Worksheet 6: Structs

Updated: 28th July, 2015

The objectives of this practical are to understand how to:

- design and use structs; and
- implement a linked list using structs.

Pre-lab Exercises

1. Creating Structs

Declare (with `typedef`) suitable structs to hold the following information:

(a) A date.

(b) A set of coordinates in 3D space.

(c) A postal address.

(d) The details of a practical class at Curtin.

(e) The details of *all* practicals for a unit.

2. Dynamic Allocation and Deallocation

Write C code to dynamically allocate and then deallocate the following:

(a) The date struct from Question 1a.

(b) An array of 25 coordinate structs from Question 1b.

(c) A struct called `Info` that contains a `char*` field called `name`, intended to store names up to 99 characters.

(d) An array of 25 `Info` structs.

(e) The following struct, so that it keeps track of an array of 25 `Info` structs:

```
typedef struct {
    int length;  /* Number of items in the list. */
    Info *list;
} InfoList;
```

3. Expressions with Pointers, Arrays and Structs

The following expressions use a combination of pointers, arrays and structs. In each case, describe (i) what "`var`" is by itself and (ii) what the overall expression is accessing.

(a) `var.b`

(b) `var->b`

(c) `var[i].b`

(d) `var[i]->b`

(e) `var->b[i]`

(f) `var[i]->b[j]`

(g) `*(var[i]->b)`

(h) `var->b->c`

(i) `var.b[i][j].c`

4. Linked List Structs

Design a set of structs to hold a linked list, where:

- The values to be stored in the list are structs called `FruitBat`.
- The size of the list must be instantly accessible.
- The last element of the list must be instantly accessible.

How does the last point complicate the insertion of a new element?

5. Miscellaneous Questions

(a) When inserting an element at the start of a linked list, you must (a) point the new node's "next" field to the existing first node, and (b) point "head" to the new node. Why does the *order* of these operations matter?

(b) Can an element belong to more than one linked list simultaneously?

(c) Can an element appear more than once in the same linked list?

Practical Exercises

1. Files and Structs

Write a C program to read a file and store the contents in a dynamically-allocated array of structs.

The file contains journal entries, each consisting of a date and a text string. Each entry occupies two lines. The first line of each entry contains the date (in "day/-month/year" format) and the second contains the text. The text can be up to 100 characters long, not including the new-line character.

The very first line of the file contains a single integer, indicating the number of journal entries in the file.

The following is an example input file:

```
4
12/04/2010
Interview went well I think, though was told to wear shoes.
18/04/2010
Doc advised me to concentrate on something... I forget.
03/05/2010
Was asked today if I was an art exhibit.
19/05/2010
Apparently mudcakes not made of mud, or angry wasps.
```

You should declare a suitable struct (in a header file), allocate an array just large enough to hold all the entries, and then read each entry into the array.

Note: You will need to make use of *both* the `fscanf()` and `fgets()` functions.

Your program should take a single command-line parameter — an integer (converted from a string). This is a journal entry index, where the first entry is index zero, the second entry is index 1, etc. Your program should print out the entry at that index.

For example, given the above input file and an index of 2, your program should output:

```
2010-05-03: Was asked today if I was an art exhibit.
```

Pay close attention to the date format — it differs from the input file!

Note: To have `printf()` use zeros instead of spaces as padding, place `0` between the `%` and the field width.

Finally, run valgrind on your program to test for memory errors, and fix any that you find.

2. Linked Lists

Write a set of C functions to manipulate a linked list of journal entry structs.

You will need to create appropriate structs, and write functions to:

(a) Create an empty linked list.

> **Note:** This is a simple function but not trivial, and it's vitally important. All of your other linked list functions will be useless if you don't know what you're doing here!

(b) Insert an element at the start. This function should take (i) a pointer to an existing linked list, and (ii) a pointer to an existing journal entry struct, to be inserted at the start of the list.

(c) Remove an element from the start. This function should take a pointer to an existing linked list, remove the first element from the list and return a pointer to it. (Return NULL instead if the list is empty.)

(d) Retrieve the *i*th element of the list (starting at zero for the first element). This function should not modify the list. (Return NULL if the index is beyond the end of the list.)

(e) Print out the contents of the list.

(f) Free the list.

You should place your code in a linked_list.c file, with a corresponding header file containing the struct and function declarations.

Write a main() function (in a separate file) to test your linked list code.

> **Note:** You *must* use valgrind to check for memory errors! We're getting into complex dynamic memory allocation, and mistakes are very easy.

3. Inserting at the End

If you create a linked list by inserting elements at the start, the list will come out in reverse order (as you may have observed).

Using diagrams and pseudocode (if necessary), devise an algorithm for inserting an element *at the end* of a linked list.

After insertion:

- The element should be the last element in the list.
- The structure should be a valid linked list.
- There should be no memory leaks (or other memory errors).

Implement a C function to do this. Test your function by inserting several elements into the list, then using your print function from Question 2e.

> **Note:** You may (or may not) choose to implement a "tail" pointer. If you do, remember that this may need to be updated when you insert or delete an element.

End of Worksheet