Laboratory exercise 4.

Formatted console input/output

Purpose of the exercise

This exercise will help you do the following:

1. Practice developing programs that correctly handle invalid user input.
2. Practice developing programs that properly format printed output.
3. Develop familiarity with statements other than just expression statements.

Overview

In **Lab 3** you were required to complete a C program by writing a function that takes in a total amount of money and performs a breakdown of the amount into the least number of coins.

In this exercise we will revisit this task with a few modifications that make the program more user-friendly, and at the same time more incapable of being misused. We will achieve these effects by scanning the input for a correct format, printing the console output with appropriate formatting as well, and using selection statements to perform input validation.

Before we focus on the breakdown problem again, we need to explore the behavior of the `scanf()` function first.

Exploring `scanf()`

We have been using *functions* in our course since our first program. Functions are one of the simplest *abstractions* - that is they let you hide the details of implementation somewhere else in the code and focus on calling them by their identifier, that is their name, without asking each time how they are implemented. When we look at a call to the `printf()` function, we think about displaying the text to the standard output, and not the function's possible implementations, which may not be even available to us programmers. Functions hide statements that do the work. We think about them as black boxes which we provide with zero, one or more inputs called parameters or arguments, observe their side effects, eg. text being printed out on the user interface, and optionally capture their returned output.

The function `scanf()` is a great example:

```
int x, y, z;
scanf("%d-%d-%d", &x, &y, &z);
```

Here, the function `scanf()` is called with 4 arguments: a format string and 3 addresses of variables `x`, `y` and `z` respectively. Do not worry about the word *address*; we will cover it later in this course. This function exhibits side effects which, in this case, facilitate user input. `scanf()` also returns a value. Let's investigate what this function returns by considering the following program:

```c
#include <stdio.h>

int main(void)
{
    int x, y, z;
    int r = scanf("%d-%d-%d", &x, &y, &z);
    printf("%d\n", r);
    return 0;
}
```

Replicate this code as a new valid program, compile it, link it and execute it with the following inputs:

- `34-59-9`
- `34-*9-9`
- `-35 -99`

Pay attention to spaces and all other individual characters. Consider what is the meaning of the output of `scanf()` assigned into `r` in each observed case. If you need to, test with other inputs.

You do not need to submit anything from this part, but this short and simple exercise can be instrumental to completing the second part related to the coin breakdown.

Coin breakdown

In the **Lab 3** we focused on breaking down monetary value into cash denominations. This time, we will add a few modifications to make this program more interesting.

Firstly, you will perform breakdown into Singapore coins and also frequently used banknotes' denominations as follows: 100 dollars, 50 dollars, 10 dollars, 5 dollars, 2 dollars, 1 dollar, 50 cents, 20 cents, 10 cents, 5 cents. We will also use the 1 cent denomination for this task.

As before you will capture the input in a format `d.c`, where `d` represents non-negative number of dollars in the value, while `c` represents the cents part, which must be in a range from 0 to 99 (inclusive). For example, a value of 12 dollars and 34 cents will be expected as the input `12.34`.

Thirdly, this time you need to gracefully handle incorrect input:

- When the user inputs only the dollar part and the period without the cents, the program must show an error message:

  ```
  You did not type in the correct format in terms of dollars and cents.
  ```

- When the user inputs a negative dollar part, the program must show an error message:

  ```
  The dollars part specified must be non-negative.
  ```

- When the user inputs the cents part outside of the range from 0 to 99, the program must show an error message:

  ```
  The cents part specified must be between 0...99 (inclusive).
  ```

To keep things simple, if a user inputs the cents part only with a single digit (i.e. `0.5`) the value should be recognized as the number of cents (as *5 cents*), and not as a fraction of a dollar (not as *50 cents*).

Lastly, when the input is captured correctly, the program needs to print the breakdown. This time it must do so in a tabular form. For example, when the user enters `12.34`, the expected output should look as shown below:

```
+----+--------------+-------+
| #  | Denomination | Count |
+----+--------------+-------+
| 1  |       100.00 |     0 |
| 2  |        50.00 |     0 |
| 3  |        10.00 |     1 |
| 4  |         5.00 |     0 |
| 5  |         2.00 |     1 |
| 6  |         1.00 |     0 |
| 7  |         0.50 |     0 |
| 8  |         0.20 |     1 |
| 9  |         0.10 |     1 |
| 10 |         0.05 |     0 |
| 11 |         0.01 |     4 |
+----+--------------+-------+
```

The table must contain 3 columns:

1. Left aligned index of the current row.
2. Right aligned denomination of a coin or a banknote.
3. Right aligned count of used coins or banknotes in a given denomination.

