COMP26120 Lab Exercise 5
Pointers and Lists

Duration: 1 session

Aims

To encourage you to find out more about the use of pointers and data structures, and about some of the memory management features of the C language.

Useful C on-line course themes etc.:

- Differences between Java and C
- Lots of information about Structs, Unions, Pointers and Linked Lists in Information Representation

Learning outcomes

On successful completion of this exercise, a student will:

- Know how to use structs, unions, pointers, malloc and free in C;
- Be aware of some fundamental differences between C and Java - the explicit use of pointers and of memory management;
- Know how to use a simple Recursive Data Type (RDT) - lists - in C;
- Know how to use valgrind to help debug C programs.

Summary

Use C to write a series of programs that repeatedly call:
`insert (structure, name, age, ...)`
to put information into 1-dimensional Linked Lists

- insert at start of linked list of struct
- insert at end of linked list
- insert in name order using compare function
- compare function passed as parameter (name order or age order)
- using enum and union with struct
- pointers to pointers

Deadlines: The unextended deadline is the end of your scheduled lab. If you need it, if you attend the lab you will get an automatic extension to the start of your next COMP26120 lab session (you must use submit to prove you finished in time and get it marked at the start of your next scheduled lab). You can also get an extension for good reason e.g. medical problems.

Description

For this lab exercise you should do all your work in your COMP26120/ex5 directory.

The initial version of this program should be a copy of your program from the "Arrays and Memory Management" lab exercise (it doesn't matter if you didn't complete all parts of that exercise). Copy the starting files (`arrays.c` and `makefile`) from your COMP26120/ex4 directory. Rename `arrays.c` as `lists.c`

You should write a single C program, `lists.c`, for all parts of this exercise. You should probably keep a working back-up each time you progress to a new part.

The parts become harder as you progress. If you are running out of time, don't try to complete all parts - instead get everything marked that you can by the deadline, and try to prepare better for the next lab exercise. (Make sure that you use submit to prove that you finished in time.)

You may want to refer back to the hints about debugging in lab exercise 4.

Part 1: Insert at start of list

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

- Get rid of 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.
- 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.
- 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 `frees` memory to call `free` **after** you have remembered where the next element in the list is. Why might this matter? (`valgrind` will help you check that you have got this right.)

Hints:
This is basically the same exercise as described in the on-line C course e.g. here Linked Lists Example and in the previous and following pages.
The algorithm for `insert` is:

```
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
```

Part 2: 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. Don't keep a pointer to the end of the list between calls, or anything like that.

Modify your program to call `insert_end`

Hint:
The algorithm for `insert_end` is:

```
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 i.e.
  set the new person's "next" link to point to the current list (i.e. NULL)
  return the (start of the) new list i.e. a pointer to the new person
otherwise
  use a loop to find the last item in the list
```

```
    (i.e. the one which has a "next" link of NULL)
  set the "next" link of this item to point to the new person
    so the new person becomes the last item in the list
    (i.e. the new person should have a "next" link of NULL)
  return the (start of the) list
```

Part 3: Insert into sorted list

Create a new copy of `insert_end`, called `insert_sorted`, and change your `main` to call this. Now modify `insert_sorted` to put people into the list in 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.
- 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.)

Hint:
The algorithm for `insert_sorted` is:

```
create a new space for the new person
(check it succeeded)
set the data for the new person
if the current list is empty or the first item on the list should follow the new person
  do the same as insert_start i.e.
  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
otherwise
  use a loop to find the last item in the list which should precede the new person
  set the new person's "next" link to point to whatever follows this list item
  set the "next" link of this item to point to the new person
  return the (start of the) list
```

Part 4: Parameterising the sort order

Add an extra parameter to `insert_sorted`, that is itself a (pointer to a) function. 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. Change your `main` to call `insert_sorted` with `compare_people_by_age` as the parameter, and check that the list is sorted by age instead of by name.

Part 5: Union

In this part, you need to modify your program to know about 2 different kinds of people: students, and staff:

- Use an `enum staff_or_student` to define `staff` and `student` and `neither`.
- Add a `staff_or_student` field to your `struct`.
- Use a union to add extra information to your `struct`:
  for students - a string holding their programme-name (e.g. "Computer Science" or "Artificial Intelligence").
  for staff - a string holding their room-number (e.g. "Kilburn 2.72")
  for other people - no further information.
- Modify the rest of your program so that `main` and `insert_sorted` together create some of each kind of `staff_or_student`, insert them into a sorted list, and print the resulting list.

(The important point of this part is to make sensible use of the `union`, not to write an elegant program - you can have a very simple mechanism for initialising the extra information e.g. extra pre-initialised arrays similar to `names` and `ages`.)

Part 6: Pointers to Pointers

In this part, you need to change `insert_sorted` to use a more complicated kind of pointer, which (eventually) allows you to simplify the algorithm.

- At the moment, you should be using a "pointer to a struct" to step through the items in the list in the while loop, to find where to do the insertion.
  Replace this by a "pointer to a pointer to a struct" (let's call it `ptr2ptr`).
- At the moment, you should be using an "if" to test whether to insert the new person at the head of the list, and then a "while" loop to find where to insert the new person in the rest of the list.
  Merge the "if" and the "while" into a single "while" loop.

Hint:
The algorithm for `insert_sorted` is:

```
As before:
  create a new space for the new person
  (check it succeeded)
  set the data for the new person
Set ptr2ptr pointing to the pointer to the start of the list
While ptr2ptr doesn't point to a null pointer and
            the item it (indirectly) points to is in the right order
  set ptr2ptr pointing to the pointer to the next item in the list
Finally
  set the new person's "next" link to point to whatever ptr2ptr (indirectly) points to
  set whatever ptr2ptr (indirectly) points to, to point to the new person
  return the (start of the) list
```

Marking Process

You must use `labprint` and `submit` as normal.

They will look for: `lists.c`

The marks are awarded as follows:

```
2 marks: Part 1 working completely
2 marks: Part 2 working completely
2 marks: Part 3 working completely
2 marks: Part 4 working completely
1 mark:  Part 5 working completely
1 mark:  Part 6 working completely

Total 10
```