Laboratory exercise 7.

Working with pointers

Purpose of the exercise

This exercise will help you do the following:

- Practice working with pointer data types and pointer-related operators.
- Practice decomposition of code into smaller functions to make programming easier.

Overview

Decomposition

When we say a thing is *easy*, we mean that it feels familiar, known and that we are accustomed to working with things like it. Likewise, *difficult* means 'unfamiliar, untested'.

When we say a thing is *simple*, we mean that it is made of one part, as opposed to *complex* things, which are knotty and multi-layered; hard to break apart and analyze.

Computer programs in their nature are *difficult* to create, because often parts of their code were written by someone else, or by us too long ago to recall; such code does not feel familiar. They also get *complex*, because to include features and functionality we compose them of more parts and more code. Dealing with difficulty and complexity is a major challenge in programming.

In software development there are many techniques and practices we can include in our toolbox to get our job done efficiently. One of them is a concept of a **modular design**, or **decomposition**. You take a large problem that is hard to tackle and decompose it into a few smaller problems, some of which you know how to solve immediately, while the rest may need further decomposition until they become simple enough to solve. This **divide-and-conquer** approach gives you two important insights:

- You start with a general view first, getting the overall structure of the code.
- No problem is too big; you can either conquer it now, or divide for later implementation.

Abstraction

Alongside decomposition, another closely related technique in your toolbox is **abstraction** - a notion of giving a certain concept (a data type, a functionality, an object, etc.) a name, and hiding the implementation details somewhere else. By using just a name we can focus on its usage, not on an implementation of its individual parts.

If you are planning your code and thinking:

> "I need a function that returns the greatest of three numbers; I will name it '`find_greatest(a, b, c);`', insert a function call here and worry about its implementation later"

you are applying abstraction. For a moment you are ignoring the specifics of a definition; you look at the big picture and hide implementation details inside a function with a given name somewhere else in the code.

As for the big picture design, follow the KISS principle - "Keep it simple stupid".

Tasks

In this exercise you will implement functions helping you find the 3rd greatest integer number in an input sequence provided by the user; the resulting program will have to do it without arrays for storing elements. High-level flow of the program can be expressed using the pseudocode:

```
count ← read from the user
if (count < 3)
    print error message
    terminate program
end if
top1 ← read an integer from the user
top2 ← read an integer from the user
top3 ← read an integer from the user
sort(top1, top2, top3) so that
    top1 contains the greatest integer,
    top3 contains the smallest integer
count ← (count - 3)
while (count > 0)
    count ← (count - 1)
    number ← read an integer from the user
    insert number between top1, top2, top3 so that
        the sorting order is maintained
end while
print top3 as the 3rd largest integer
```

This pseudocode is reflected in the definition of the `main()` function in the driver *third_largest.c*.

Your task is to define the following functions to make this code work:

- `size_t read_total_count(void);`

  Reads in one integer from the user and tests if the value is less than 3. In such case it prints an error message and causes the entire program to exit by calling the function exit() - read the online reference or the textbook (pages 203 and 204) for more information.

- `void read_3_numbers(int* first, int* second, int* third);`

  Performs a simple task of reading in 3 integer values from the user and initializing the objects pointed by the function parameters. These parameters are meant to allow for side-effect output; programmers sometimes refer to such parameters as *output parameters*.

- `void swap(int* lhs, int* rhs);`

  Exchanges the values of the left-hand side object indicated by `lhs` and the right-hand side object indicated by `rhs`.

- `void sort_3_numbers(int* first, int* second, int* third);`

  Sorts the values of integer objects indicated by pointer parameters, so that when the function returns the largest value is in the object indicated by `first`, the second largest in the object indicated by `second`, and the third largest in the object indicated by `third`.

- `void maintain_3_largest(int number, int* first, int* second, int* third);`

  Assumes that objects indicated by `first`, `second`, and `third` have been already sorted and inserts a new value `number` into these objects as necessary, so that when the function returns the largest value is in the object indicated by `first`, the second largest in the object indicated by `second`, and the third largest in the object indicated by `third`; the fourth unnecessary value can be discarded.

These functions have been declared in *utils.h*; you must define them inside *utils.c*. You must observe the following constraints while implementing them:

- You can only declare variables of an `int` or `size_t` types with automatic storage - local variables.
- You cannot use any arrays.
- The function `sort_3_numbers()` must call `swap()`.

Take note that the type `size_t` returned from `read_total_count()` is an alias to the most appropriate unsigned integer type for counting elements on a machine of the current type; on a 64-bit machine it may represent `unsigned long long int`, which is also 64-bit. Prefer using it over a specific integer type for counting and indexing elements to make your code work across different platforms. Its corresponding format specifiers for `printf()` and `scanf()` are `%zu`, `%zo` and `%zx` for decimal, octal and hexadecimal representations respectively.

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 into this directory. Then create a new file *utils.c* and open it for editing.

