Lab assignment 3: Data structures

Dr Martin Porcheron

You have two weeks to complete this lab assignment—you must have it marked/signed off by a lab helper in a timetabled lab session. You must get this assignment marked by **18th Februrary 2022** but I 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.

I have given instructions in this lab assignment for compiling code by command. I strongly encourage you to practice doing this, but you are also free to use an IDE if you wish. Using an IDE is your choice and you are responsible for configuring it.

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 `struct`s (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.

   ```
   #include <stdio.h>

   #include "constituency.h"

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

       printf("\n");

       destructConstituency(&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 6 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 `numNeighbours`.

   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, `constructConstituency`, `destructConstituency`, and `printConstituency`.

   - `constructConstituency` 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.

   - `destructConstituency` 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`.

- printConstituency 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 constructConstituency(struct Constituency * const obj,
                           char const *name,
                           unsigned int* const neighbours,
                           unsigned int const numNeighbours);
void printConstituency(struct Constituency * const obj);
void destructConstituency(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 numNeighbours, 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 either the memcpy or the strcpy functions (you can choose!) 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, nameLength 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, nameLength * 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 numNeighbours. Therefore, you need to malloc space for numNeighbours * 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 numConstituencies = 5;
   struct Constituency constituencies[numConstituencies];

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

   Make sure to properly destruct all the constituency structures before the program terminates using your `destructConstituency` function.
3. Loop over each of the five constituencies and invoke the `printConstituency` 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: Deep copying structs (10 marks)

In this exercise, we are go extend task 2 by making a deep copy of our array of structs, deleting the original, and then printing the copy.

1. Create a new file, *task3.c*, copying the code from *task2.c*
2. In *task3.c*, after you have constructed the 5 constituencies but before you loop through and print this data, add the following code:

   ```
   struct Constituency * copy = copyConstituencies(constituencies,
                                                   numConstituencies);

   int i = 0;
   for(i = 0; i < numConstituencies; i++) {
     destructConstituency(&constituencies[i]);
   }
   ```

   This code calls a function we will implement shortly called `copyConstituencies`. It then deletes our existing `constituencies` array by calling our `destructConstituency` function.
3. In *constituency.h* declare a function `copyConstituencies`. The function should take two parameters: an array of `struct Constituency` structures and a `const unsigned int` for the number of constituencies.
4. The function should return a pointer to `struct Constituency` (i.e., what will be the start of the array).
5. Implement this function in *constituency.c*. You should allocate new memory for the relevant variables within the struct, and copy across data.
6. Run your code now. You should either get a memory access error of some form, or perhaps your program will simply display null data if you are lucky.

   If you get the correct output as per task 2 still, check that your `destructConstituency` function is working correctly.
7. Adjust your code that loops through and calculates the constituency with the most/fewest neighbours and prints out the data so that it now uses this deeply copied array. Your program should work again as per task 2 with exactly the same output.