While implementing this task you have to observe the following constraints:

- Split the code into 3 functions:
  - `print_line(index, cents, value)` - the function prints a single line of the output table; it takes in `index` (to be displayed in the first column), the value `cents` remaining for breakdown (the breakdown is shown in the third column), and `value` corresponding to the denomination used in the current line; it returns no output.
  - `coins(cents)` - the function prints the header of the output table, all rows (by calling `print_line()`) and the borders of the table; it takes in the value `cents` for breakdown; it returns no output.
  - `main(void)` - the entry point of the program; handles all user input and input validation; it does not take in any input and always returns `0`.
- Do not include any header files except *stdio.h*.
- Use only `int` and `void` data types.

This part will require you to submit the entire program in a single *q.c* file. For the details of this task go through the steps listed in the section below.

Task 1

In this exercise you will write three programs. The first one, for exploring the result of `scanf()` function will not be submitted. You will need to submit only the last two programs which offer improved money breakdown functionality.

Step 1. Prepare your environment

Open your WSL Linux environment, prepare an empty *sandbox* directory where the development will take place, create a new file *s.c* and open it for editing:

```
touch s.c
code s.c
```

These commands should open Microsoft Visual Studio Code for editing your code.

Step 2. Copy code to explore `scanf()`

Copy the code shown above in the section **Exploring** `scanf()` into the file *s.c*. Make sure that you copy by hand to develop practice of writing the code. Copy-paste is always faster, but it also requires no mental work, and so does not bring any educational value.

Step 3. Compile and test the code

This entire program contains a single translation unit. You can compile it and link it with the following single command:

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

If there are any compilation or linking errors, correct them and try again. Assuming clean compilation without any diagnostic messages, an executable has been created. Run the executable and observe the output as you provide different input sequences. Make sure that you understand the behavior of the function `scanf()`.

If you seek more information, examine the function's reference documentation (find a section "Return value"; as the page covers the entire family of related functions, focus on a version "*(1)*" only):
https://en.cppreference.com/w/c/io/fscanf.

If you successfully completed this part of the exercise, move forward to the main task in **Step 4**. Take note that the steps you have completed so far did not result in any code you have to submit.

Step 4. Clean-up your environment

Prepare your *sandbox* directory for development of your second program. You could do it from the Microsoft Windows GUI with *File Explorer*, but it may be a better choice to practice your skills in using Linux CLI.

Use the command rm to remove files such as the executable *main* of the first program, or mkdir and mv create a new directory and move the files you want to preserve there to keep the *sandbox* folder empty.

Then create a new file *q.c* and open it for editing:

```
touch q.c
code q.c
```

These commands should open Microsoft Visual Studio Code for editing your code.

Step 5. Add file-level documentation

In the **Laboratory exercise 3.** you have learnt that every source code file you submit for grading must start with a file-level documentation header. Open the specification of that exercise, and use the template of a header provided there in your new file. Replace `@todo` and the rest of each line with your information. When you edit the file later, remember to check if the information is up-to-date.

Step 6. Include required headers

To use the functions `printf()` and `scanf()` we need to include a header that contains their prototypes. These functions have been declared in a header *stdio.h* provided by a C compiler vendor. You have to include this header in *q.c* using the preprocessor directive `include` like so:

```
#include <stdio.h>      // printf, scanf
```

The single line comment included above gives us a valuable information about why the header was included; do not forget to add it too.

Step 7. Prepare function stubs

As this exercise requires you to split the code into 3 functions, start your implementation by adding their *stubs* - empty function blocks acting as placeholders:

```
void print_line(int index, int cents, int value)
{
    // TODO: provide proper implementation later
    printf("print_line(%d, %d, %d)\n", index, cents, value);
}

void coins(int cents)
{
    // TODO: provide proper implementation later
    printf("coins(%d)\n", cents);
}

int main(void)
{
    // TODO: provide proper implementation later
    printf("main()\n");
}
```

Before progressing to the next part, make sure that you fully understand the purpose of each of these functions. If it is not clear, review the **Coin breakdown** section again, and pay particular attention to the expected output format.

Step 8. Add function-level documentation

Since all functions in this exercise are well defined, you are ready to write their function-level documentation.

In the **Laboratory exercise 3.** you have learnt 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. Open the specification of that exercise, and review the example of a function header documentation provided there; add relevant function-level documentation for all functions in *q.c*.

Step 9. Compile the code for the first time

The program does not perform its intended role yet, but starts to have a proper structure. For a good start, try to compile it and execute it:

```
gcc -std=c11 -Wall -Werror -Wextra -Wconversion -pedantic-errors -Wstrict-prototypes -o main q.c
./main
```

You should see that the function `main()` has been executed, but as the functions have not been connected yet by function calls, it is clear that the code needs more work.