Step 2. Complete *test0.c*

An important benefit of decomposition is ability to implement and test individual parts separately. In this exercise you have been provided with a few test drivers and pairs of input and expected output files (their file names will tell you which driver program they belong with).

2.1. Investigate the driver

Open *test0.c* and review its 4 input files and their corresponding output files. To compile and test this driver, you have to implement only one of the functions: `read_total_count()`. It is a good place to start your implementation.

2.2. Define `read_total_count()`

Comment out unnecessary function declarations in *utils.h*. Inside *utils.c* define the function `read_total_count()` needed by *test0.c*, compile and test the code:

```
gcc -Wall -Werror -Wextra -Wconversion -Wstrict-prototypes -pedantic-errors -std=c11 -o test0 test0.c utils.c
./test0 < input-test0-0.txt > actual-output-test0-0.txt
diff actual-output-test0-0.txt expected-output-test0-0.txt --strip-trailing-cr
./test0 < input-test0-1.txt > actual-output-test0-1.txt
diff actual-output-test0-1.txt expected-output-test0-1.txt --strip-trailing-cr
./test0 < input-test0-2.txt > actual-output-test0-2.txt
diff actual-output-test0-2.txt expected-output-test0-2.txt --strip-trailing-cr
./test0 < input-test0-3.txt > actual-output-test0-3.txt
diff actual-output-test0-3.txt expected-output-test0-3.txt --strip-trailing-cr
```

Work on the code until there are no differences between the actual and the expected outputs in each of the test cases.

Step 3. Complete *test1.c*

3.1. Investigate the driver

Open *test1.c* and review its 2 input files and their corresponding output files. To compile and test this driver, you have to implement only one more function: `read_3_numbers()`.

3.2. Define `read_3_numbers()`

Uncomment the declaration of the function `read_3_numbers()` in the *utils.h* header file, and define the function inside *utils.c*. Compile and test the code using the same approach as previously. Work on the code until there are no differences between the actual and the expected outputs in each of the test cases.

Step 4. Complete *test2.c*

4.1. Investigate the driver

Open *test2.c* and review its 2 input files and their corresponding output files. To compile and test this driver, you have to implement only one more function: `swap()`.

4.2. Define `swap()`

Uncomment the declaration of the function `swap()` in the *utils.h* header file, and define the function inside *utils.c*. Compile and test the code using the same approach as previously. Work on the code until there are no differences between the actual and the expected outputs in each of the test cases.

Step 5. Complete *test3.c*

5.1. Investigate the driver

Open *test3.c* and review its 4 input files and their corresponding output files. To compile and test this driver, you have to implement only more function: `sort_3_numbers()`.

5.2. Define `sort_3_numbers()`

Uncomment the declaration of the function `sort_3_numbers()` in the *utils.h* header file, and define the function inside *utils.c*. Compile and test the code using the same approach as previously. Work on the code until there are no differences between the actual and the expected outputs in each of the test cases.

Step 6. Complete *test4.c*

6.1. Investigate the driver

Open *test4.c* and review its 6 input files and their corresponding output files. To compile and test this driver, you have to implement only one more function: `maintain_3_largest()`.

6.2. Define `maintain_3_largest()`

Uncomment the declaration of the function `maintain_3_largest()` in the *utils.h* header file, and define the function inside *utils.c*. Compile and test the code using the same approach as previously. Work on the code until there are no differences between the actual and the expected outputs in each of the test cases.

Step 7. Compile and link *third_largest.c*

Now that you have implemented all required function definitions, it is time to check the main program of the exercise: *third_largest.c*. Compile and link the code:

```
gcc -Wall -Werror -Wextra -Wconversion -Wstrict-prototypes -pedantic-errors -std=c11 -o third_largest third_largest.c utils.c
```

There are 6 input files and corresponding output files. 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 8. Add file-level documentation

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

Step 9. Add function-level documentation

Add the function-level documentation to every function in *utils.c*. Remember that every function in every source code file 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 10. Clean up the code

Clean up the formatting, make 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.

Check if you have followed all the constraints of the exercise. Assess if you are properly reusing code by calling functions where appropriate, rather than repeating the same statements twice.

Step 11. Test the code

During grading, actual output files are generated for each test and all merged into a single *actual-output.txt* file using the Linux shell script provided as *build.sh* (due to its length it is not included in this document). The script uses the *diff* command to compare *actual-output.txt* with a provided *expected-output.txt*. To run the script in Linux run:

```
./build.sh
```

The script should produce no output, indicating no file differences.

On WSL by default the user who owns the script file (`u`) should have a required permission for execution (`x`), but in case you have to add it manually, you can use the following shell command:

```
sudo chmod u+x ./build.sh
```

Submission

Once your implementation of *utils.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 25 lines of expected output and 25 corresponding automated tests cases. To get the maximum grade, make sure that all test cases match.

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

The exercise requires you not to use arrays in your implementation. For practice, try to implement an alternative version of *third_largest.c* that solves the same problem but firstly reads in all inputs into an array and sorts them using an algorithm of your choice.

Do not submit this program.