# Assignment: Converting Decimal System to Roman Numeral System

## Learning Outcomes

- Arithmetic and relational expressions.
- Selection structures.
- Practice functional decomposition and algorithm design

## Task

### Roman Numeral System

Numbers in Roman numeral system are formed according to the following rules:

1. Numbers are formed by combining $7$ different numerals: $I, V, X, L, C, D$, and $M$. The decimal values represented by these Roman numerals is illustrated in the following picture:

| Roman numeral | I | V | X | L | C | D | M |
|---|---|---|---|---|---|---|---|
| Decimal value | 1 | 5 | 10 | 50 | 100 | 500 | 1000 |

2. These $7$ Roman numerals are *usually* written largest to smallest from left to right to make up thousands of numbers using an *additive interpretation*. Here are some examples:

   - Number $2$ $(2 \times 1)$ is written by mashing together numeral $I$ like this: $II$.
   - Number $13$ $(10 + 3 \times 1)$ is obtained by mashing together numerals $X$ and $I$ like this: $XIII$.
   - Number $37$ $(3 \times 10 + 5 + 2 \times 1)$ is obtained by mashing together numerals $X, V$, and $I$ like this: $XXXVII$.
   - Number $268$ $(2 \times 100 + 50 + 10 + 5 + 3 \times 1)$ is represented by mashing together numerals $C, L, X, V$, and $I$ like this: $CCLXVIII$.
   - Number $3726$ $(3 \times 1000 + 500 + 2 \times 100 + 2 \times 10 + 5 + 1)$ is represented by mashing together numerals $M, D, C, X, V$, and $I$ like this: $MMMDCCXXVI$.

3. However, Romans did not like writing the same letter more than *three* times. For example, they did not want to write number $4$ as $IIII$. Instead, they developed a method of *subtraction* to avoid repeating the same numeral four times. In this subtractive system, number $4$ is represented as $IV$ by placing smaller numeral $I$ (1) before larger numeral $V$ (5). This means that when you see a smaller numeral before a larger numeral, the subtractive interpretation must be employed. Therefore, when you see Roman numerals $IX$, since numeral $I$ (1) is smaller than the next numeral $X$ (10), you must employ subtractive interpretation to deduce that $IX$ represents $9$. On the other hand, if a larger numeral is before a smaller numeral, additive interpretation is employed. Using additive interpretation, Roman numerals $VI$ represents decimal number $5 + 1 = 6$.

4. Romans used subtraction in six specific instances:

   1. Numeral $I$ (1) is placed before numerals $V$ (5) and $X$ (10) to make values $4$ ($IV$) and $9$ ($IX$), respectively.
   2. Numeral $X$ (10) is placed before numerals $L$ (50) and $C$ (100) to make values $40$ ($XL$) and $90$ ($XC$), respectively.

3. Numeral $C$ (100) is placed before numerals $D$ (500) and $M$ (1000) to make values $400$ ( $CD$) and $900$ ($CM$), respectively.

5. Only numbers up to $3{,}999$ are represented.

## Problem Description

You're to declare in header file `q.h` a function `decimal_to_roman` and define the function in source file `q.c`. Function `decimal_to_roman` will take a `int` parameter specifying a positive decimal integer and return nothing. As its name indicates, the purpose of the function is to print to standard output the Roman numerals equivalent to the positive decimal integer that is passed to the function. The statement `decimal_to_roman(1978);` in function `main` [defined in source file `qdriver.c`] will result in the following text printed to the console:

```
1   MCMLXXVIII
2
```

Notice that function `decimal_to_roman` will insert a newline to standard output after writing the numerals representing the number in the Roman system.

The statement `decimal_to_roman(555);` will result in the following text printed to the console:

```
1   DLV
2
```

A third statement `decimal_to_roman(994);` will result in the following text printed to the console:

```
1   CMXCIV
2
```

You do not require anything more than the problem-solving techniques that have been discussed the past four weeks: understanding decimal numbers and Roman numerals and an algorithm to convert decimal numbers to Roman numerals; sequence statements involving integer arithmetic and assignment expressions; relational expressions and selection structures involving `if-else` or `switch` statements; and the ability to write a single character to the standard output stream using either function `printf` or function `putchar`.

## Strategy for converting decimal numbers to Roman numerals

Begin with the complete list of Roman numerals formed by the subtractive rule:

| Roman numeral | I | IV | V | IX | X | XL | L | XC | C | CD | D | CM | M |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Decimal value | 1 | 4 | 5 | 9 | 10 | 40 | 50 | 90 | 100 | 400 | 500 | 900 | 1000 |

and the representation of numbers $1$ to $9$ in the Roman numeral system:

| Roman numeral | I | II | III | IV | V | VI | VII | VIII | IX |
|---|---|---|---|---|---|---|---|---|---|
| Decimal value | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

Suppose you want to convert $3456$ to Roman numerals. Remembering that numbers only up to $3999$ can be represented in Roman numerals, write $3456$ as $3 \times 1000 + 4 \times 100 + 5 \times 10 + 6$. Since Roman numeral $\mathrm{M}$ is $1000$, $3 \times 1000$ can be written with Roman numerals $\mathrm{MMM}$. The $4 \times 100 = 400$ can be represented from the above table with Roman numerals $\mathrm{CD}$. The $5 \times 10 = 50$ is represented with Roman numeral $\mathrm{L}$. Finally, digit 6 is represented by Roman numerals $\mathrm{VI}$. Concatenating these Roman numerals provides the Roman numeral representation of $3456$ as $\mathrm{MMMCDLVI}$.

