Lab 4: While Loops, Formatted I/O

Learning Outcomes

- Practice developing programs that correctly handle invalid user input.
- Practice developing programs that properly format printed output.
- 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.

Denomination breakdown

In **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.

This time you are to 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 consist of two digits in a range from `00 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:

- If the user does not enter the input as required, for example if the user inputs only the dollar part and the period without two digits for the cents part right after the decimal point, 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:

  ```
  You did not type in the correct format in terms of dollars and 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 |
+----+--------------+-------+
```

Note that we use the largest denomination possible before moving to the next largest one. 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, rest, denomination)` - This function prints a single line of the output table; it takes in 1. `index` to be displayed in the first column; 2. the parameter `rest` which is the number of cents remaining for break-down into denominations; and 3. `denomination` corresponding to the denomination in cents that is currently being considered. `print_line` returns nothing.
    - `break_down(cents)` - This function takes in `cents` which is the value in cents to be broken down into the various denominations. It prints the header of the output table, a row for each denomination by calling `print_line`, and the borders of the table. `break_down` returns `void`.
    - `go(void)` - This function handles all user input for the amount of money to break-down into denominations. It also performs input validation and calls `break_down` to execute the break-down and print out the results.
    - `main(void)` - This is the entry point of the program that does not take in any parameters. It calls a function `go()` for the money to break-down into denominations.
    - Define other functions if any as required for your implementation.
- Include `<stdbool.h>`. and use `bool` `true` `false` macros at appropriate places in your code.

In this exercise you will have two tasks. You will need to submit `q.c` and `q.h` files for only the second task offering money breakdown functionality.

Task-1

Step 1. 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 new *q.c* and *q.h* files and open them for editing, like so:

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

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

Step 2. Add file-level documentation

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

Step 3. 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 valuable information about why the header was included; do not forget to add it too.

Step 4. 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 in `q.c` - in the order below:

```
void print_line(int index, int rest, int denomination)
{
    // @todo later: do implementation and remove the next lines -
    printf("print_line(%d, %d, %d)\n", index, rest, denomination);
}

void break_down(int cents)
{
     // @todo later: do implementation and remove the next lines -

    printf("break_down(%d)\n", cents);
    print_line(1, 100, 2);
}

void go(void)
{
    // @todo later: do implementation and remove the next lines -
    printf("go()\n");
    break_down(1000);

}

int main(void)
{
     // @todo later: do implementation and remove the next lines -
    printf("main()\n");
    go();
}
```

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 **Denomination breakdown** section again, and pay particular attention to the expected output format.

Step 5. Add function-level documentation

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

In **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. Add relevant function-level documentation for all functions in *q.c*.

Step 6. 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 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 completed and connected by function calls, it is clear that the code needs more work.

Step 7. Design the solution

Before you write any additional statements, focus for a moment and imagine a sample scenario of interaction between a user and 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 would call which other function and would this happen once or repeatedly?

If you are not sure, consider formalizing this design through an algorithm; perhaps a flowchart diagram or pseudocode would help you plan your work better. The algorithm is roughly translated into the following steps:

- Take in the user input
- Validate the user input
    - If the user input does not comply with the required format, print a relevant error message
    - If the user input is valid -
        - Print a border
        - Print the column headers
        - Print a border
        - Call `print_line` with `denomination` of $100, i.e. 10,000 cents. `print_line`:
            - Calculates the number of $100 bills
            - Prints a line in the table for $100 bills
        - Call `print_line` with `denomination` of $50, i.e. 5,000 cents. `print_line`:
            - Calculates the number of $50 bills
            - Prints a line in the table for $50 bills
        - Call `print_line` with `denomination` of $10, i.e. 1,000 cents. `print_line`:
            - Calculates the number of $10 bills
            - Prints a line in the table for $10 bills
        - ...
        - Call `print_line` with `denomination` of 1 cent. `print_line`:
            - Calculates the number of 1 cent coins
            - Prints a line in the table for 1 cent coins
        - Print a border

Step 8. 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 of an incremental development process. We want to be able to make small changes and execute the code immediately for testing.

It is 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 `go()` . From `go()` call `break_down()` with a hard coded value of `cents`. From `break_down()` call `print_line()` with some arbitrary values of `index`, `rest`, and `denomination`. This gives our program a flow similar to the final program.