Step 9. Design the solution

Before you write any additional statements, focus for a moment and imagine a sample scenario of a basic interaction of a user with the program. Visualize what operations the user and the program will perform, and try to envision how the control of execution will flow from one function to another. Do you know where does the execution start? Which function will call which other function and whether it will happen once or repeatedly?

If you are not sure, consider formalizing this design through an algorithm; perhaps a flowchart diagram or pseudo code would help you plan your work better. This algorithm will roughly translate to the following steps:

- Take in the user input.
- Validate the user input.
  - If the user input is incomplete, print a relevant message.
  - If the user input is out of range, print relevant messages.
  - If the user input is valid, perform the breakdown:
    - Calculate coins and banknotes at each denomination.
    - Print the output:
      - Print the borders.
      - Print the header.
      - Print the borders.
      - Print a line for `100.00`.
      - Print a line for `50.00`.
      - ...
      - Print the borders.

Step 10. Edit-compile-link-run cycle

It is time to enter the development cycle.

Recall that the *edit-compile-link-run* cycle helps us give a structure to an incremental development process. We want to be able to make small changes and to execute the code immediately for testing.

It is a good practice to keep code uncompilable for as short as possible. We can achieve it by developing easily testable code first. If we start by implementing the code to take in the user input, to test it we would have to develop more code for printing the values as well. On the other hand, we can defer implementation of user input handling for later and start from the last part. By printing out the output for some predefined, known input, we should be able to test the code immediately.

Make a mental note to expand the code in the following steps:

1. From `main()` call `coins()` with an arbitrary value of `cents` (for now you can ignore it; later you will replace it with a meaningful value), and from `coins()` call `print_line()` with some arbitrary values of `index`, `cents`, and `values`. This gives our program a flow similar to the final program.
2. Complete the definition of `print_line()` so that for given inputs it prints a single well-formatted line into the output stream.
3. Modify the definition of `coins()` so that it prints a well-formatted table into the output stream, even if the numeric values are not correct.
4. Complete the definition of `coins()` so that it prints a well-formatted table with correct values.
5. Modify the definition of `main()` so that it captures the user input and without validation passes it to `coins()`.
6. Complete the definition of `main()` so that it performs all required validation.

Step 11. Define `print_line()`

This function has only a single purpose: to print a correctly formatted line from a table.

Use the formatting options of the function `printf()` to achieve a desired result with a single function call. Remember that a general format for a conversion specifier in `printf()` is: `%[flags][min_width][.precision]specifier`.

Each line contains the borders and 3 table cells with values. The borders of the table are created using a combination of pipe (`|`) and plus (`+`) ASCII characters. There is a single space (` `) character of fixed padding at each side of a table cell. All additional padding results from the use of appropriate formatting specifiers.

As mentioned earlier in this document, the columns with values must use the following formatting:

- **Column 1** must be aligned to the left with a sufficient space for 2 digits.
- **Column 2** must be aligned to the right with a sufficient space for 9 digits of the integer part of the value representing dollars, a single character of a decimal separator (`.`) and 2 decimal positions representing cents. Take note that cents must show the leading 0 if they are expressed with a single digit value.
- **Column 3** must be aligned to the right with a sufficient space for 5 digits.

If you do not remember available options, you can check the reference documentation (focus on a version *(1)* only):
https://en.cppreference.com/w/c/io/fprintf.

Step 12. Define `coins()`

This function prints the entire table. Except for the formatting used, it will contain a similar logic to the one included in **Programming Assignment 2**, that is, it will calculate the notes or coins in each denomination and output them to the standard output.

Replace the placeholder code with the code that uses `printf()` to print out the table header that has a fixed structure and formatting. This never changes.

Next, add the code that calls `print_line()` for each listed denomination. You may need to declare an additional local variable to keep track of the index of each row, and increment it after every line.

Lastly, print the bottom border.

Step 13. Capture user input in `main()`

At this stage, the `coins()` function should be correctly displaying a breakdown for any value that is passed into `coins()` inside the function `main()`. Now you can pass a monetary value entered by the user.

Add the `printf()` statement to gracefully prompt the human user for input. Follow it with the `scanf()` statement to capture the input. You will need variables to store the input values.

Rather than capturing the input as a real number like `float` or `double` you may want to scan for a period-separated pair of two `int` numbers for dollars and cents respectively and calculate the total value in cents yourself. Using floating-point precision numbers may lead to precision related rounding errors in numbers with multiple significant digits, and when you handle money it must add up exactly! When money is at stake use fixed-point precision values.

