CSC3002 2023 spring Assignment 2

Due 23:59, March 12, 2023

Problem 1 (Points: 50)

P1Part1: (Exercise 2.11, Points: 20)

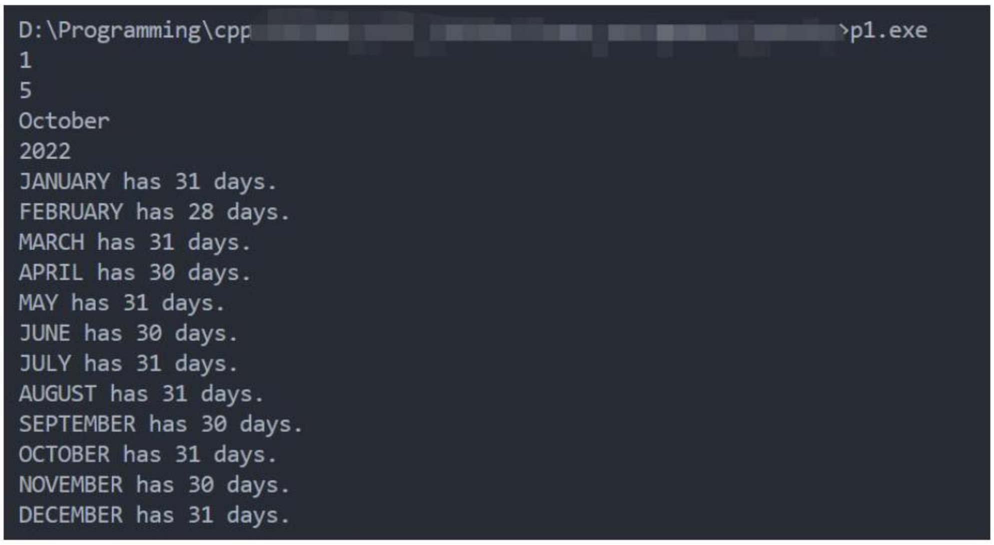
Problem Description

Using the `direction.h` interface [Figure 2-7 in textbook] as an example, using `p1calendar.h` design and implement functions in `p1calendar.cpp` that exports the Month type from Chapter 1.
TODO: You need to complete functions `daysInMonth` and `isLeapYear`. Your should also implement the `monthTostring` function that returns the constant name for a value of type Month.

In & Out Statement

Input of this program has 4 positions, see file *in/p1_ in1.txt*. The types and meanings are all in the following table. Note that values of variables in positions 2 and 3 will not affect your output contents. To test your implementation in this part, index of the task will always =**1**.

Position	type	meaning
0	`int`	index of the task
1	`int`	day
2	`string`	month
3	`int`	year



```
D:\Programming\cpp[redacted]>p1.exe
1
5
October
2022
JANUARY has 31 days.
FEBRUARY has 28 days.
MARCH has 31 days.
APRIL has 30 days.
MAY has 31 days.
JUNE has 30 days.
JULY has 31 days.
AUGUST has 31 days.
SEPTEMBER has 30 days.
OCTOBER has 31 days.
NOVEMBER has 30 days.
DECEMBER has 31 days.
```

Figure 1: Capture of an in & out example.

You can test your program by the following options. `make p1` or `make all` is required before your running and this command will be eliminated in the following contents.

- Test with an sample input file:

  ```
  p1.exe < in/p1_in1.txt  // for win user
  ./p1 < in/p1_in1.txt // for Mac user
  ```

  You may create a new input file manually for further test.

- Test with manual input:

  ```
  //start running by the corresponding command
  p1.exe  // for win user
  ./p1 // for Mac user
  ```

  Possible input format 1:

  ```
  1 1 MAY 1900 // single line
  ```

  Possible input format 2:

  ```
  1
  1 MAY 1900  //multiple lines
  ```

Output Your implementation in `monthToString()`, `daysInMonth()` and `isLeapYear()` will be test automatically in the `main()` function. Please refer to the file *out/p1_out1.txt* for standard output format.

