Lab 7: Working with Pointers

Learning Outcomes

- 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 or 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 by their very 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. Programs also get *complex*, because to include features and functionality we compose them of many parts and then we become more ambitious and then add more features which in turn requires more code. Dealing with difficulty and complexity is a major challenge in programming.

In software development there are many techniques and practices we include in our toolbox to get our job done efficiently. One of them is the concept of **modular design**, or **decomposition**. You take a large problem that is hard to tackle and decompose it into 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 its implementation.

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.

Tasks

In this exercise you will implement functions helping you find the 3^{rd} 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 first integer from the user
top2 ← read second integer from the user
top3 ← read third integer from the user
sort(top1, top2, top3) so that
    top1 now contains the greatest of the three integers
    top3 now contains the smallest of the three integers
count ← (count - 3)
while (count > 0)
    count ← (count - 1)
    number ← read an integer from the user
    update top1, top2, and top3 with number so that
        the sorting order is maintained
end while
print top3 as the 3rd largest integer
```

This pseudocode is reflected in the definition of function `main` function in driver $\texttt{qdriver.c}$.

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 program to exit by calling C standard library function `exit` [read the online reference for more information on function `exit`. Notice the strange type `size_t` in the declaration of function `read_total_count`.

  > Think of `size_t` as a new name for an existing type that is provided by the C standard library to facilitate portable C programs across a variety of CPUs. From that perspective, type `size_t` is an alias for some unsigned integer type, typically `unsigned int` or `unsigned long`, or even `unsigned long long`. Each C standard implementation is supposed to choose the unsigned integer that's big enough – but no bigger than needed – to represent the size of the largest possible unsigned integral value on the target platform. Your $64-$bit implementation of GCC will evaluate `sizeof(size_t)` to 8. Since `sizeof(unsigned long)` also evaluates to 8, your GCC implementation will define `size_t` as an alias for type `unsigned long`. The standard library provides the definition of type `size_t` in a number of standard library headers including `<stdio.h>`, `<stddef.h>`, `<string.h>`, and `<time.h>`.
  >
  > By definition, `size_t` is the type of the integral result evaluated by `sizeof` operand. Thus, the appropriate way to declare `x` to make the assignment `x = sizeof(some_thing)` both portable and efficient is to declare `x` as type `size_t`. Similarly, the appropriate way to declare a function `foo` to make the call `foo(sizeof(some_thing))` both portable and efficient is to declare `foo`'s parameter as type `size_t`. Functions with parameters of type `size_t` often have local variables that count up to or down from that size and index into arrays, and `size_t` is often a good type

> for these variables. Using `size_t` appropriately makes your code a more literate and self-documented. When you see an object declared as `size_t`, you immediately know it represents a size in bytes or an index, rather than a general arithmetic value. Further, using `size_t` for indexing or representing sizes makes your code work across different platforms.
>
> Since your current GCC implementation defines `size_t` to be an alias for type `unsigned long int`, you can use format specifier `%lu` in `printf` and `scanf` to read and write values of type `size_t`. If one doesn't know the type that `size_t` is an alias for, `printf` and `scanf` provide format specifiers `%zu`, `%zo` and `%zx` for writing and reading values of type `size_t` in decimal, octal and hexadecimal representations respectively.

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

  Performs the simple task of reading 3 integer values from the user and storing these integer values in `int` objects pointed to by the function parameters. As the function parameter names indicate, the 1^{st} integer read from the standard input stream is assigned to the `int` object pointed to by parameter `first`, and the 2^{nd} integer read from the standard input stream is assigned to the `int` object pointed to by parameter `second`, and so on. The three parameters are referred to as *output parameters* since they only serve the purpose of extracting information from function `read_3_numbers`.
- `void swap(int *lhs, int *rhs);`

  Exchanges the values of the left-hand side object pointed to by parameter `lhs` and the right-hand side object pointed to by parameter `rhs`.
- `void sort_3_numbers(int *first, int *second, int *third);`

  Sorts the `int` objects pointed to by the parameters, so that the object with the largest `int` value is now pointed to by parameter `first`, the object with the second largest `int` value is now pointed to by parameter `second`, and the object with the third largest `int` value is now pointed to by parameter `third`.
- `void maintain_3_largest(int number, int *first, int *second, int *third);`

  Assumes the `int` objects pointed to by parameters `first`, `second`, and `third` are sorted in increasing order. The function compares `number`'s value to the `int` values in the three objects [pointed to by the parameters] and if necessary assigns `number`'s value to one of these three objects so that the relative ordering of the `int` objects [pointed to by the parameters] is maintained when the function returns.

Declare these functions in header file $\texttt{q.h}$ and define these functions in source file $\texttt{q.c}$, while observing the following constraints:

- You cannot use any arrays. The online grader will not accept any submission containing tokens `[` or `]`.
- Function `sort_3_numbers` must call function `swap`.

Take note that type `size_t` in function prototype of `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.

Step 1. Prepare your environment

Open a Window command prompt, change your directory to C:\sandbox [create the directory if it doesn't exist], download archive file lab07.zip, unzip the downloaded archive file as directory lab07. In directory lab07, you will find a Bash script file build-and-test.sh, an output file output-all.txt [more on these later], and a sub-directory tests containing this assessment's framework of drivers and test files. Finally, launch the Linux shell in directory lab07.

Step 2. Add file-level documentation

In directory lab07 create and open for editing a header file q.h and a source file q.c. Add necessary file-level documentation to header file q.h and source file q.c. Remember that every source code file you submit for grading must start with an updated file-level documentation header.

An important benefit of decomposition is the ability to implement and test individual parts separately. We will use this idea to independently define and test the necessary functions.

Step 3. Define function `read_total_count`

Review driver source file ./tests/test0.c. You will notice that function `main` in the driver source file makes a call to function `read_total_count` [that you will define]. Further, review input files ./tests/input-test0-?.txt containing input values that will be used to test your definition of function `read_total_count` and corresponding output files ./tests/output-test0-?.txt containing the correct outputs.

After examining the driver source file, input and corresponding output files, begin by providing a declaration for function `read_total_count` in header file q.h. Devise an algorithm for function `read_total_count` and document it in its function-level header in q.h for use by your clients. Define function `read_total_count` in source file q.c. Add function-level documentation of your strategies in implementing this function as future notes for yourself and other programmers [in case you need to later debug this function]. Compile and link to create a program ./tests/test0.out:

