Lab 3: Problem Solving, I/O, and Expressions

Learning Outcomes

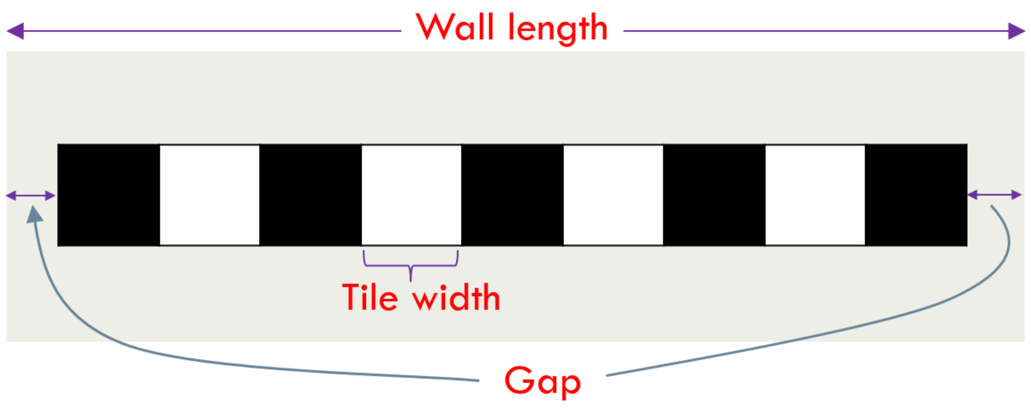
- Practice functional decomposition and algorithm design.
- Apply standard C library functions to solve practical problems.
- Understand the use and limitations of integer and floating-point values.
- Gain familiarity with programming tools and environment.
- Parse and understand behavior of C code.

Task 1

Many programming problems require that you carry out arithmetic computations. This section shows you how to turn a problem statement into pseudocode and, ultimately, a C function. As explained in week 1, a very important step for developing an algorithm is to first carry out the computations by hand. If you can't compute a solution yourself on paper, it's unlikely that you'll be able to write a program that automates the computation. Consider the following problem:

> *A row of black and white tiles needs to be placed along a wall. For aesthetic reasons, the architect has specified that the first and last tile shall be black. Given the wall's length and each tile's width, your task is to compute the number of tiles needed and the gap at each end and to print the computed values to standard output.*

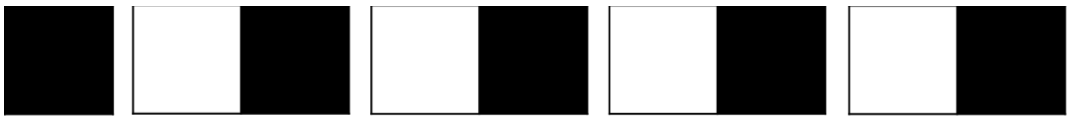
The following picture illustrates the problem:



To perform hand calculations, concrete values must be picked for a typical situation. Let's assume the following dimensions:

- Wall length of 100 inches
- Tile width of 5 inches

The naïve solution would be to start with a black tile, alternate with a white tile and continue in this manner so that the wall is filled with 20 tiles. However, this naïve solution wouldn't work since the final tile would be white when the problem statement requires both the first and last tiles must be black. After mulling over the problem, we construct the following picture that allow us to frame the problem this way:



> ***The first tile must always be black, and then we add some number of white/black pairs.***

The first tile takes up 5 inches, leaving 95 inches to be covered by pairs. Each pair is 10 inches wide. Therefore the number of pairs required is $\frac{95}{10} = 9.5$. However, the final tile must be black and therefore the fractional part of tile pairs must be discarded. Therefore, 9 tile pairs or 18 tiles will be used. Together with the initial black tile, there are a total of 19 tiles which span $19 \times 5 = 95$ inches leaving a total gap of $100 - 19 \times 5 = 5$ inches. Since the total gap must be evenly distributed at both ends, the gap at each end is $\frac{5}{2} = 2.5$ inches.