Suppose you want to convert $987$ to Roman numerals. Write $987$ as $0 \times 1000 + 9 \times 100 + 8 \times 10 + 7$. Skipping $0 \times 1000$, $9 \times 100 = 900$ is represented from the above table with Roman numerals $\mathrm{CM}$; $8 \times 10 = 80$ is represented as $\mathrm{LXXX}$; and $7$ is represented as $\mathrm{VII}$. Concatenating these Roman numerals provides the Roman numeral representation $\mathrm{CMLXXXVII}$ for decimal number of $987$.

Suppose you want to convert $46$ to Roman numerals. Write $46$ as $0 \times 1000 + 0 \times 100 + 4 \times 10 + 6$. Skipping $0 \times 1000$ and $0 \times 100$, $4 \times 10 = 40$ is represented from the above table with Roman numerals $\mathrm{XL}$; $6$ is represented as $\mathrm{VI}$. Concatenating these Roman numerals provides the Roman numeral representation of $46$ as $\mathrm{XLVI}$.

Suppose you want to convert $9$ to Roman numerals. Write $9$ as $0 \times 1000 + 0 \times 100 + 0 \times 10 + 9$. Ignoring $0 \times 1000$, $0 \times 100$, and $0 \times 10$, $9$ is represented from the above table with Roman numerals $\mathrm{IX}$. Thus, the Roman numeral representation of $9$ is $\mathrm{IX}$.

Finally, year $2020$ (written as $2 \times 1000 + 0 \times 100 + 2 \times 10 + 0$) will have Roman numeral representation $\mathrm{MMXX}$. Next year $2021$ will have Roman numeral representation $\mathrm{MMXXI}$ while the previous year $2019$ will have Roman numeral representation $\mathrm{MMXIX}$.

# 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 `ass04` , and launch the Linux shell.

Download incomplete source and header files `q.c` and `q.h`, respectively.

Using Visual Code, open header file `q.h` and add file- and function-level documentation blocks and declaration of function `decimal_to_roman`.

To test your definition of function `decimal_to_roman`, you'll require a driver that defines function `main` that will make calls to function `decimal_to_roman`. For this assignment, you'll have to author your own version of `qdriver.c` that could look like this:

```c
#include <stdio.h> // declares printf, scanf
#include "q.h"      // declares decimal_to_roman

int main(void) {
  printf("Enter a number (CTRL-D to quit): ");
  int value;
  while (1 == scanf("%d", &value)) {
    if (value <= 0 || value >= 4000) {
      printf("Enter a number (CTRL-D to quit): ");
      continue;
    }
```

```
12        printf("%d: ", value);
13        decimal_to_roman(value);
14        printf("Enter a number (CTRL-D to quit): ");
15      }
16      printf("\nQuitting ...\n");
17      return 0;
18  }
19
```

Notice that line 7 specifies an infinite loop that will allow you to repeatedly test function `decimal_to_roman`. The loop will terminate only when you enter `CTRL-D`.

Compile (only) `qdriver.c` using the full suite of `gcc` options:

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

Use the [strategy](#) described earlier to devise a complete algorithm that practices all the steps of the algorithm formulation process. Edit source file `q.c` by providing file-level and function-level [even though the function-level documentation is not graded] documentation blocks. Include necessary C standard library header file(s). Next, convert your previously constructed algorithm into a definition of function `decimal_to_roman`.

As explained [here](#), your function `decimal_to_roman` must print the computed Roman numerals to standard output followed by a newline.

## Compiling, linking, and testing

Compile (only) your source file `q.c` using the full suite of `gcc` options:

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

Link object files `qdriver.o` and `q.o` plus C standard library functions [such as `printf` and `scanf`] into an executable file:

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

Test your program with a variety of input. Make sure to test what are called edge cases such as the smallest number 1 and the largest number 3999 and numbers such as 4, 9, 40, 90, and so on.

Entering many individual integer values to the program is a cumbersome and slow process. A speedier testing approach relies on collecting the numbers to test in a text file `your-in.txt`. You can then redirect the contents of `your-in.txt` to standard input stream and redirect the standard output stream to a text file `your-out.txt` like this:

```
1  $ ./q.out < your-in.txt > your-out.txt
```

After testing your definition with a variety of integer values, you're now ready to test your code with the exact same input as the online grader. Download input file `qinput.txt`, and the corresponding output file `qoutput.txt` from the assignment web page.

> *To obtain output compatible with output file* `qoutput.txt`*, you must first comment lines 5, 9, 14, and 16 in* `qdriver.c`*, recompile* `qdriver.c`*, and relink* `qdriver.o` *and* `q.o` *to create the executable* `q.out`*.*

Test your program with values in `qinput.txt` like this:

```
1  $ ./q.out < qinput.txt > your-out.txt
```

Compare your output file `your-out.txt` and the correct output file `qoutput.txt` using `diff`:

```
1  $ diff -y --strip-trailing-cr --suppress-common-lines your-output.txt
   qoutput.txt
```

If `diff` is not silent, then your definition of function `decimal_to_roman` is incorrect and will require further work. Also remember to exhaustively test your implementation since text file `qinput.txt` does not contain an exhaustive sample of the $4000$ possible numbers that can be provided as input to the function.

## 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.

> *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 such copy-and-paste scenarios.*

## Submission and automatic evaluation

1. In the course web page, click on the submission page to submit the necessary 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, your submission will be assigned grade $F$.
    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 $F$.