# Tutorial 3: Problem Solving

## Topics

- Gain familiarity with programming tools and environment.
- Continue to get better at computational thinking.
- Introductory C programming.

## Problem Statement

Write a function `dispense_change` that simulates the change dispenser in a vending machine. A customer selects an item for purchase and inserts a bill (can be any denomination such as $\$1$, $\$2$, $\$3$, 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 compute how many coins of each type to return to the customer. Suppose the change is $299$ cents (or $\$2.99$), the function will print the following text to standard output stream:

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

You'll have to define function `dispense_change` in source file `q.c` to print information about change dispensed to standard output. A driver file `qdriver.c` with the definition of function `main` containing a call to function `dispense_change` is provided:

```c
#include <stdio.h> // for printf
void dispense_change(int, int);   // defined by student in q.c

int main(void) {
  int denomination;
  printf("Enter bill denomination inserted into machine: ");
  scanf("%d", &denomination);

  printf("Enter purchase price: ");
  int dollars, cents;
  scanf("%d.%d", &dollars, &cents);
  int price_in_cents = dollars*100 + cents;
  // denomination is in dollars, price_in_cents is price in cents
  dispense_change(denomination, price_in_cents);

  return 0;
}

```

## Testing

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

```
digipen@dit0701sg:/mnt/c/sandbox/tut02$ gcc -std=c11 -pedantic-errors -
Wstrict-prototypes -Wall -Wextra -Werror qdriver.c q.c -o q.out
```

Test your code with a variety of input.

```
digipen@dit0701sg:/mnt/c/sandbox/tut02$ ./q.out < input.txt > your-output.txt
```

## Submission and automatic evaluation

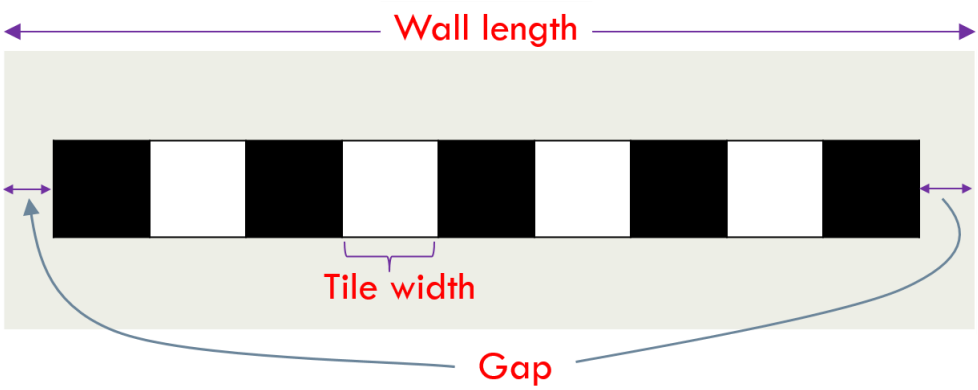
1. In the course web page, click on *Lab/Tutorial 3 Submission Page* to submit `q.c`.
2. Please read the following rubrics to maximize your grade. Your submission will receive:
   - $F$ grade if your `q.c` doesn't compile with the full suite of `gcc` options.
   - $F$ grade if your `q.c` doesn't link to create an executable.
   - $F$ grade if output of function `dispense_change` doesn't match correct output in `output.txt`.
   - $A+$ grade if output of function `dispense_change` matches correct output in `output.txt`.
   - A deduction of one letter grade for each missing documentation block. Your submission `q.c` must have one file-level documentation block and one function-level documentation block for function `dispense_change`. 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 one 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 two documentation blocks are missing, your grade will be later reduced from $C$ to $E$.

# Problem solving: Another review

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 program. 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, it's unlikely that you'll be able to write a program that automates the computation. To illustrate the use of hand calculations, 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. Your task is to compute the number of tiles needed and the gap at each end, given the length of the wall and each tile's width.