```
$ gcc -Wall -Werror -Wextra -Wconversion -Wstrict-prototypes -pedantic-errors -std=c11
./tests/test0.c q.c -o ./tests/test0.out
```

Now, test your definition of function `read_total_count` with values in each of the input files:

```
$ ./tests/test0.out < ./tests/input-test0-0.txt > ./tests/your-output-test0-0.txt
$ ./tests/test0.out < ./tests/input-test0-1.txt > ./tests/your-output-test0-1.txt
$ ./tests/test0.out < ./tests/input-test0-2.txt > ./tests/your-output-test0-2.txt
$ ./tests/test0.out < ./tests/input-test0-3.txt > ./tests/your-output-test0-3.txt
```

Compare the output values generated by your definition of function `read_total_count` with the correct values:

```
$ diff -y --strip-trailing-cr --suppress-common-lines ./tests/your-output-test0-0.txt
./tests/output-test0-0.txt
$ diff -y --strip-trailing-cr --suppress-common-lines ./tests/your-output-test0-1.txt
./tests/output-test0-1.txt
$ diff -y --strip-trailing-cr --suppress-common-lines ./tests/your-output-test0-2.txt
./tests/output-test0-2.txt


$ diff -y --strip-trailing-cr --suppress-common-lines ./tests/your-output-test0-3.txt
```

