Assignment 3: "Here Be Dragons!"

Computer Programming (TPTG11) Autumn 2021

1 Assignment Description

The intended learning outcomes of this assignment includes using structs, files, dynamic memory management and modules. As part of this, you will also be working with arrays, strings and pointers. You will implement a database containing records of dragons (stored as an array of structs on the heap). The database will be created with an *initial capacity* (i.e. maximum number of dragons in the database), but your will implement functionality to dynamically grow the database to store additional dragons (when the *current capacity* has been reached). Your program should provide the following functionality:

1. Insert (add) a dragon (with all its attributes) to the database.
2. Update a dragon in the database.
3. Delete a dragon from the database.
4. Display all dragons in the database (brief).
5. Display all dragons in the database (detailed).
6. Select (retrieve) and display a specific dragon in the database (detailed).
7. Display statistics about the database.
8. Sort the database in ascending order of the dragons' `id` or `name`.

A *brief* display only includes a dragon's `id` and `name`, whereas a *detailed* display includes all the dragon's attributes (see below). You will enter all strings and characters into the database in upper case (it will make searching and sorting the database easier). Furthermore, you are expected to follow the coding conventions used in the book (such as naming variables, constants, types and functions) and structure your program into well-defined functions and modules. Document (with comments) everything in each module's header file (*.h* file), and where appropriate in a module's implementation file (*.c* file).

2 Program Startup

When your program starts, it should print the following message to the screen:

```
This program helps organize information about dragons. You may add and
remove dragons and their attributes, list the dragons currently in the
database, and their attributes, look up the attributes of an individual
dragon, get statistics from the database, or sort the database.
```

Then the program should ask the user for the name of a **text file**. The text file will be used to persist the database to your hard disk (it will contain all the dragon records in the database). If the user supplies the name of an existing text file, its contents should be read in to the program. If the text file doesn't exist, an empty file with the chosen name should be created.

Finally, the program should present the user with the **menu** shown below, through which the main functionality of the program is accessed. The user should be able to repeatedly select choices from the menu until the program terminates (and any invalid menu choices must be handled by the program).

```
  0. Display menu.
  1. Insert a dragon.
  2. Update a dragon.
  3. Delete a dragon.
  4. List all dragons (brief).
  5. List all dragons (detailed).
  6. Show details for a specific dragon.
  7. List database statistics.
  8. Sort database.
 -1. Quit.
?: 9
Invalid selection. Please try again.
?: -1
Have a good one! See ya!
```

2.1 Functionality

The functionality provided for each menu choice is described below.

2.1.1 Display menu (menu choice 0)

When the user chooses to display the menu, the menu shown above (with choices 0-8 and -1) should be displayed, including the prompt `?:`.

2.1.2 Insert a dragon (menu choice 1)

When the user chooses to insert (add) a new dragon to the database, the user should be asked to enter the following details; the dragon's `name` (i.e. a string only containing letters from the English alphabet), `isVolant` (i.e. if the dragon can fly or not, which is represented by the character `'Y'` or `'N'` respectively), `fierceness` (i.e. how fierce or dangerous the dragon is on a scale from 1-10), and `colours` (i.e. an array of strings representing the dragon's colours, where a minimum of 1 colour and a maximum of 5 colours is possible).

If the user doesn't enter a valid `name` (i.e. anything but letters from the English alphabet), the program should output an error message and ask the user to try again.

When a valid name has been entered, the program should ask the user to enter if the dragon can fly or not, i.e. `isVolant`. Only the characters `'Y'` or `'N'` (or their lower case equivalents `'y'`) or `'n'`) are valid. An invalid choice is again handled with an error message and asking the user to try again.

Next, the program should ask the user to enter the dragon's `fierceness`, where the only valid values are in the integer range 1-10. An invalid choice is yet again handled with an error message and asking the user to try again.

Finally, the program should ask the user to supply between 1-5 colours for the dragon (where a colour is an arbitrary string representing a colour). The user should not select the number of colours to enter, prior to entering the colours. Instead, the program should stop asking for additional colours when the user enters an empty string, or when 5 colours have been entered.

The dragon's `name`, `isVolant` and `colours` should all be **converted to upper case**, before inserting the dragon into the database.

