Lab Assignment 3: Structures

Dr Martin Porcheron

You have two weeks to complete this lab assignment—you must have it marked on Zoom by **19th February 2021**, but I would recommend you try and complete this as soon as possible. This assignment is worth 20 marks.

This lab task involves using structures to group together variables referring to the same objects within your program. For this you implement a structure and the necessary constructor and destructor functions. Your structure will represent a Welsh constituency, including the name of the constituency and a list of its bordering constituencies. As always, all dynamically allocated memory must have its lifespan managed properly and safely using the best practices we have discussed in the lectures.

You are required to name your functions exactly as given in this lab sheet.

I recommend you keep all your lab assignments in a new directory. If working on the lab machines, create this within your home directory for CSC371. Keep each lab assignment in a separate directory.

Note: Do not copy and paste code from the PDF as it will contain non-standard characters and will not compile.

This lab assignment was mostly originally written by Dr Joss Whittle.

Task 1: Building and destroying `structs` (4 marks)

In this exercise, you are going to construct, print, and destruct a `struct` with dummy data.

1. Download a copy of *task1.c* from the Canvas assignment page for Lab Assignment 3 but do NOT modify the `main` function, your solution for this task should execute without error using this exact `main` function.

```c
#include <stdio.h>

#include "constituency.h"

int main(int argc, char *argv []) {
    struct Constituency test;
    construct_Constituency(
        &test,
        "Swansea East",
        (unsigned int[]){1, 2, 3, 4},
        4);
    print_Constituency(&test);

    printf("\n");

    destruct_Constituency(&test);

    return 0;
}
```

   Observe the expression `(unsigned int[]){0, 1, 2, 3}` which declares a constant array of 4 `unsigned integers` (known and fixed at compile time). This is a nice shorthand notation rather than creating a variable with this array. Because this is known and fixed at compile time, the value passed to the function for that argument will be of type `const unsigned int * const`.

   We will encounter `const` in Lecture 7 more, but you should show initiative and research how it works now.

2. Create a pair of files *constituency.h* and *constituency.c*. **You must** use header guards in the header file.

3. In *constituency.h*, declare a structure `struct Constituency` with three member variables. Your `struct` should contain a pointer to a `char` called `name`, a pointer to an `unsigned int` called `neighbours`, and an `unsigned int` called `num_neighbours`.

   Look up information on `#pragma pack(1)` in C documentation online. What does this do and what are the implications of using it? Is it worth including this in your code or not?

4. In *constituency.h* declare 3 functions, `construct_Constituency`, `destruct_Constituency`, and `print_Constituency`.

   - `construct_Constituency` should take a pointer to a `struct Constituency` object and modify it to allocate space for the name and neighbour array. It should also take the `name` of the constituency as a `char` pointer, an `int` pointer to an array of `neighbours`, and the number of `neighbours` as an `int`, in that order. It should return a `void`.

     Hint: think about using `const` in function arguments and whether it is the pointer that is constant, or the content itself.

   - `destruct_Constituency` should take a pointer to a `struct Constituency` object and modify it to free its resources. Any non-`NULL` pointers within the structure should be freed and set to `NULL`. It should return a `void`.

- `print_Constituency` should take a pointer to a `struct Constituency` object and not modify it. It should print out the name of the constituency, the number of bordering consistencies, and the `integer` indexes of those neighbours. It should return a `void`.

From that description, I have the following protoypes:

```
void construct_Constituency(struct Constituency * const obj,
                            char const *name,
                            unsigned int* const neighbours,
                            unsigned int const num_neighbours);
void print_Constituency(struct Constituency * const obj);
void destruct_Constituency(struct Constituency * const obj);
```

5. In *constituency.c* implement the 3 functions:
   - In the constructor, you must for clear junk values from the `struct`'s data by setting them to `NULL` (or in the case of num_neighbours, `0`).
     - Then, you must allocate enough space for the `name` string and for the `neighbours` array using two calls to `malloc` and the correct sizes in bytes.
     - Use the `strlen` function to determine the length of a `\0` null terminated string.
     - You will then need to use the `memcpy` function to copy the name from the where the pointer passed into the function points to the `struct`'s newly `malloc`'d location for `name`.

