Laboratory Exercise

Working with pointers and arrays, declarations and definitions

Purpose of the exercise

This exercise will help you do the following:

- Practice the syntax related to pointers to `struct` objects and to arrays.
- Practice working with type aliases and data types beyond the basic types available in C.
- Develop familiarity with the concept of encapsulation.

Overview

Big Numbers

In this exercise you will be performing addition of unsigned integer numbers, but do not think it is a trivial task; far from it! If a data type of a variable is `unsigned long long int` with the size of 64 bits, the biggest number it can possibly represent is: `18,446,744,073,709,551,615`.

This looks impressive until we consider that this number has only 20 digits. What if we want to add numbers up to 100 digits long? What other data type allows for even larger storage? Arrays!

In this exercise we are interested in adding extremely big numbers and using arrays to help us do that. For example, our program should be able to perform the following addition:

```
1772836615228346888745873818175356155612556737278934
                      476974164987831717398187573754
----------------------------------------------------
1772836615228346888746350792340343987329954924852688
```

As we do not know the length of a big number upfront, its digits will be stored in a dynamically allocated array of `unsigned char` values. To eliminate ambiguity, we will give this type an alias - `BigDigit`. With this abstraction we do not have to think about the exact underlying data type.

We will assume that each element of an array will store a single digit - an value from `0` to `9` only (not an ASCII character), and that the most significant digit will be placed at the position with the index 0. We will deal only with non-negative numbers.

Each number will be therefore composed of two variables:

- `digits` - a pointer to a dynamically allocated array of `BigDigit` elements.
- `length` - an unsigned integer indicating how many digits are in the array.

We can store these variables together in a structure `BigNumber`. Define it inside *add_digit.c* as:

```c
struct BigNumber
{
    BigDigit* digits;
    size_t    length;
};
```

The data members from this definition will not be available in any other translation unit.

Encapsulation

In previous laboratory exercises you have learnt about two important programming techniques:

- **Decomposition** - breaking hard to tackle problems into smaller ones to address them separately,
- **Abstraction** - giving concepts names and modeling their use, with their implementation provided separately to reduce the amount of details a programmer should concern themselves with.

Now it is time to add a new one to your toolbox: **encapsulation**, a concept related closely to abstraction of data. Encapsulation is a technique used to hide internal, private data related to an abstraction, away from client code that uses this abstraction, and to make it available only to the authorized functions that are permitted to operate on this internal data.

For example, in this exercise you will implement functions operating on `struct BigNumber`. But the main program and any code in its translation unit should not have access to its data members. Only the functions that you have to define for creating, destroying and operating on `BigNumber` objects should understand what data members are hidden inside. We say that such data members are encapsulated.

How to achieve such encapsulation in C? This approach is not complex, but requires you to understand and appreciate a difference between *a declaration* and *a definition*. In C, you can declare file-scoped variables or functions (which by default have `extern` linkage) and use them in the same translation unit, while providing their definitions in a separate translation unit, but this is only partially true for `struct` data types. We can declare them in one translation unit (in this exercise, `struct BigNumber` has been declared in *main.c* via an included *add_digit.h* header) without specifying their data members as shown below:

```
struct BigNumber;
```

Such `struct` declarations, called forward declarations, impose an important limitation. As the compiler does not know the memory layout or even the size of objects of this type, this type is considered *incomplete* until the declaration is followed by a definition. We cannot perform operations on such a type, instantiate objects of this type, or even declare functions accepting parameters or returning results of this type. Luckily, we can work on *pointers* to forward declared `struct` types; a compiler knows the size and the representation of a pointer, even if the pointer arithmetic may not be available. We can also create `typedef` aliases to such `struct` types. The header *add_digit.h* offers both an alias and declarations of functions that work on pointers to the forward declared `BigNumber` data type.

If you are curious how to define the functions (after all, the definitions may need to operate on a forward declared data type), the answer is simple: they will be defined in the same translation unit as the definition of the complete data type. Definitions of these privileged functions will be encapsulated together with the definition of the `struct` data type, while their declarations - operating only on pointers to the data type - will be offered with a forward declaration, making sure that its data members are not accessible.

Encapsulation protects data members from being accessed by unauthorized functions (they can be still be modified by direct manipulation of bytes in an object's memory), thus better safeguarding consistent states of objects of that type. For example, since the only function that knows how to create `BigNumber` objects will be `create_BigNumber()`, a correct implementation of this function guarantees that when an object is created, its `length` properly corresponds to the space available in the dynamically allocated array for its digits; no unprivileged function can break this assertion.

Tasks

In this exercise you will implement functions that will help the program encapsulate the `BigNumber` data type. The *main()* function should be able to operate on such objects exclusively through your provided functions, without ability to access the data members directly.

Inside *add_digit.c*, you will need to define `struct BigNumber` as shown earlier, and define the four functions that have been declared in the *add_digit.h* (take note that they operate on `BigNumber*` values, not `BigNumber` values):

1. Creating `BigNumber` objects

```
BigNumber* create_BigNumber(const char* text);
```

This function creates a new `BigNumber` object with the digits corresponding to the string of text. To make sure the object survives the end of the function's block scope, the object **must** be dynamically allocated in the heap segment using malloc(). Once the object is allocated, the `length` data member of the object must be set to the same value as the number of characters in the input text (use strlen() to determine the length). The `digits` data member must be set to an address of a dynamically allocated array on the heap as well (since its length is not known during compilation). Consider how many bytes each of these allocations will require.

