* so that, instead of always going to the end of the list, at each step it calls *compare_people* to decide whether to insert at this point in the list or to try the next person. Avoid calling *compare_people* when the list is empty, or when you reach the end of the list. If two people are equal then the person inserted first should appear first.

Figure **??** gives pseudocode for this. Again, submit and check.

Step D: Parameterising the sort order

Add an extra parameter to *insert_sorted*, that is itself a (pointer to a) function e.g. a function pointer. Call it *compare_people*. Edit your main to call *insert_sorted* with *compare_people* as the actual parameter. Check that your program still behaves as in the previous part.

Now, rename *compare_people* to be *compare_people_by_name*, and make another copy of it called *compare_people_by_age*. Again, edit your main to call *insert_sorted* with *compare_people_by_name* as the actual parameter and check that your program still behaves as in the previous part.

Edit *compare_people_by_age* to do what it says. Finally, extend your program to take a second command-line argument only when the first argument is “insert_sorted” that is either “name” or “age” (defaulting to “name” if a second option is not given). Add appropriate error-handling and ensure that the program calls *insert_sorted* with *compare_people_by_name* or *compare_people_by_age* appropriately.

Submit and check.

Part 2

In this part we will look at how a union data structure can be used to store different kinds of data in the same kind of variable and we will extend our list to be doubly-linked. **Note that most of the marks this week are for Part 1 but still try and get as far as you can with Part 2.**

Step A: Prototypes in a Header File

Copy *lists.c* to *slists.c* (standing for singly-linked lists) without removing *lists.c* (you need to keep it around for the part1 tests). Now rearrange your code so that your list functions are in *slists.c* and your main function (and its helper files e.g. `compare_people_by_name`) is in a file called *test.c*, you will need to add the people typedef and prototypes for your list functions into a new file called *slists.h* and include this in both *slists.c* and *test.c*.

If you add the following lines to your makefile

```
part2:
        $(CC) $(CFLAGS) slists.c test.c -o test
```

the command `make part2` will compile the necessary things. The resulting test program should behave in the same way as the lists program produced in part1.

If you have a duplicate symbol error then you are defining a symbol more than once – make sure you only #include ''data.h'' in *test.c*.

Before you continue make sure you understand what is going on here. Try just running `gcc test.c -o test` and you will get a *linking error*, what is this?

Step B: Union

In this step, you need to modify your program to know about 3 different kinds of people: students, staff, and neither:

- Use an *enum staff_or_student* to define *staff* and *student* and *neither*.
- Add a *staff_or_student* field to your *struct*.
- Note that *data.h* already contains arrays *person_type* and *info* similar to *names* and *ages*.
- Use a *union* variable named *extraInfo* to add the following extra information to your *struct*:

  for students - a string holding their programme-name.

  for staff - a string holding their room-number

  for other people - no further information.
- Modify the rest of your program (e.g. to insert the appropriate data from *person_type* and *info*)
- Print the resulting list using the appropriate function from *printing.h*. This is to ensure that your person nodes are printed in the format expected by the testing tool. You may need to change the variable names from people to make it work as we have made some assumptions about what you might use.

Step C: Doubly Linked List

In this step, you need to create *dlists.h* and *dlists.c* by copying *slists* and changing the current single linked list into a doubly linked list, that is, to add an extra pointer in structure, pointing to the **previous** node.

Now update *test.c* so that it uses doubly-linked lists if the complier flag `-DDOUBLE` is provided. You probably want to add a `part2d` case to your makefile to support this. Note that you can check this flag as follows.

```
#ifdef DOUBLE
  // this code is compiled if the flag is given
#else
  // otherwise this code is compiled
#endif
```

Finally, if a doubly-linked list is used the list should be printed *backwards* (otherwise it should still be printed forwards). For this you may wish to keep track of the last node in the list as well as the first.

We will stop there as the lab is already quite long but we have not properly explored the advantages of doubly-linked lists (other than being able to easily print the list backwards). One thing we could have asked you to do was implement two functions that first *find* a particular node in the list (e.g. with a given name) and then (separately) *delete* that node. Why would this be easier with a doubly-linked list than a singly-linked list? **The TA will probably ask you this question during marking.**

Marking Scheme

Note that this may be refined to introduce extra cases reflecting special cases if required.

Are Part1 steps A and B complete, correct, and understood?	
All of the output tests pass and the student can explain the code	(2)
Some tests fail and/or the explanation is insufficient	(1)
None of the tests pass	(0)

Are Part1 steps C and D complete, correct, and understood?	
All of the output tests pass and the student can explain the code	(2)
Some tests fail and/or the explanation is insufficient	(1)
None of the tests pass	(0)

Does Part1 have any memory errors?	
All of the memory tests pass e.g. there are no memory leaks or invalid reads/writes	(2)
Some but not all of the memory tests pass	(1)
None of the tests pass	(0)

Does Part1 handle incorrect input properly?	
All of the error-handling tests pass	(1)
Some but not all of the error-handling tests pass	(0.5)
None of the tests pass	(0)

Is Part 2 complete, correct, and understood?	
All steps have been completed correctly (tests pass) and the student understands how a union works and can describe the advantages of a doubly-linked list.	(2)
All steps have been completed but there are some small mistakes.	(1.75)
At least steps A and B have been completed correctly (tests pass).	(1.5)
A reasonable attempt has been made at this part but it is not finished and/or contains many mistakes	(1)
No submission for this part	(0)

Is the overall quality of all code good?	
Code is well formatted and commented with appropriate variable names	(1)
The code is acceptable but there are some issues	(0.5)
The code is not readable	(0)