       For example, the following call to `memcpy` would copy, in bytes, `name_length` number of `char`'s worth of information from whatever is stored at `name` into the `name` of a struct pointed to in `obj`:

       ```
       memcpy(obj->name, name,    name_length * sizeof(char));
       ```

     - *Remember:* as `obj` is a pointer to a struct, we can use the syntax `->` to follow the pointer to the object and get the variable after the arrow (i.e., this is equivalent to dereferencing the pointer and then using the `.` syntax, e.g. `(*obj).name`).
     - If there is one or more neighbours, you will also need to repeat the same step for the `neighbours`. Note that here you do not need to calculate the length—it is given to you in `num_neighbours`. Therefore, you need to `malloc` space for `num_neighbours * sizeof(unsigned int)` bytes, and copy this many bytes from the neighbours `paramter` into the `struct`'s variable.
   - In the print function, loop over the data and print it out using `printf()`.
   - In the destructor, `free` the data allocated on the heap and set the pointers to `NULL` (if they are not `NULL` already). Set the number of neighbours to `0` too.

   Hint: clear the values of the member variables first when constructing the `struct`. Remember that you need to add an extra space for the null terminator when allocating memory for a string.

6. From the command line compile and link your program with the GCC compiler using:

   ```
   $ gcc --std=c99 task1.c constituency.c -o task1
   ```

Expected output

If you run your program using the command:

```
$ ./task1
```

...you should get the output:

```
Swansea East | 4 neighbours | [ 1 2 3 4 ]
```

Task 2: Multiple test inputs (6 marks)

In this exercise you are required to modify the program you previously wrote for task 1 to add additional test data. You will have to expand the program to iterate through the data to identify constituencies with the most/fewest number of neighbours. You do not need to modify `constituency.h` or `constituency.c` for this task.

1. Copy your task 1 file and name this new version *task2.c*.
2. Modify the `main` function in *task2.c* with the following code which allocates an array of five constituencies and constructs them to have names and neighbours matching the local Swansea area.

```
const int num_constituencies = 5;
struct Constituency constituencies[num_constituencies];

construct_Constituency(
    &constituencies[0], "Swansea East",
    (unsigned int[]){1, 2, 3, 4}, 4);
construct_Constituency(
    &constituencies[1], "Swansea West",
    (unsigned int[]){0, 2}, 2);
construct_Constituency(
    &constituencies[2], "Gower",
    (unsigned int[]){0, 1, 3}, 3);
construct_Constituency(
    &constituencies[3], "Neath",
    (unsigned int[]){0, 2, 4}, 3);
construct_Constituency(
    &constituencies[4], "Aberavon",
    (unsigned int[]){0 , 3}, 2);
```

   Make sure to properly destruct all the constituency structures before the program terminates using your `destruct_Constituency` function.
3. Loop over each of the five constituencies and invoke the `print_Constituency` function so that they print to the command line.
4. While looping over the constituencies, determine the constituency with the most neighbours. After the loop has printed, you should print the constituency with the most neighbours, followed by the names of its bordering constituencies as such:

```
Swansea East has the most bordering constituencies:
    Swansea West
    Gower
    Neath
    Aberavon
```

   In the event of a tie, report which ever constituency is first in the array.
5. Likewise, you should do the same for the constituency with the fewest neighbours:

```
Swansea West has the fewest bordering constituencies:
    Swansea East
    Gower
```

   In the event of a tie, report which ever constituency is first in the array.
6. From the command line compile and link your program with the GCC compiler using:

```
$ gcc --std=c99 task2.c constituency.c -o task2
```

Expected output

If your run your program using the command:

```
$ ./task2
```

...you should get the output:

```
Swansea East | 4 neighbours | [ 1 2 3 4 ]
Swansea West | 2 neighbours | [ 0 2 ]
Gower        | 3 neighbours | [ 0 1 3 ]
Neath        | 3 neighbours | [ 0 2 4 ]
Aberavon     | 2 neighbours | [ 0 3 ]

Swansea East has the most bordering constituencies:
    Swansea West
    Gower
    Neath
    Aberavon

Swansea West has the fewest bordering constituencies:
    Swansea East
    Gower
```

Task 3: Data import (10 marks)

In this exercise, we are going to replace our test data with data imported from a text file.