P1Part2: (Exercise 6.5, Points: 20)

Problem Description

Based on the `p1calendar.h` interface above, complete the declaration of `Date` class so that it also exports the following methods:

- A default constructor that sets the date to January $1, 1900$.
- A constructor that takes a month, day, and year and initializes the Date to contain those values. For example, the declaration

  `Date moonLanding (JULY, 20,1969);`

  should initialize `moonLanding` to represent July 20,1969.
- An overloaded version of the constructor that takes the first two parameters in the opposite order, for the benefit of clients in other parts of the world. This change allows the declaration of `moonLanding` to be written as `Date moonLanding (20, JULY, 1969);`
- The getter methods:: `getDay`, `getMonth`, and `getYear`.
- A tostring method that returns the date in the form $dd - mmm - yyyy$, where dd is a one- or two-digit date, mmm is the three-letter English abbreviation for the month, and yyyy is the four-digit year. Thus, calling `tostring(moonLanding)` should return the string "20-Jul-1969".

In & Out Statement

Input format of this part follows the same as that in part 1. To test your implementation in this part, index of the task will always =**2**. You can test your program by the following options:

- Test with an sample input file:

  ```
  p1.exe < in/p1_in2.txt // for win user
  ./p1 < in/p1_in2.txt // for Mac user
  ```

- Test with manual input:

```
//start running
p1.exe // for win user
./p1 // for Mac user
```

  Possible input format 1:

```
2 1 MAY 1900 // single line
```

  Possible input format 2:

```
2
1 MAY 1900 // multiple lines
```

Output Your implementation of `Date` initialization, getter methods and `toString()` method will be test automatically in the `main()` function. Please refer to the file *out/p1_ out2.txt* for standard output format.

P1Part3: (Exercise 6.6, Points: 10)

Problem Description

Extend the `Date` by adding overloaded versions of the following operators:

- The insertion operator $<<$, implemented in `ostream &operator<<`.
- The relational operators $==, ! =, <, <=, >,$ and $>=$, implemented in `bool operator==, operator!=, operator<, operator<=, operator>, operator>=`. We define later date $>=$ earlier date $=$ true, and vice versa.
- The expression date $+n$, which returns the date n days after date, implemented in `Date operator+`.
- The expression date $-n$, which returns the date n days before date, implemented in `Date operator-`.
- The expression $d_1 - d_2$, which returns how many days separate d_1 and d_2, implemented in `int operator-`.
- The shorthand assignment operators $+ =$ and $- =$ with an integer on the right, implemented in `Date &operator+=, Date &operator-=`.
- The $++$ and $--$ operators in both their prefix and suffix form, implemented in `Date operator++(Date &date), Date operator++(Date &date, int), Date operator(Date &date), Date operator(Date &date, int)`.

Suppose, for example, that you have made the following definitions: `Date electionDay(6, NOVEMBER, 2012);`
`Date inaugurationDay(21, JANUARY, 2013);`

Given these values of the variables,

- `electionDay < inaugurationDay` is **true** because `electionday` comes before `inaugurationDay`.
- Evaluating `inaugurationDay - electionday` returns **76** , which is the number of days between the two events.
- The definitions of these operators, moreover, allow vou to write a for loop like `for (Date d<=electionDay; d = inaugurationDay; d++ )` that cycles through each of these days, including both endpoints.

In & Out Statement

Input format of this part is similar to that in the previous parts. To test your implementation in this part, index of the task will always =**3**. Five dates will be input at the same time.