These hand computations should give us enough information to devise an algorithm with arbitrary values for the wall length and tile width:

```
$\textbf{Algorithm}\ \ tile\,(wall\ length, tile\ width)$

$\text{Input:}\quad WL \leftarrow wall\ length,\ TW \leftarrow tile\ width$
$[\text{Both inputs are floating-point values}]$

$[\text{Algorithm first computes } number\ of\ tiles \text{ and } gap\ at\ each\ end$
$\text{and then prints these computed values to standard output}]$
$[\text{Note that } number\ of\ tiles \text{ is integral while } gap\ at\ each\ end \text{ is a floating-point value}]$

$\text{Output: } Nothing$
$\textbf{1.}\ number\ of\ pairs\ = integer\ part\ of\ \left(\frac{(WL - TW)}{2 \times TW}\right)$
$\textbf{2.}\ number\ of\ tiles = 1 + 2 \times number\ of\ pairs$
$\textbf{3.}\ gap\ at\ each\ end = \frac{(WL - number\ of\ tiles\ \times\ TW)}{2}$
$\textbf{4.}\ print\ computed\ values\ to\ standard\ output$
```

The algorithm $tile$ that I've devised is precise: it clearly specifies the inputs, output, and detailed sequence of steps that transform the inputs into the outputs.

Based on our current knowledge of C, we only know how to return a single value from a function. Therefore, for now, we simply decide to print to standard output stream the two values computed by algorithm $tile$: the number of tiles and the gap at each end.

Task 2

Consider the following problem that simulates the change dispense in a vending machine:

> A customer selects an item for purchase and inserts a dollar bill of any denomination such as $\$1$, $\$2$, $\$5$, and so on into the vending machine. The vending machine dispenses the purchased item and gives change in the correct combination of loonies [dollar coins], half-loonies [50 cent coins], quarters [25 cent coins], dimes [10 cent coins], nickels [5 cent coins], and pennies [1 cent coins].
>
> Your task is to devise an algorithm that:
>
> 1. prompts the customer for a purchase price [of some item that the customer wishes to buy and snack on] and the dollar bill denomination value,
> 2. reads the purchase price and bill value,
> 3. computes how many coins of each type to return to the customer, and then
> 4. prints the coins dispensed starting from loonies through to pennies.
>
> Your algorithm must always choose as many of the largest possible coin denomination before moving on to the next largest denomination. For example, suppose the purchase price is $\$2.01$ and the bill offered is $\$5$, then the change is 299 cents [or $\$2.99$]. In this case, the algorithm will receive as input the purchase price $\$2.01$ and the buyer's bill value $\$5$ as input and must generate the following change: 2 loonies, 1 half-loonie, 1 quarter, 2 dimes, 0 nickels, and 4 pennies.

Devise an algorithm $dispense\_change$ that simulates the change dispenser of a vending machine.

Implementation Details

Open a Window command prompt, change your directory to `C:\sandbox` [create the directory if it doesn't exist], create a sub-directory `lab03` , and launch the Linux shell.

Download driver source files `qdriver.c` and `q.c`, header file `q.h`, and output files `tile.txt` [representing the correct text printed to standard output by calls to function `tile` from function `main`], `change.txt` [representing the correct text printed to standard output by calls to function `dispense_change` from function `main`], and `output.txt` [representing the combined output of calls made to functions `tile` and `dispense_change` by function `main`].

Function declarations in `q.h`

The conversion of algorithms $tile$ and $dispense\_change$ to C code must begin by declaring the two functions in header file `q.h`. Begin by adding a file-level documentation block to `q.h`. Use the inputs and outputs listed in the devised algorithm $tile$ to provide a function-level documentation block for function `tile` and then write the function prototype or declaration of function `tile`. Similarly, provide a function-level documentation block for function `dispense_change` and then write its function declaration.