When all the dragon's attributes have been supplied by the user, the dragon should be inserted (entered) into the database, and the user notified about this. The database should **assign a unique `id`** to the dragon, when inserting it into the database (the `id` should not be entered by the user). A dragon's `id` will never change, as long as the dragon is in the database (even when loading the database from a text file). Finally, the prompt `?:` should be displayed, so that the user can select another menu choice.

2.1.3 Update a dragon (menu choice 2)

When the user chooses to update a dragon, the user should be asked to enter the `id` **or** `name` of a dragon. The user should **not** first be asked if an `id` or `name` should be entered. Instead the program should read in the entry as a **search string**, and interpret the search string as an `id` or `name` (remember, an `id` is an integer, and a valid `name` only contains letters from the English alphabet).

If the user enters a `name`, and the database contains multiple dragon's with the same `name`, the program should list all dragons that matched the query. Only the dragon's `id` and `name` should be listed (i.e. a *brief* list). Then the user should be asked to pick one of the dragons. This process should be repeated until only one dragon is selected.

If no dragon matches the query, the program should inform the user about this, and then display the prompt `?:` so that the user can select another menu choice.

If one specific dragon is selected, the user should be able to enter new values for the dragon's `isVolant`, `fierceness` and `colours` using the same process described above for inserting a dragon. The dragon's `id` or `name` should **not** be changed. When all attributes have been entered by the user, the dragon should be updated in the database with its new values, and the user notified about this. Finally, the prompt `?:` should be displayed, so that the user can select another menu choice.

2.1.4 Delete a dragon (menu choice 3)

When the user chooses to delete a dragon, the user should be asked to enter the `id` **or** `name` of a dragon. The process should be the same as described above for updating a dragon with respect to handling multiple matches or no matches. If one specific dragon is selected, the dragon should be removed from the database, and the user notified about this. Finally, the prompt `?:` should be displayed, so that the user can select another menu choice. Note that a dragon with the same details can be inserted into the database again (menu choice 1), but a new unique `id` will be assigned to the dragon.

2.1.5 List all dragons (brief) (menu choice 4)

When the user chooses to display a brief list all dragons in the database, the `id` and `name` of each dragon in the database should be displayed on the screen. Use conversion specifiers with proper field widths to produce a good looking print-out. Also include a header (column names) for the dragons' two attributes. Finally, the prompt `?:` should be displayed, so that the user can select another menu choice.

2.1.6 List all dragons (detailed) (menu choice 5)

When the user chooses to display a detailed list of all dragons in the database, all attributes of each dragon in the database should be displayed on the screen. Once again, use conversion specifiers with proper field widths to produce a good looking print-out, and also include a header (column names) for all the dragons' attributes. Finally, the prompt `?:` should be displayed, so that the user can select another menu choice.

2.1.7 Show details for a specific dragon (menu choice 6)

When the user chooses to display the details of a specific dragon, the program should ask the user to enter an `id` **or** a `name`. The user should **not** first be asked if an `id` or `name` should be entered. Instead the program should read in the entry as a **search string**, and interpret the search string as an `id` or name (remember, and `id` is an integer, and a valid `name` only contains letters from the English alphabet). All attributes of all dragons matching the query string should be displayed. Once again, use conversion specifiers with proper field widths to produce a good looking print-out, and also include a header (column names) for all the dragons' attributes. Finally, the prompt `?:` should be displayed, so that the user can select another menu choice.

2.1.8 List database statistics (menu choice 7)

When the user chooses to list database statistics, the program should display the following details; the **number of dragons** in the database, the maximum and minimum `fierceness` among all dragons in the database, the **number of dragons that can fly** (i.e. where `isVolant` is `'Y'`) and the **number of dragons that can't fly** (i.e. where `isVolant` is `'N'`). Once again, use conversion specifiers with proper field widths to produce a good looking print-out, and also include a header (column names) for the various statistics. If the database is empty, the program should notify the user about this. Finally, the prompt `?:` should be displayed, so that the user can select another menu choice.

2.1.9 Sort database (menu choice 8)

When the user chooses to sort the database, the program should ask the user if the database should be sorted on the dragons' `id` or `name`. Then the program should sort the database in **ascending** order (i.e. smallest `id` first, or alphabetically when sorting on the `name`), and the user should be notified about this (you can modify one of the sorting algorithms provided during the lectures for this). Note that the database should not be displayed automatically after it has been sorted. Finally, the prompt `?:` should be displayed, so that the user can select another menu choice.