1. Create a new file, *task3.c* with the following code:

```c
#include <stdio.h>

#include "constituency.h"

int main(int argc, char *argv[]) {

    // Parse the input file in to a new array and return
    // the pointer and size in the variables provided

    struct Constituency *constituencies;
    unsigned int num_constituencies;

    if (!load_Constituencies("wales.graph",
        &constituencies, &num_constituencies)) {
        // If function returns error code terminate ...
        return -1;
    }

    // If we have reached this line, the file has
    // been successfully loaded and the variables
    // constituencies and num_constituencies have
    // been set properly.

    // ... the rest of the program ...

    // Don't forget to deallocate when done!

    return 0;
}
```

2. In *constituency.h* declare a function `load_Constituencies`. The function should take as parameters: a pointer to a `char` containing a string file path to a file containing data to be loaded, a pointer to a pointer to an array of `Constituency` structures, and a pointer to an `unsigned integer` for storing the number of constituencies loaded.

   Hint: consider whether the pointer or underlying data should be `const`.

   The function should return an `integer` representing an error code, a return value of `0` indicates an error and `1` a success.

3. In *constituency.c* implement the `load_Constituencies` function. We need to use the `fopen`, `fclose`, and `fscanf` functions to read in parts of the file in their expected order.

   So here is some code which will open the file at `filepath` in read-only mode. It will check that it opened the file, and if it didn't, sets the pointers where the data is meant to be populated to `NULL` and then returns from the function:

```c
FILE * fp;
int line = 1;

fp = fopen(filepath, "r");
if (fp == NULL) {
        *out_constituencies = NULL;
        *out_num_constituencies = 0;
        printf("Could not open file \"%s\"\n", filepath);
        return 0;
```

```
}
```

- The format of the text file begins with a line containing a single `integer` representing the number of constituencies stored in the file. Make sure your code works at this stage before moving on to the next bullets.

  To retrieve a line from the file, we can use `fscanf` like so, pattern matching to an `unsigned integer` (with %u) and saves the value to `num_constituencies`:

  ```
  unsigned int num_constituencies;
  fscanf(fp, "%u\n", &num_constituencies);
  ```

  If there is not an `unsigned int` read in, you must print an error message and exit.

- After this, you will need to `malloc` for the number of constituencies you are expecting to parse.
- Each line after the first line is the same in format, and represents a single constituency (so loop for `num_constituencies` times!):
  - The first comma separated integer on each line represents the number of neighbours the current constituency has. Use `fscanf` as above to parse this value.
  - This is then followed by that many comma separated integers representing the 0-based indices of the neighbouring constituencies relative to their ordering in the file. You will have to loop over these values, using `fscanf` once for each neighbour.
  - After the last comma on each line the remaining characters up to the newline represent the name of the constituency.

  To get the name of the constiuency, you need to scan ahead to the end of the line to work out the amount of memory that needs allocating for the constituency name.

  The functions `ftell`, `fscanf`, and `fseek` will be helpful here. In other words, get the current position in the file (`ftell`), scan ahead for anything that isn't a new line character (search the web for how to do this in `fscanf`!), get the new position in the file, and the difference between the two positions is the length! You will need to seek back to the initial position (using `fseek`). Then allocate enough space in an array for the given number of characters, and read in the name using `fscanf` again.

  Finally, call `construct_Constituency` to create your struct, and then set the `out_constituencies` and `out_num_constituencies` pointers with correctly.

- Example file format:

  ```
  40
  6,6,29,32,17,26,33,Aberavon
  3,16,19,3,Aberconwy
  4,18,16,15,38,Alyn_and_Deeside
  ... omitted for space on lab sheet ...
  2,2,15,Wrexham
  1,3, Ynys_Mon_(Anglesey)
  ```

  This data file contains *40* constituencies. The first one, *Aberavon*, has *6* neighbours, which correspond to the constituencies on lines *7*, *30*, *33*, (and so on).

4. Use your code from task 2 to determine the constituency with the most and fewest neighbours and print their names, followed by the names of their bordering constituencies.