> ***As explained in class lectures, you must standardize to type `double` for floating-point values and type `int` for integral values.***

Stub functions

Function `main` in `qdriver.c` makes calls to both functions `tile` and `dispense_change`. Rather than defining both functions and then testing them collectively, it is wiser to define and test each individual function separately. Suppose the strategy is to first define and test function `tile` before proceeding to the definition of function `dispense_change`. Begin by adding a file-level documentation block to `q.c`. Add all necessary C standard library header files required to successfully compile this source file. As explained in Assignment 1, introduce stubs [that is, empty function definitions] of functions `tile` and `dispense_change` in source file `q.c` with temporary calls to function `printf` that print the function parameter values in function `tile` and just print the function name in function `dispensie_change`.

Now, you're ready to compile and link the two source files. Begin by compiling [only] your source file `q.c`:

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

If there are any compile errors, read the diagnostic messages from the compiler to identify the source file(s) emitting these message(s).

Compile the driver source file:

```
$ gcc -std=c11 -pedantic-errors -Wstrict-prototypes -Wall -Wextra -Werror -c qdriver.c
```

Link the two source files together to create an executable:

```
$ gcc q.o qdriver.o -o q.out
```

Assuming clean compilation without diagnostic messages, run executable `q.out` to get the expected [incorrect] output from the stub functions.

Definition of function `tile`

The next task is provide a complete definition of function `tile` in source file `q.c`. Begin by adding a file-level documentation block for this source file. Add the necessary C standard library header files required. An obvious inclusion would be of header file `stdio.h`.

Step 1 of algorithm *tile* might pose a problem during the conversion of the algorithm to function `tile`:

$$\textbf{Algorithm}\ \ tile\,(wall\ length, tile\ width)$$

$$\textbf{1.}\ number\ of\ pairs\ \ = integer\ part\ of\ \left(\frac{(WL - TW)}{2 \times TW}\right)$$

How can your function definition obtain the integral portion of a floating-point value? There are a number of ways to extract the integer portion from a floating-point value. The simplest would be use the `floor` function that is a part of the C standard library and is declared in header file

> ***Make note of the fact that C linkers [for historical reasons] do not automatically link with the math library as they do with the rest of the standard library. Therefore, you must explicitly make the linker link with the math library using `-lm` option:***
>
> ```
> $ gcc q.o qdriver.o -o q.out -lm
> ```

How should function `tile` present the computed results to users? A sample of the required output is below and also given in `tile.txt`:

```
Wall length: 100.00 | Tile width: 5.00
Number of tiles: 19
Gap at each end: 2.50

```

Compile and link to generate an executable file `q.out`. Redirect the output from your program to a file `your-tile.txt`:

```
$ ./q.out > your-tile.txt
```

Compare your output to the correct output in file `tile.txt` by using the `diff` program:

```
diff -y --strip-trailing-cr --suppress-common-lines your-tile.txt tile.txt
```

If `diff` completes the comparison without generating any output, then the contents of the two files are an exact match - that is, your source file generates the exact required output. If `diff` reports differences, you must go back and amend `q.c` to remove the reported differences.

Definition of function `dispense_change`

The conversion of algorithm $dispense\_change$ to function `dispense_change` poses a few unique problems - read this how-to guide for this specific task before continuing with the definition of function `dispense_change`.

The first task in the definition of function `dispense_change` is to read the purchase price and dollar bill denomination from the user. As explained in class lectures and here, you must use function `scanf` to read the purchase price as two `int` values: the first `int` value represents the dollar portion of the purchase price and the second `int` value represent the cents portion of the purchase price. For example, if the console input is `2.01 5`, the function will extract 2 as the dollar amount and 1 as the cents. Further, the function must print the following text to the standard output stream with a newline character at the end:

```
2 loonies + 1 half-loonies + 1 quarters + 2 dimes + 0 nickels + 4 pennies

```

If the console input is `63.4 100`, your function must print the following text:

```
36 loonies + 1 half-loonies + 1 quarters + 2 dimes + 0 nickels + 1 pennies

```

```
5 loonies + 1 half-loonies + 1 quarters + 0 dimes + 1 nickels + 1 pennies
```

If the console input is `7.61 10`, your function must print the following text:

```
2 loonies + 0 half-loonies + 1 quarters + 1 dimes + 0 nickels + 4 pennies

```

The output file `change.txt` contains the output for console inputs `2.01 5`, `63.4 100`, and `44.19 50`.