Lastly, populate the array with digits derived from the text's ASCII characters. The `digits` buffer is not null-terminated as the element `0` is a valid digit value; this is why the `length` data member is necessary to record the count of available digits (needed during printing and addition).

2. Destroying `BigNumber` objects

```
void destroy_BigNumber(BigNumber* number);
```

This function frees the heap memory of a `BigNumber` object using free(). Remember that you need to release memory after the `digits` data member first before you release memory of the `BigNumber` object itself.

3. Adding two `BigNumber` objects

```
BigNumber* add_BigNumber(
    const BigNumber* number1,
    const BigNumber* number2);
```

This function allocates a result `BigNumber` object, stores in it a value that corresponds to the sum of `number1` and `number2`, and returns the result.

Because the result of addition can have as many digits as the longer of two numbers plus one more position for a carry digit, find the longer of two inputs, allocate the result accordingly and then add the inputs using an addition with carrying approach. If the result contains a leading zero, eliminate it by shifting all the digits by one position towards the lowest index in the array and reduce the `length` value.

While performing addition, loop over both inputs from the least significant position to the most significant position of the longer input (if an input does not have that many digits, assume that you are working with one or more leading zeros), and for each position call the `add_BigDigit()` function provided for you in *main.c*; it takes two input digits and a carry over digit from the previous position, adds them together and updates the carry over.

4. Printing the sum of `BigNumber` objects

```
void print_BigNumber_sum(
    const BigNumber* number1,
    const BigNumber* number2,
    const BigNumber* sum);
```

This function takes in three values and displays the text output to the standard output stream as shown below:

```
  10000
+  1234
-------
  11234
```

All three values must be aligned to the right, with the biggest number printed on the third column onwards, and the second input with the plus symbol on the first column followed by one or more spaces. The inputs and the result should be also separated by a line of the minus symbols, dashes, of the appropriate length as demonstrated in the example above and in `expected-output.txt`.

While defining the four aforementioned functions, take note that they have been declared with a keyword `extern`. Recall that in the C programming language a function declaration that appears in the file scope has default external linkage. This means that, unlike file scope variables, functions do not have to be marked as `extern` when they are defined in other translation units. However, in the header *add_digit.h* these functions have been marked explicitly as `extern` to remind you that you must define them outside the driver program. Likewise, if you want to define any helper functions inside *add_digit.c* that should not be available outside, mark them as `static` to allow only internal linkage within this translation unit.

Step 1. Prepare your environment

Open your WSL Linux environment, prepare an empty *sandbox* directory where the development will take place, save the provided text files in this directory. Then create a new file *add_digit.c* and open it for editing.

Step 2. Investigate *add_digit.h*

Open *add_digit.h* and get familiar with the declarations of the `struct` type and functions that you must define in *add_digit.c*.

Step 3. Add file-level documentation

Add the file-level documentation to *add_digit.c*. Remember that every source code file you submit for grading must start with an updated file-level documentation header.

Step 4. Implement code for printing

Define `struct BigNumber` as shown at the beginning of this document. Then define the functions:

- `create_BigNumber()`
- `destroy_BigNumber()`
- `print_BigNumber_sum()`

Comment out from the *main.c* file the function `add_string()` and all test code related to it - this would allow you to test just the printing functionality first.

Then compile and test the code:

```
gcc -Wall -Werror -Wextra -Wconversion -Wstrict-prototypes -pedantic-errors -std=c11 -o main add_digit.c main.c
./main > actual-output.txt
diff actual-output.txt expected-output.txt --strip-trailing-cr
```

Work on the code until there are no differences between the actual and the expected outputs from the initial part of the output. Keep in mind that you still have not provided the code for performing addition.

Step 5. Implement code for addition

Restore the commented out code in *main.c*. Then define the function:

- `add_BigNumber()`

Again, compile and test the code. This time the entire output should match exactly. If needed, work on the code again until there are no differences between the actual and the expected outputs in each of the test cases.

Step 6. Add function-level documentation

Add the function-level documentation to every function in *add_digit.c*. Remember that every function in every source code file that you submit for grading must be preceded by a function-level documentation header that explains its purpose, inputs, outputs, side effects, and lists any special considerations.

Step 7. Clean up the code

Clean up the formatting, making sure it is consistent and easy to read. Break long lines of code; a common guideline is that no line should be longer than 80 characters.

Avoid repetition. If you see that multiple functions perform the same operation, or that certain statements logically represent distinct functionality, consider abstracting them away into separate helper functions; remember to make them `static`.

Step 8. Test the code

Once again, before submission, test the code. Your actual output must exactly match the contents of the expected output. Use the *diff* command to compare the files; no differences in the output should be reported.

Submission

Once your implementation of *add_digit.c* is complete, again ensure that the program works and that it contains updated file-level and function-level documentation comments.

Then upload the files to the laboratory submission page in Moodle. There are 92 lines of expected output and 92 corresponding automated test cases. To get the maximum grade, make sure that all test cases match.

If you're done early and are bored...

The program in its current form has two shortcomings:

- It does not produce values with leading zeros after addition, but when creating a `BigNumber` object from the text string, its value may include a leading zero (for example line 18 of the expected output).
- It does not handle negative numbers or other operations like subtraction.

Consider how to address these two challenges. **Do not submit this program.**