2. Complete the definition of `print_line()` so that for given inputs it calculates the number of bills or coins and prints a single well-formatted line to the output stream.
3. Modify the definition of `break_down()` so that it prints a well-formatted table to the console even if the numeric values are not yet correct.
4. Complete the definition of `break_down()` so that it prints a well-formatted table with correct values.
5. Modify the definition of `go()` so that it captures the user input and performs validation, passes it to `break_down()`.
6. Complete the definition of `main()` so that it calls `go`.

Step 9. Define `print_line()`

This function has to calculate the number of bills or coins and print a correctly formatted line of the 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`, where each field enclosed in `[` and `]` is optional.

Each line contains 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 on 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 for the integer part of the value representing dollars. There must be 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 sufficient space for 5 digits.

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

Step 10. Define `break_down()`

This function prints the entire table by calling `print_line` repeatedly, each time with a new `index`, remaining value in `rest`, and `denomination`. Replace the placeholder code with the code that uses `printf()` to print out the table header.

Next, add the code that calls `print_line()` for each denomination. Lastly, print the bottom border.

Step 11. Capture user input in `go()`

At this stage, the `break_down()` function should be correctly displaying a break-down for a value passed in. Now you can add code to use a value entered by the user.

Add the `printf()` statement to gracefully prompt the human user for input. Follow it with the `scanf()` statement. 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 an `int` for the number of dollars followed by calls to `getc` to examine the characters following the dollar amount. This would allow you to validate the input as mentioned in the next section. Don't handle the monetary values as floating-point precision numbers as this may lead to precision related rounding errors, 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 correct output for every set of valid inputs provided by the user.

Step 12. Validate user input in `go()`

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 the following validation checks:

- Ensure that the user input was captured in all necessary variables.
- Check if the monetary value entered is non-negative.
- Check if a period follows immediately after the number of dollars.
- Check if two digits follow right after the period.

To capture different cases, you may need to include selection statements, as well as logical and comparison operators. You may check the data input using the return value of `scanf()` to see how many variables have had their values successfully assigned from `stdin`. To check the user's input format you may use `getc()` to get individual characters from the input stream `stdin` for examination. Having obtained a character and examined a character from `stdin`, if you would like to you may return the character to `stdin` using `ungetc`.

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

Documentation on `getc` and `ungetc` are available at:

https://cplusplus.com/reference/cstdio/getc/

https://cplusplus.com/reference/cstdio/ungetc/

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

Take note that *c99* provides header file that supplies macros `bool`, `true`, and `false`. All the Boolean comparisons in your code must use these macros and **not** the old style `0` (for false) and `1` (for true).

Step 13. Test without loop

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 *noloop-input-00*.txt to *noloop-input-06.txt. Execute the program with each input file, appending the output to the file _actual-output.txt*:

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

Your are also given an output file *noloop-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 noloop-expected-output.txt
```

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

Task-2 (This needs to be submitted...)

Test with loop

In *q.c* modify the function `go()` so that it repeatedly prompts the user for input and prints the corresponding cash denominations if the input is valid. `go()` must take no parameters and return nothing.

The loop in `go()` 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 function should print an error message:

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

Use the same *q.h* as before with a declaration of `go()` and compile with *qdriver.c* to test and submit *q.h* with *q.c* using the submission page.

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

```
./main < loop-input.txt >  actual-output.txt
```

Your are also given an output file *loop-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 loop-expected-output.txt
```

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

Submission and automatic evaluation

1. In the course web page, click on *Lab/Tutorial 4 Submission Page* to submit the necessary files.
2. Please read the following rubrics to maximize your grade. Your submission will receive:
    - F grade if your code doesn't compile with the full suite of `gcc` options.
    - F grade if your code doesn't link to create an executable.
    - F grade if output doesn't match correct output.
    - A+ grade if output matches correct output.
    - Possible 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 have one file-level documentation block. A teaching assistant may physically read submitted source files to ensure that these documentation blocks are authored correctly. Each missing block may result in a deduction of a letter grade. For example, if the automatic grader gave your submission an A+ grade and one documentation block is missing, your grade may be later reduced from A+ to B+. Another example: if the automatic grader gave your submission a C grade and two documentation blocks are missing, your grade may be later reduced from C to F.