You should further test your implementation for the following inputs. If the console input is `1.99 5`, your function must print the following text:

```
3 loonies + 0 half-loonies + 0 quarters + 0 dimes + 0 nickels + 1 pennies

```

If the console input is `2345.67 5000`, your function must print the following text:

```
2654 loonies + 0 half-loonies + 1 quarters + 0 dimes + 1 nickels + 3 pennies

```

If the console input is `2000.00 8000`, your function must print the following text:

```
6000 loonies + 0 half-loonies + 0 quarters + 0 dimes + 0 nickels + 0 pennies

```

Compiling, linking, and testing

Compile and link your source file `q.c` along with driver source file `qdriver.c` using the full suite of required `gcc` options:

```
$ gcc -std=c11 -pedantic-errors -Wstrict-prototypes -Wall -Wextra -Werror
qdriver.c q.c -o q.out -lm
```

Test your code with a variety of input. Your program is ready for submission if it can generate the exact same output as the combined output in file `output.txt`.

File-level and function-level documentation

Every source and header file you submit *must* contain file-level documentation blocks whose purpose is to provide human readers [yourself and other programmers] useful information about the purpose of this source file at some later point of time

Every function that you declare in a header file [and define in a corresponding source file] must contain a function-level documentation block.

See specs from Lab 2 for a template of an appropriate file-level documentation block and Assignment 2 for a template of an appropriate function-level documentation block.

> ***Don't copy and paste documentation blocks from previous assignments. Annoyed graders will definitely subtract grades to the full extent specified in the rubrics below when they detect instances of copy-and-paste scenarios.***

Submission and automatic evaluation

1. In the course web page, click on the submission page to submit the necessary files.
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, the auto grader will provide a proportional grade 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.

Strategy for solving change problem

This *how to* guide shows you how to turn a problem statement into pseudocode and, ultimately, a C program.

Step 1

Understand the problem. What are the inputs? What are the desired outputs? Should inputs be represented using integer or floating-point types? To achieve the correct result, the program should be able to accurately represent prices such as $2.01 and $4.99. Class lectures have indicated that values represented by floating-point types [such as `float` and `double`] are imprecise because they represent approximations of real values. Let's determine whether this is true or not. Type the following code in a source file:

```c
#include <stdio.h>

int main(void) {
  printf("Enter a fractional value: ");
  double dbl;
  scanf("%lf", &dbl);
  printf("dbl: %f\n", dbl);
  return 0;
}

```

Statements on lines 4, 6, and 7 involve calls to C standard library functions `printf` and `scanf`. We use the term ***argument*** to refer to each [input] value supplied to a function. For example, the call to function `scanf` consists of 2 arguments: `"%lf"` and variable `dbl`'s address `&dbl`.

Compile, link, and execute the test program. Suppose you enter value `2.01` at the prompt - the program displays the text `dbl: 2.010000`. When you enter value `5.99` at the prompt - the program displays the text `dbl: 5.990000`. So far, it seems that floating-point type `double` is able to accurately represent prices [in dollars and cents].

Notice that `printf` displays floating-point values with a default precision of 6 fractional digits. Lets add more precision to the value displayed by `printf` by replacing the statement on line 7 with this statement:

```
printf("dbl: %.20f\n", dbl);
```

In this case, we're asking `printf` to write a more precise representation using 20 fractional digits for a `double` value. If you enter value `2.01` at the prompt, the modified `printf` statement will display the text

```
2.00999999999999978684
```

implying that variable `dbl` doesn't have a precise representation of value `2.01`, that is variable `dbl` is unable to accurately represent the price $\$2.01$.