2.1.10 Quit (menu choice -1)

When the user chooses to quit, the program should display an exit message, and then terminate.

2.2 Dynamic Memory Allocation and File-Handling

Since the program should be able to store an unlimited amount of dragons in the database, the array containing the dragon records (structs) must be allocated on the *heap* and dynamically grown to accommodate more dragons if necessary. The database should start with an *initial capacity* (i.e. the maximum number of dragons it can store). We will define a symbolic constant `INITIAL_CAPACITY` for this. When the database's *size* reaches its *current capacity*, the database's *current capacity* should grow in size according to the value of another symbolic constant `GROWTH_FACTOR`. If the database's *size* reaches its *current capacity*, and the value of the `GROWTH_FACTOR` is `2`, then the database's *current capacity* should double in size. Note that the database's `INITIAL_CAPACITY` and *current capacity* are only initially the same (when the program starts), but the *current capacity* can grow in size, whereas the `INITIAL_CAPACITY` cannot.

Remember that a dragon can have, at most, 5 colours? Let's also define this as a symbolic constant `MAX_COLOURS` so that we can change the program's functionality with respect to this, only by changing the value of the symbolic constant.

The program keeps its database in Random Access Memory (RAM) on the `heap` while the program is running. But, since RAM is *volatile memory* (i.e. if we turn off the power to the computer, all contents in RAM is lost), we need to persist our database to the file system. Therefore, when the program starts, it will load all contents from a text file containing dragon records into the program (into RAM), and likewise, when the program terminates, it will first save all the records in the database (from RAM) to the text file (overwriting the text file). The reason why we are using a text file in this assignment (and not a more efficient binary file), is so that you can actually open the file and make sure everything is stored correctly.

We need to decide how to structure the text file so that we save and load the text file using the same format. We will use the format shown below (a sample text file `"dragons.txt"` containing 5 dragons has been provided for you on Canvas):

```
5
1
SMAUG
Y
8
2
RED
GOLD
2
NOBERT
N
4
3
BROWN
BLACK
WHITE
...
6
```

The first number 5 is the number of dragons stored in the file. This is followed by the attributes (details) of the first dragon; the number 1 is the dragon's unique `id`, `"SMAUG"` is the dragon's `name`, `"Y"` is the value for the dragon's `isVolant`, the number 8 is the dragon's `fierceness`, the number 2 tells us the dragon has two colours, followed by the actual colours `RED` and `GOLD`. After this, the attributes (details) of the next dragon, starting with the unique `id` follow, and so on. The final number in the text file 6 is the next available unique `id` that the database should assign to the next dragon inserted (added) to the database (note that this number should not be used for a new dragon when loading the dragon records from the text file into the program (into RAM), but only when inserting new dragons from the program's menu). The `...` shown in the sample file above is not an actual row in the text file (it is only used here to represent more dragon records in the text file - so don't include it in your text file!). Note that the next available unique `id` will never reset or decrease. It will only increase as more dragons are inserted into the database via the menu.

When the database is sorted, all attributes (details) for a dragon, of course, remain with that dragon instance, but the records might change place in the text file after sorting the database and terminating the program (i.e. when the sorted database in RAM is saved to the text file).

3 Suggested Approach

This section gives some guidance on what order to implement the various functionality in the program.

You can use the same file structure and Visual Studio Code configuration files (`launch.json`, `tasks.json` and `c_cpp_properties.json`) as in assignment 2. You might want to place the database files into yet another subfolder in your workspace (e.g. `files`). If you do, remember to include the subfolder name when referring to the text file from within the program. For example, if you have placed the sample file `"dragons.txt"` in the subfolder `files`, you would specify `"files/dragons.txt"` when opening the file in your program (the path is relative to your Visual Studio Code workspace folder, not to the folder `bin` in which your executable file resides).