Position	Type	meaning
0	`int`	index of the task
1,2,3	`int, string, int`	Date 1 in form of dd-mm-yyyy
4,5,6	`string, int, int`	Date 2 in form of mm-dd-yyyy
7,8,9	`string, int, int`	Date 3 in form of mm-dd-yyyy
10,11,12	`int, string, int`	Date 4 in form of dd-mm-yyyy
13,14,15	`int, string, int`	Date 5 in form of dd-mm-yyyy

- Test with an sample input file:

  ```
  p1.exe < in/p1_in3.txt // for win user
  ./p1 < in/p1_in3.txt // for Mac user
  ```

- Test with manual input:

  ```
  //start running
  p1.exe // for win user
  ./p1 // for Mac user
  ```

  Input format:

  ```
  3 20 JULY 1969
  NOVEMBER 22 1963
  DECEMBER 31 2009
  21 AUGUST 2011
  6 JANUARY 2010
  ```

Output Your implementation of the above operators will be tested automatically in the `main()` function. Please refer to the file *out/p1_out3.txt* for standard output format.

About P1

- In all the test cases, all month name inputs will only includes upper cases.
- In p3 test cases, date 5 will always be assigned as a earlier date than date 4.
- Pre-test cases on the OJ platform are test case of part1, part2, and part3, respectively.
- The three parts in this problem are combined together. The source file is in `p1calendar.cpp`. You need to finish all `TODO` parts in this file. DO NOT modify the `main()` part, which is for the test unit.

Problem 2 (Exercise 5.13, Points:25)

Problem Description

And the first one now
Will later be last
For the times they are a-Changin'.
-Bob Dylan, "The Times They Are a-Changin'," 1963

Following the inspiration from Bob Dylan's song (which is itself inspired by Matthew 19:30), complete the functions in `p2reversequeue.cpp` :

```
void reverseQueue (queue < string > & queue);
void listQueue(queue < string > & queue);
```

that reverses the elements in the queue. Remember that you have no access to the internal representation of the queue and must therefore come up with an algorithm - presumably involving other structures - that accomplishes the task.

In & Out Statement

Input In this problem, you are recommended to use a `.txt` file to input. Files *in/p2_ in1.txt* and *in/p2_ in2.txt* are provided for your tests.

```
p2.exe < in/p2_in1.txt // for win user
./p2 < in/p2_in1.txt // for Mac user
```

You may create new input files by your own, however, we DO NOT recommend inputting the queue from the terminal directly.

Output Both the original queue and the reversed queue need to be output. Please refer to the file *out/p2_ out1.txt* and *out/p2_ out2.txt* for standard output format.

About P2

- You don't need to re-split or combine any elements in the input queue.
- Please fill in the `TODO` parts of `p2reversequeue.cpp`. DO NOT modify the `main()` part, which is for the test unit.

Problem 3 (Exercise 5.19, Points: 25)

In May of 1844 , Samuel F. B. Morse sent the message "What hath God wrought!" by telegraph from Washington to Baltimore, heralding the beginning of the age of electronic communication. To make it possible to communicate information using only the presence or absence of a single tone, Morse designed a coding system in which letters and other symbols are represented as coded sequences of short and long tones, traditionally called dots and dashes. In Morse code, the 26 letters of the alphabet are represented by the codes shown in Figure .