If you performed this step correctly, you should be able to see a correct output for every valid input provided by the user.

Step 14. Validate user input in `main()`

Never trust the user input. Users make accidental mistakes, they may act in a malicious way, or may not be aware of requirements or expectations of a program. Every user input must be validated.

Basic validation is achieved by using a proper format specifier in a call to `scanf()` which constrains data type and width. It is still the programmer's responsibility to check for completeness of input, validity of data ranges, consistency, including with other data, and compliance with specific business constraints.

In this exercise you have to perform three validation checks:

- Ensure that the user input was captured in all necessary variables.
- Check if the number of dollars is non-negative.
- Check if the number of cents is between 0 and 99 inclusive.

To capture different cases, you may need to include selection statements, as well as logical and comparison operators. You will also have to check the data input using the return value of `scanf()`.

For example, if we assume that a variable `cents` represent the amount of cents from the user input, the last rule can be implemented with the following code:

```
if ((cents < 0) || (99 < cents))
{
    // Input is invalid; cents amount is out of range.
}
else
{
    // Input is valid.
}
```

For your convenience, the links to the relevant reference documentation have been provided below.

Selection statements (`if`, `if`/`else`) let you conditionally execute a branch of code. A complete reference documentation is available at:
https://en.cppreference.com/w/c/language/if

Logical operators (logical *NOT* `!`, logical *AND* `&&`, and logical *OR* `||`) apply *boolean* algebra on their operands, considering them `false` for values equal to 0, and `true` otherwise. They return `int` value `1` if the result is `true`, and `0` otherwise. Complete reference documentation is available at:
https://en.cppreference.com/w/c/language/operator_logical

Comparison operators like equal `==`, not equal `!=`, less than `<`, greater than `>`, less than or equal to `<=`, and greater than or equal to `>=` return `int` value `1` if the result is `true`, and `0` otherwise. Complete reference documentation is available at:
https://en.cppreference.com/w/c/language/operator_comparison

Take note that logical and comparison operators actually return a value of a data type `int`, not `bool`. This is because until *c99* the programming language did not offer a boolean data type, but rather used integers of zero/non-zero values to represent `true` and `false`.

Step 15. Test

Compile and link your source file *q.c* using the full suite of required gcc options:

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

Then run the executable:

```
./main
```

Test a few examples entered by hand and check if the output is properly formatted and matches your expectations.

In this exercise you have been provided with files *input-00.txt* to *input-09.txt*. Execute the program with each input file, appending the output to a file *actual-output.txt*:

```
./main < input-00.txt >  actual-output.txt
./main < input-01.txt >> actual-output.txt
./main < input-02.txt >> actual-output.txt
./main < input-03.txt >> actual-output.txt
./main < input-04.txt >> actual-output.txt
./main < input-05.txt >> actual-output.txt
./main < input-06.txt >> actual-output.txt
./main < input-07.txt >> actual-output.txt
./main < input-08.txt >> actual-output.txt
./main < input-09.txt >> actual-output.txt
```

Your are also given an output file *expected-output.txt* that contains the correct result. Your actual output must exactly match the contents of the expected output. Use the *diff* command in the Linux bash shell to compare the output files:

```
diff --strip-trailing-cr -y --suppress-common-lines actual-output.txt expected-output.txt
```

If *diff* completes the comparison without generating any output, then the contents of the two files are an exact match.

Submission

Once your implementation of *q.c* is complete, again ensure that the program works and that it contains updated file-level and function-level documentation comments. Upload the file *q.c* to the laboratory Task 1 submission page in Moodle.

There are 71 lines of expected output and 71 corresponding automated test cases. To get the maximum grade, make sure that all test cases match. In addition, there are 4 expected documentation blocks, one file-level and three function-level.

Task 2

Copy *q.c* to a new file called *q2.c*. In *q2.c* add a loop to `main` that repeatedly prompts the user for input and prints the corresponding cash denominations. The loop must keep running till the user input for the amount of money is invalid. Here, entering invalid input is a way of signaling to the program that the user is done entering values to be converted into currency denominations. On invalid input, the program does not print an error message but prints `Program ended` with a newline character `'\n'` at the end:

```
Program ended
```

You may see the test input and expected output under Task 2 in Moodle. Submit *q2.c* using the Task 2 submission page.

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

There are two elements of this assignment you can investigate:

- See how the loss of precision in floating-point precision numbers looks in action by capturing the user input into a single `float` variable and providing a number with more than six significant digits. Your validation code and printing code may need adjustments.
  Do not submit this code!

- Force the number of cents to always be provided as a valid two digit number, such that the input `12.3` would be recognized as incorrect.
  Do not submit this code!