Try to separate your code into modules. Remember a module contains functions with similar functionality. For example, you might create a module for a user interface (functionality for communicating with the user via the terminal). You might choose to create a module for the database, a module for the dragon, a module for file-handling, and so on. Also remember to separate a module's code into a header file (*.h* file), which constitutes the module's public interface, and an implementation file (*.c* file). Function prototypes, typedefs, defines, symbolic constants, etc that you want other modules and your client code (`main.c`) to access, go into the *.h* file. Everything else (private function prototypes, typedefs, defines, and symbolic constants, including all function implementations) go into the *.c* file. Remember, that if you declare a function, function prototype or global variable as `static`, it becomes private, i.e. it is only visible inside the *.c* file in which it is defined (therefore, `static` should never be used in a *.h* file, which contains the module's public interface!). Document (with comments) your types, function prototype and symbolic constants in the *.h* file, and if necessary (e.g. to clarify how an algorithm or function implementation works) in the *.c* file.

Lastly, use the principle of structured programming with stepwise refinement to break down your functions into smaller and smaller functions with specific functionality. The Visual Studio Code debugger will also be an invaluable aid in this assignment (use it to debug e.g. dynamic memory allocation on the heap).

3.1 Step 0

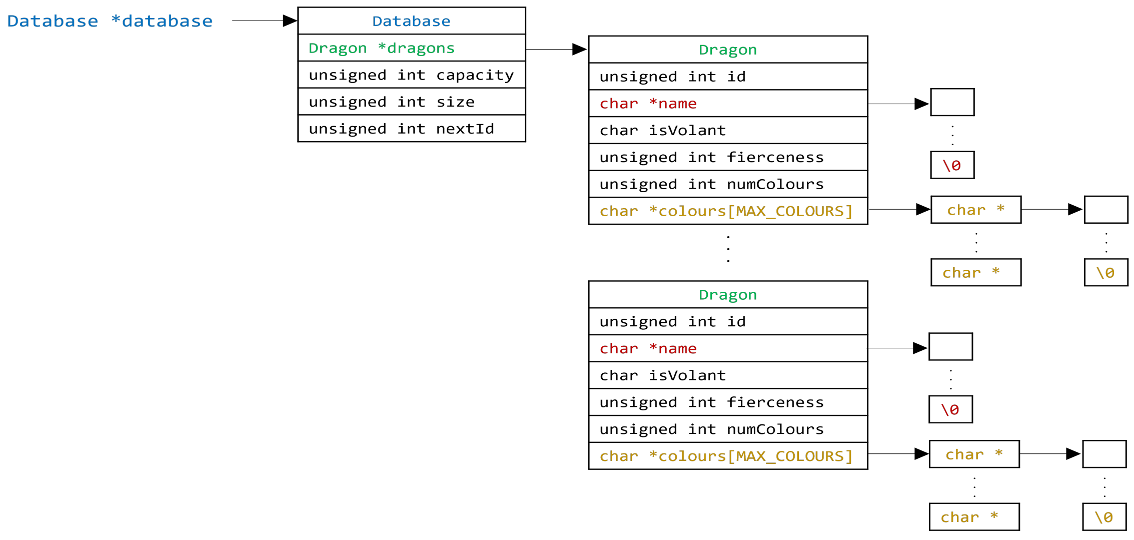
Start by placing the defines and typedefs below into appropriate modules and document them using comments. These are the defines and typedefs you **must** use to represent a dragon and the database.

```
#define MAX_COLOURS 5

typedef struct Dragon {
        unsigned int id;
        char *name;
        char isVolant;
        unsigned int fierceness;
        unsigned int numColours;
        char *colours[MAX_COLOURS];
} Dragon;

#define INITIAL_CAPACITY 10
#define GROWTH_FACTOR 2

typedef struct Database {
        Dragon *dragons;
        unsigned int capacity;
        unsigned int size;
        unsigned int nextId;
} Database;
```

The symbolic constants `MAX_COLOURS`, `INITIAL_CAPACITY` and `GROWTH_FACTOR`, including the `Dragon` struct, should be familiar from the description above in this document.

The `Database` struct represents the database. The `dragons` member is a pointer to a `Dragon`, and points the array that will be dynamically grown when the number of dragon records reach the database's *current capacity*. The database's *current capacity* (i.e. the maximum number of dragon records that can currently be stored in the dynamic `dragons` array) is stored in the member `capacity`. The actual number of dragon records currently contained in the `dragons` array, is stored in the member `size`. Lastly, the next available unique `id` is stored in the `nextId` member.