```
./tests/output-test0-3.txt
```

Continue to edit the definition of function `read_total_count` until there are no differences between the actual and the expected outputs in each of the test cases.

Step 4. Define function `read_3_numbers`

Review driver source file ./tests/test1.c. You will notice that function `main` in the driver source file makes a call to function `read_3_numbers` [that you will define]. Further, review input files ./tests/input-test1-?.txt containing input values that will be used to test your definition of function `read_3_numbers` and corresponding output files ./tests/output-test1-?.txt containing the correct outputs.

After examining the driver source file, input and corresponding output files, begin by providing a declaration [and function-level documentation for use by your clients] for function `read_3_numbers` in header file q.h. Next, add function-level documentation and the definition of function `read_3_numbers` in source file q.c.

Use the same approach as in the previous step to test your definition of function `read_3_numbers`.

Step 5. Define function `swap`

Review driver source file ./tests/test2.c. You will notice that function `main` in the driver source file makes a call to function `swap` [that you will define]. Further, review input files ./tests/input-test2-?.txt containing input values that will be used to test your definition of function `swap` and corresponding output files ./tests/output-test2-?.txt containing the correct outputs.

After examining the driver source file, input and corresponding output files, begin by providing a declaration [and function-level documentation for use by your clients] for function `swap` in header file q.h. Next, add function-level documentation and the definition of function `swap` in source file q.c.

Use the same approach as in the previous step to test your definition of function `swap`.

Step 6. Define function `sort_3_numbers`

Review driver source file ./tests/test3.c. You will notice that function `main` in the driver source file makes a call to function `read_3_numbers` [that you have previously defined] and a call to function `sort_3_numbers` [that you will define]. Further, review input files ./tests/input-test3-?.txt containing input values that will be used to test your definition of function `sort_3_numbers` and corresponding output files ./tests/output-test3-?.txt containing the correct outputs.

After examining the driver source file, input and corresponding output files, begin by providing a declaration [and function-level documentation for use by your clients] for function `sort_3_numbers` in header file q.h. Next, add function-level documentation and the definition of function `sort_3_numbers` in source file q.c.

Use the same approach as in the previous step to test your definition of function `sort_3_numbers`.

Step 7. Define function `maintain_3_largest`

Review driver source file ./tests/test4.c. You will notice that function `main` in the driver source file calls functions `read_3_numbers` and `sort_3_numbers` [that you have previously defined] and a new function `maintain_3_largest` [that you will define]. Further, review input files ./tests/input-test4-?.txt containing input values that will be used to test your definition of function `maintain_3_largest` and corresponding output files ./tests/output-test4-?.txt containing the correct outputs.

After examining the driver source file, input and corresponding output files, begin by providing a declaration [and function-level documentation for use by your clients] for function `maintain_3_largest` in header file q.h. Next, add function-level documentation and the definition of function `maintain_3_largest` in source file q.c.

Use the same approach as in the previous step to test your definition of function `maintain_3_largest`.

Step 8. Final tests

Now that you have implemented all required function definitions, it is time to test them again together using a driver ./tests/qdriver.c. Compile and link your definitions in q.c with the driver code and the standard library.

The test data is located in input files ./tests/input-test5-?.txt and the correct output in corresponding output files ./tests/output-test5-?.txt. Use the same approach as in the previous steps to test all your definitions of the necessary functions.

Step 9. Review contents of your files

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.

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 10. Automating test using Bash script

The Bash command shell is an interactive interpreter for file manipulation and this is what you are using to interact with files and directories in Linux. It can also be programmed. However, the Bash programming language is quite unintuitive and unforgiving with a strange syntax. You can view a basic version of the language in the script build-and-test.sh [that you can download from the assessment webpage]. It generates each of the executables involved in steps 3 through 8. It concatenates the resulting output into a single file your-output-all.txt. The script ends after using command diff to compare your-output-all.txt with the provided output-all.txt. Run the script in your Linux shell like this:

```
$ bash build-and-test.sh
```

If your implementation is perfect, your output file your-output-all.txt and the provided output-all.txt will match exactly, and the script will produce no output.

In Linux, 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-and-test.sh
```

Submission and automatic evaluation

1. In the course web page, click on the submission page to submit files q.h and q.c.
2. Read the following rubrics to maximize your grade. Your submission will receive:
    1. F grade if your submission doesn't compile with the full suite of gcc options [shown above].

2. F grade if your submission doesn't link to create an executable.
3. $A+$ grade if the submission's output matches the correct output. Otherwise, a proportional grade is assigned based on how many incorrect results were generated by your submission.
4. A deduction of one letter grade for each missing documentation block. Every submitted file must have one file-level documentation block. Every function that you declare in a header file must provide a function-level documentation block. A teaching assistant will physically read submitted source files to ensure that these documentation blocks are authored correctly. Each missing block will result in a deduction of a letter grade. For example, if the automatic grader gave your submission an $A+$ grade and the three documentation blocks are missing, your grade will be later reduced from $A+$ to $B+$. Another example: if the automatic grade gave your submission a C grade and the three documentation blocks are missing, your grade will be later reduced from C to F.