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 obvious 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. Instead, look at 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:

$$\textbf{Algorithm } \textit{tile}\,(\textit{wall length}, \textit{tile width})$$
$$\text{Input: } \quad WL \leftarrow \textit{wall length},\ TW \leftarrow \textit{tile width}$$
$$\text{Output: } \textit{number of tiles},\ \textit{gap at each end}$$

$$\textbf{1. } \textit{number of pairs} = \textit{integer part of } \left( \frac{(WL - TW)}{2 \times TW} \right)$$
$$\textbf{2. } \textit{number of tiles} = 1 + 2 \times \textit{number of pairs}$$
$$\textbf{3. } \textit{gap at each end} = \frac{(WL - \textit{number of tiles} \times TW)}{2}$$

First, an interface file `tile-decl.h` is provided for clients:

```
// file header is left as an exercise for reader

#include <math.h>  // required for floor function
#include <stdio.h> // required for printf

// provide file header for function tile so that your clients
// will know how to use your function ...
// example information: what is the minimum values for inputs
// what happens if inputs are negative
void tile(double length_of_wall, double width_of_each_tile);

```

Implementing a C function `tile` that encapsulates this algorithm in source file `tile-defn.h` is straightforward:

```
#include "tile-decl.h"

// writing this function's header is left as an exercise for the reader
void tile(double wall_len, double tile_wid) {
  // the floor function presented in the C standard library returns the
```

```
  // largest integer smaller than the argument to the function.
  // that is, floor(3.9) would evaluate to 3.0.
  double num_tile_pairs = floor( (wall_len - tile_wid) / (2.0 * tile_wid) );
  double num_tiles      = 1.0 + 2.0 * num_tile_pairs;
  double gap_per_end    = ( wall_len - num_tiles * tile_wid ) / 2.0;
  // output to standard output stream ...
  printf("Wall length: %.2f | Tile width: %.2f\n", wall_len, tile_wid);
  printf("Number of tiles: %.0f\n", num_tiles);
  printf("Gap at each end: %.2f\n", gap_per_end);
}

```

Note that no assumptions are made as to whether the wall's length or a tile's width is restricted to integer values. Therefore, the algorithm has been implemented using variables of type `double`. It would not incorrect to impose the restriction that both wall length and tile width must be integers. However, this policy should be clearly indicated in the interface file `tile-decl.h`.

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 function `tile`: the number of tiles and the gap at each end.

The driver `main` function can be implemented like this:

```
#include "tile-decl.h"

int main(void) {
  // test 1
  double wall_length = 100.0;
  double tile_width  = 5.0;
  tile(100.0, 5.0);

  // reader should add more tests here ...

  return 0;
}

```

# Strategy for problem

This *how to* guide 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?

### Representing prices

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 up 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;
}

```

The 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 of the values 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 the value `2.01` at the prompt - the program displays the text `dbl: 2.010000`. When you enter the 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 represent prices (in dollars and cents) accurately.

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:

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

In this case, we're asking `printf` to write a more precise representation of the real value `2.01` to standard output. The modified `printf` statement displays the value:

```
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 value:

```
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 $15^{\text{th}}$ of December $2009$. Let's think about how the representation of dates in a computer. Suppose, we've the day, month, and year for a particular date represented in a program using integer variables:

```c
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:

```
1  int day, month, year;
2  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
10   return 0;
11 }
12
```

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 the dollars and cents amounts as two separate integral values:

```
1  #include <stdio.h>
2
3  int main(void) {
4    printf("Enter a price in dollars and cents: day-month-year format: ");
5    int dollars, cents;
6    int ret_val = scanf("%d.%d", &dollars, &cents);
7    printf("Price read by scanf: %d.d\n", dollars, cents);
8
9    return 0;
10 }
11
```

This can be done it seems that a better alternative to reading monetary values is based on reading the dollars and cents amounts as two separate integral values. 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. values can be combined into a single number representing the price in cents.

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 step - if you can't pick or create or choose or determine a sample input data set, understand how to process the data by performing hand computations at hand very important step for developing an algorithm is to first carry out the computations by hand that generate the output. If you can't compute a solution yourself, it's unlikely that you'll 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 or be able to research using your textbook or online references.