When the database's current `size` reaches its current `capacity`, its capacity should be multiplied by the `GROWTH_FACTOR` and the new capacity stored back into the `capacity` member. Also, the dynamic array `dragons` must be increased to the new `capacity`. Initially (when the program starts), the database's current capacity is set to the value of the symbolic constant `INITIAL_CAPACITY`. Each time a dragon is added to the database (i.e. the dynamic array `dragons`), the database's `size` is incremented. Then, when the database's `size` reaches its current `capacity`, the database is grown as describe above. To do this, you can use the functions `malloc` or `realloc` in `stdlib.h`.



3.2 Step 1

Create a file `main.c` with a `main` function, containing the following structure, and also include any necessary .h files (note that the code won't compile as is):

```
int main(void)
{
    char filename[MAX_FILENAME];

    Database *database = createDatabase();
    getDatabaseFilename(filename);
    loadDatabase(filename, database);

    printWelcomeMessage();
    executeCommands(database);

    saveDatabase(filename, database);
    destroyDatabase(database);

    return 0;
}
```

The function `createDatabase` creates an empty database (on the heap) and returns a `Database` pointer to it. The function `getDatabaseFilename` prompts the user for a database **filename** (text file, e.g. `"dragons.txt"`). The function `loadDatabase` then uses the **filename** and the `Database` pointer to load the records stored in the text file into the database. Next, the function `printWelcomeMessage` prints a welcome message on the screen (as previously described). The program's main functionality (the menu) is contained in the function `executeCommands`. When the user chooses to quit the program (from the menu), the function `executeCommands` returns, whereby the function `saveDatabase` is called to store the database to the chosen text file (e.g. `"dragons.txt"`), which is overwritten. Once the database is persisted to the hard disk, the function `destroyDatabase` is called to destroy (free) the database (i.e. deallocate the database from the heap). Finally, the `main` function returns, and the program terminates.

Note that the function `createDatabase` should allocate memory on the heap for both the `Database` struct and the dynamic array contained in its member `dragons`! Also initialise its members `capacity` to the `INITIAL_CAPACITY`, `size` to `0`, and `nextId` to `1`. The sample file `"dragons.txt"` contains `5` dragon records, so start by setting the symbolic constant `INITIAL_CAPACITY` to something like `10`. That way you won't have to bother about growing the database until you have finished implementing and testing loading the database from the text file into RAM.

Test your program so that you can load the text file into the database and save it back to the text file (overwriting the file), whereby the file contents should be the same as when the text file was loaded. As part of this step, you can also implement menu choice 0 (displaying the menu) and -1 (quitting the program). Then it's time to proceed to the next step.

3.3 Step 2

Now implement the functionality to list dragons in the database (menu choice 4, 5 and 6). Start by implementing the brief list (menu choice 4) of all dragons in the database (i.e. only printing the dragons' `id` and `name`).

After you have tested this successfully, add the functionality for the detailed list (menu choice 5), i.e. all attributes of all dragons in the database.

When this is working, implement the functionality to display a detailed list, i.e. all attributes, for a specific dragon (menu choice 6). The tricky part here is querying the user for a search string (`id` or `name`), and searching through the dragon array in the database to find the desired dragon (or dragons). Don't forget to break down the problem into smaller pieces using stepwise refinement, to obtain a well-structured program.

3.4 Step 3

Next, implement the functionality for listing database statistics (menu choice 7). This shouldn't be too difficult after the previous step. When you can list dragons and database statistics, you have a way of verifying that the database looks correct as you implement the remaining functionality.

3.5 Step 4

It's time to implement the functionality to update a dragon in the database (menu choice 2). This requires searching for a specific dragon using a search string based on `id` or `name` (functionality you have already implemented as part of the detailed list for a specific dragon above, and these functions should be re-used for this menu choice). When you have found the dragon record in the database, modify its members with the new values (remember, the user should be able to modify all attributes, except for the dragon's `id` and `name`). Then use your existing functionality to list the dragons in the database, and make sure the dragon's details were updated correctly. Then save the database to the text file, and load it back in again, to verify that also works.