   Hint: ideally, your code from task 2 should *just work* without needing to be edited.

5. Ensure that the dynamically allocated array of `Constituency structs` is safely deallocated along with all of its elements before the program terminates.

6. Compile your program:

   ```
   $ gcc --std=c99 task3.c constituency.c -o task3
   ```

Expected output

If your run your program using the command:

```
$ ./task3
```

...you should get the output:

```
Aberavon                    | 6 neighbours | [ 6 29 32 17 26 33 ]
Aberconwy                   | 3 neighbours | [ 16 19 3 ]
Alyn_and_Deeside            | 4 neighbours | [ 18 16 15 38 ]
Arfon                       | 3 neighbours | [ 19 1 39 ]
Blaenau_Gwent               | 5 neighbours | [ 21 35 24 5 23 ]
Brecon_and_Radnorshire      | 8 neighbours | [ 24 25 14 12 26 17 23 4 ]
Bridgend                    | 3 neighbours | [ 37 0 29 ]
Caerphilly                  | 6 neighbours | [ 9 28 21 23 17 30 ]
Cardiff_Central             | 3 neighbours | [ 10 9 11 ]
Cardiff_North               | 6 neighbours | [ 8 10 28 11 30 7 ]
Cardiff_South               | 5 neighbours | [ 37 11 8 9 28 ]
Cardiff_West                | 5 neighbours | [ 37 30 9 10 8 ]
Carmarthen_East_and_Dinefwr | 7 neighbours | [ 22 20 26 5 14 31 13 ]
Carmarthen_West_and_...     | 2 neighbours | [ 12 31 ]
Ceredigion                  | 5 neighbours | [ 31 12 5 25 19 ]
Clwyd_South                 | 5 neighbours | [ 2 16 19 25 38 ]
Clwyd_West                  | 6 neighbours | [ 1 19 15 2 18 36 ]
Cynon_Valley                | 7 neighbours | [ 23 5 26 0 32 30 7 ]
Delyn                       | 3 neighbours | [ 36 16 2 ]
Dwyfor_Meirionnydd          | 5 neighbours | [ 25 15 16 1 3 ]
Gower                       | 5 neighbours | [ 34 33 26 12 22 ]
Islwyn                      | 5 neighbours | [ 7 28 35 4 23 ]
Llanelli                    | 2 neighbours | [ 20 12 ]
Merthyr_Tydfil_and_Rhymney  | 5 neighbours | [ 4 5 17 7 21 ]
Monmouth                    | 5 neighbours | [ 5 4 35 28 27 ]
Montgomeryshire             | 4 neighbours | [ 5 14 19 15 ]
Neath                       | 7 neighbours | [ 0 32 17 5 12 20 33 ]
Newport_East                | 2 neighbours | [ 38 24 ]
Newport_West                | 7 neighbours | [ 10 9 7 21 35 24 27 ]
Ogmore                      | 5 neighbours | [ 37 6 0 32 30 ]
Pontypridd                  | 7 neighbours | [ 37 29 32 17 7 9 11 ]
Preseli_Pembrokeshire       | 3 neighbours | [ 13 12 14 ]
Rhondda                     | 5 neighbours | [ 30 17 26 0 29 ]
Swansea_East                | 4 neighbours | [ 0 26 20 34 ]
Swansea_West                | 2 neighbours | [ 33 20 ]
Torfaen                     | 4 neighbours | [ 21 28 24 4 ]
Vale_of_Clwyd               | 2 neighbours | [ 16 18 ]
Vale_of_Glamorgan           | 5 neighbours | [ 10 6 11 29 30 ]
Wrexham                     | 2 neighbours | [ 2 15 ]
Ynys_Mon_(Anglesey)         | 1 neighbours | [ 3 ]

Brecon_and_Radnorshire has the most bordering constituencies:
    Monmouth
    Montgomeryshire
    Ceredigion
    Carmarthen_East_and_Dinefwr
    Neath
    Cynon_Valley
    Merthyr_Tydfil_and_Rhymney
    Blaenau_Gwent

Ynys_Mon_(Anglesey) has the fewest bordering constituencies:
```

```
    Arfon
```