Try an other price, say $\$5.99$. In this case, the new `printf` statement will display the text

```
5.99000000000000021316
```

All of this means that prices cannot be represented using floating-point types in a program. In fact, monetary software cannot rely on floating-point types to represent money and prices.

What is to be done then? Well, if prices cannot be represented as floating-point values, then they have to represented using integer types. But how?

Suppose that calendar dates are represented using the day-month-year format, as in $15-12-2009$ for 15^{th} of December 2009. Let's think about how to represent dates in a computer. Suppose, we've the day, month, and year for a particular date represented in a program using integer variables:

```
int day, month, year;
```

How can function `scanf` be used to extract the day, month, and year from dates represented in the input stream in the format `15-12-2009`? Each number in the date needs to be stored, but the dashes that separate the numbers have to be discarded. Luckily for us, function `scanf` is versatile enough to be made to skip certain characters in the input stream. For example, the following call to function `scanf` extracts just the day, month, and year from a date while skipping the dashes:

```
int day, month, year;
scanf("%d-%d-%d", &day, &month, &year);
```

To confirm that dates are being read correctly, test the following code.

```
 1  #include <stdio.h>
 2  
 3  int main(void) {
 4    printf("Enter a date in day-month-year format: ");
 5    int day, month, year;
 6    int ret_val = scanf("%d-%d-%d", &day, &month, &year);
 7    printf("Number of items reads by scanf: %d\n", ret_val);
 8    printf("Date is %d-%d-%d\n", day, month, year);
 9    return 0;
10  }
11  
```

Line 6 is using the return value from function `scanf` to confirm that the three items in a date are properly read.

We can apply our enhanced understanding of function `scanf` to the problem of reading prices by reading dollars amount and cents amount as two separate integral values:

```
#include <stdio.h>

int main(void) {
  printf("Enter a price in dollars and cents: dollars.cents format: ");
  int dollars, cents;
  int ret_val = scanf("%d.%d", &dollars, &cents);
  printf("Price read by scanf: %d.d\n", dollars, cents);
  return 0;
}
```

The dollar amount can be scaled by 100 (cents) and combined with the cents value of the price to compute a purchase price in cents. That is, if the price is $\$5.23$, values 5 and 23 are extracted as integers and combined into the value 523.

Based on this discussion, the algorithm will receive two inputs:

1. The first input is an integer type representing the denomination of the bill presented to the vending machine.
2. The second input is an integer type representing the purchase price ***in cents***.

The output from the algorithm is the change in the correct combination of loonies [dollar coins], half-loonies [50 cent coins], quarters [25 cent coins], dimes [10 cent coins], nickels [5 cent coins], and pennies [1 cent coins].

Step 2

Work out examples by hand. This is a necessary and important step for developing an algorithm - if you can't pick or create or choose or determine a sample input data set and then understand how to generate the output by performing hand computations, you'll probably not be able to write a program that automates the computation.

Step 3

Write pseudocode for computing the answers. In the previous step, you worked out a specific instance of the problem. You now need to come up with a method that works in general. Test the algorithm with a broader variety of data. The algorithm must first be tested with the hand examples from Step 2 because those results are already computed. Once the algorithm works for the hand-calculated examples, it should also be tested with additional sets of data to be sure that the algorithm works for other valid data sets.

Step 4

Begin the coding process by authoring an interface file in the manner explained in class lectures and handouts. Next, author an implementation file that ultimately will encapsulate the pseudocode into a function. For now, define a stub or shell function that performs some dummy actions. Confirm your interface and implementation files can be integrated with the driver source file [that is given to you and shouldn't be altered by you] to ensure that an executable binary can be created and run.

Step 5

Encapsulate the pseudocode into a function `dispense_change`. If you did a thorough job with the pseudocode, this step should be easy. Of course, you have to know how to express mathematical operations in C or be able to research using your textbook or online references.