3.6 Step 5

Let's implement the functionality to insert (add) a new dragon to the database next (menu choice 1). The tricky part here is to query the user for all the dragon's details (you can skip input validation initially, and then add validation after you have successfully inserted a dragon). Don't forget to convert all strings an characters to upper case. Actually adding the dragon to the database should be similar to (but not quite the same as) loading dragons from a text file into the database. The big difference, in this case, is that you must assign the next unique `id` to the dragon (and not use an existing `id` as you did when you loaded the dragons from the text file into the database). Use your existing functionality for listing all dragons in the database, to verify everything works. Then save the database to the text file, and load it back in again, to verify that also works. Don't exceed your database's capacity just yet. We will worry about growing the database's size dynamically later.

3.7 Step 6

Now add the input validation to menu choices 1 and 2 (adding and updating dragons). Make sure to break this down into suitable functions that are used for input both when adding a new dragon and updating the details of an existing dragon.

3.8 Step 7

We have one basic database operation left to implement. Deleting a dragon from the database (menu choice 3). When deleting a dragon from the database, we have some work to do with regards to the dynamic array of dragons, since simply deleting a dragon from the array would leave a gaping "hole" in the array. We need to "fill in" this "hole" by shifting all dragon records succeeding the deleted dragon record one step "left", to cover the "hole". Note that all dragons still keep their respective `id` (and all other attribute values) when shifting them. An example is shown below when deleting the dragon with `id` 3.

dragon id	1	2	3	4	5	6	⋯	$< undefined >$
array index	0	1	2	3	4	5	⋯	CAPACITY-1

`dragons array (before)`

dragon id	1	2	4	5	6	⋯	⋯	$< undefined >$
array index	0	1	2	3	4	5	⋯	CAPACITY-1

`dragons array (after)`

Use Visual Studio Code's debugger as an aid, and your existing functionality for listing all dragons in the database, to verify everything works. Then save the database to the text file, and load it back in again, to verify that also works.

3.9 Step 8

Now that we can insert, update, delete and list dragons, including listing statistics, we are ready to implement functionality for sorting the database in **ascending** order, based on `id` or `name` (menu choice 8). You can modify one of the sorting algorithms from the lectures to implement the sorting functionality. Once again, use your existing functionality for listing all dragons in the database to verify everything works. Then save the database to the text file, and load it back in again, to verify that also works. The dragons should be written to (and read from) the text file in their sorted order. Note that sorting the dragons should not change the `id` a dragon has (or any other members of a dragon record). Only the order of the dragons in memory should be affected when sorting.

3.10 Step 9

We're almost done. So far we have delayed implementing the functionality for dynamically growing the database (i.e. the dynamic array containing the dragon records). We initialized the database's current `capacity` to an initial capacity (.e.g by setting `INITIAL_CAPACITY` to `10`). When the database's current `size` reaches it's current `capacity`, we need to dynamically grow the database (i.e. the dynamic array stored in the database's `dragons` member) by a factor determined by the symbol constant `GROWTH_FACTOR`. A `GROWTH_FACTOR` of `2` means allocating twice as much memory, copying the existing dragon records to the new memory, and then freeing the old memory. You can use the functions `malloc` and `free` to do this, or the function `realloc` (all in `stdlib.h`). Here's an example approach using the first alternative:

1. Declare a new dynamic array of type `Dragon`.
2. Allocate twice as much heap memory for the new array, compared to the old (existing) array's capacity (if `GROWTH_FACTOR` is set to `2`).
3. Copy the contents of the old array to the beginning of the new array.
4. Free the old array's heap memory.
5. Assign the new array to the database's `dragons` member.
6. Update the database's `capacity` member with the new capacity.

Once again, use your existing functionality for listing all dragons in the database, to verify everything works. Then save the database to the text file, and load it back in again, to verify that also works. Visual Studio Code's debugger can be useful here too.

3.11 Step 10

Phew! Were almost done! All that's left to do is polish up the program's menu and any other functions that need refactoring. Test the program with various inputs, and make sure everything works as expected before submitting the assignment. Look over your programs design (should any code be moved into its own module, or should any large function be broken down into smaller, more manageable functions?). Make sure you have documented (with comments) everything in your .h files (and, if necessary, added comments to your .c files).