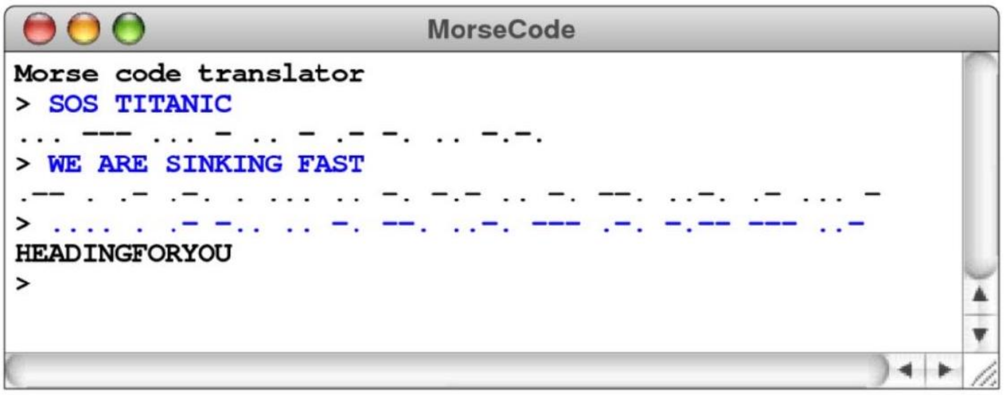
A	•▬	H	••••	O	▬▬▬	V	•••▬
B	▬•••	I	••	P	•▬▬•	W	•▬▬
C	▬•▬•	J	•▬▬▬	Q	▬▬•▬	X	▬••▬
D	▬••	K	▬•▬	R	•▬•	Y	▬•▬▬
E	•	L	•▬••	S	•••	Z	▬▬••
F	••▬•	M	▬▬	T	▬		
G	▬▬•	N	▬•	U	••▬		

Write a program that reads in lines from the user and translates each line either to or from Morse code depending on the first character of the line:

- If the line starts with a letter, you need to translate it to Morse code. Any characters other than the 26 letters should simply be ignored. Lower case letters will be translated after they are transferred into upper cases.

- If the line starts with a period (dot) or a hyphen (dash), it should be read as a series of Morse code characters that you need to translate back to letters. Each sequence of dots and dashes is separated by **spaces**, but any other characters should be ignored. Because there is no encoding for the space between words, the characters of the translated message will be run together when your program translates in this direction.

The program should end when the user enters a blank line. A sample run of an existing program (not the demonstration of this assignment, taken from the messages between the Titanic and the Carpathia in 1912) might look like this:



```
Morse code translator
> SOS TITANIC
... --- ... - .. - .- -. .. -.-.
> WE ARE SINKING FAST
.-- . .- .-. . ... .. -. -.- .. -. --. ..-. .- ... -
> .... . .- -.. .. -. --. ..-. --- .-. -.-- --- ..-
HEADINGFORYOU
>
```

In & Out Statement

Input a string of letters or Morse code. Files *in/p3_in1.txt* and *in/p3_in2.txt* are provided for your tests. To test your code with a text file:

```
p3.exe < in/p3_in1.txt // for win user
./p3 < in/p3_in1.txt // for Mac user
```

To test your program manually:

```
p3.exe // for win user
./p3 // for Mac user

Joker // input
.--- --- -.- . .-. // output
... --- ... // input
SOS // output
```

Output The translated sequence needs to be output. Please refer to the file *out/p3_out1.txt* and *out/p3_out2.txt* for standard output format.

About P3

- A sample of creating Morse code map is given in `p3MorseCode.cpp`. You do not need to implement the letter-to-Morse translation by yourself. For inverted map, generating inverted map manually (typed inverted map directly) is not allowed. You need to create the inverted map using the created map before.
- Input will only include pure letters or pure Morse codes, and no mixed input will be tested.
- Letters input contains upper and lower case letters only.
- Morse code input contains dot, dash and spaces (to split the codes) only.
- Please fill in the `TODO` part of `p1MorseCode.cpp`. DO NOT modify the `main()` part, which is for the test unit.

About this assignment

- You don't need to modify or submit any other files other than `p1calendar.cpp`, `p2reversequeue.cpp`, `p3MorseCode.cpp`.
- Submission to OJ platform is required, while submission to Blackboard again is not necessary.
- Remember to modify the `make clean` commant into `rm -f *.o *.a $(PROGRAM)` if you are a Mac or Linux user.
- You can only view your pre-test scores and such scores do NOT equal to your final score. Pre-test cases take up 10-30% cases of the final test (also known as formal test) cases in this assignment. The formal test will be conducted after the assignment. Your final score will ONLY depend on the codes in your latest submission.
- If you are late, 5 points will be deducted from each missing problem immediately after 00:00, and 5 points more every hour afterward until no more points can be deducted.