4 Sample Execution

```
> main
This program helps organize information about dragons. You may add and
remove dragons and their attributes, list the dragons currently in the
database, and their attributes, look up the attributes of an individual
dragon, get statistics from the database, or sort the database.

Database file name (max 99 characters): files/dragons.txt

------------------------------------------
Menu
------------------------------------------
 0. Display menu.
 1. Insert a dragon.
 2. Update a dragon.
 3. Delete a dragon.
 4. List all dragons (brief).
 5. List all dragons (detailed).
 6. Show details for a specific dragon.
 7. List database statistics.
 8. Sort database.
-1. Quit.
------------------------------------------

?: 0

------------------------------------------
Menu
------------------------------------------
 0. Display menu.
 1. Insert a dragon.
 2. Update a dragon.
 3. Delete a dragon.
 4. List all dragons (brief).
 5. List all dragons (detailed).
 6. Show details for a specific dragon.
 7. List database statistics.
 8. Sort database.
-1. Quit.
------------------------------------------

?: 1

Enter name: test
Enter volant (Y, N): y
Enter fierceness (1-10): 5
Colour 1 (of 5): red
Colour 2 (of 5):
Dragon inserted.

?: 2

Enter id or name of dragon: test

Enter volant (Y, N): n
Enter fierceness (1-10): 1
Colour 1 (of 5): green
Colour 2 (of 5): blue
Colour 3 (of 5):
Dragon updated.
```

```
?: 5

--------------------------------------------------------------------------------
ID Name                 Volant Fierceness #Colours Colors
--------------------------------------------------------------------------------
 1 SMAUG                     Y          8        2 RED GOLD
 2 NORBERT                   N          4        3 BROWN BLACK WHITE
 3 APOPHIS                   N          5        2 WHITE BLACK
 4 GLAURUNG                  N          8        1 GOLDEN
 5 ANCALAGON                 Y          9        1 BLACK
 6 TEST                      N          1        2 GREEN BLUE

?: 3

Enter id or name of dragon: 6
Dragon deleted.

?: 4

--------------------------------------------------------------------------------
ID Name
--------------------------------------------------------------------------------
 1 SMAUG
 2 NORBERT
 3 APOPHIS
 4 GLAURUNG
 5 ANCALAGON

?: 6

Enter id or name of dragon: 1

--------------------------------------------------------------------------------
ID Name                 Volant Fierceness #Colours Colors
--------------------------------------------------------------------------------
 1 SMAUG                     Y          8        2 RED GOLD

?: 8
Enter sort by id (0) or name (1): 1
Database sorted.

?: 5

--------------------------------------------------------------------------------
ID Name                 Volant Fierceness #Colours Colors
--------------------------------------------------------------------------------
 5 ANCALAGON                 Y          9        1 BLACK
 3 APOPHIS                   N          5        2 WHITE BLACK
 4 GLAURUNG                  N          8        1 GOLDEN
 2 NORBERT                   N          4        3 BROWN BLACK WHITE
 1 SMAUG                     Y          8        2 RED GOLD

?: 7

---------------------------------------------------
Size MinFierceness MaxFierceness #Volant #NonVolant
---------------------------------------------------
   5             4             9       2          3

?: 9
```

```
Invalid selection. Please try again.

?: -1

Have a good one! See ya!
```

5 Requirements

- You must write the code yourselves. Code snippets from the textbook and/or lectures can of course be used, but it is not allowed to find and use code from the internet.
- The program that you submit must compile and run without modification. This means that any code that gives compilation or runtime errors must be commented out.

6 Grading criteria

You will receive the grade pass or fail. In order to pass, the program must be complete, i.e. have the full functionality described above. In addition, the program must be well-designed and well-coded, especially with regards to:

- The separation of interface and implementation into each module's .h and .c files respectively.
- How it is broken down into functions (especially if you use helper functions in you modules).
- Adequate data types and consistency in this between functions.
- Proper use of structs, file-handling, memory management, pointers, arrays and strings.
- Naming of functions and variables must be in accordance with the principles presented in the course book.
- Comments (documentation) for defines, typedefs, symbolic constants and function prototypes in header files (.h files),
- Comments describing non-trivial functionality in implementation files (.c files).

7 Submission

You must submit your solution via Canvas, in a zip file containing your Visual Studio Code workspace folder with all your source code (all documentation is done using comments in your source code). Don't submit anything else (just one zip file).

Deadline for submission is Friday 17th of December at 17.15.

Re-examination deadline is January 31st, 2022.