CS253 HW1: Dates

Description

Months are confusing! They're all kinds of different lengths! Begone! Weeks are also confusing! Let's avoid them! All we need is a year and the day within the year!

For this assignment, you will write a C++ program called **`hw1`** that will read dates in *year*`.`*day* format and produce equivalent output in a more conventional style.

Input Format

Each input line consists of a year and day number within the year, separated by a period. The first day of this year is **`2020.001`**, the last is **`2020.366`**, and today is **`2020.132`**. Here are some examples, both valid & invalid:

Valid	Invalid
`476.248`	`2020.123A`
`0000476.00000248`	`2020 . 123`
`2020.1`	`2020.900`
`2020.132`	`2019.0`
`002020.33`	`2019.366`
`2020.366`	
`1956.288`	`20🐟19.22`

Output Format

The output format is:

weekday two-digit-day-of-month month four-digit-year newline

Input of **`2020.132`** produces output of **`Mon 11 May 2020`**, corresponding to today, because today is day number 132 of the year 2020.

Sample Run

Here is a sample run, where `%` is my prompt.

```
% cmake .
… cmake output appears here …
% make
… make output appears here …
% cat in
0000476.00000248
2020.1
2019.000000000060
002020.60
2020.366
1.1
9999.365
% ./hw1 <in
Fri 04 Sep 0476
Wed 01 Jan 2020
Fri 01 Mar 2019
Sat 29 Feb 2020
Thu 31 Dec 2020
Mon 01 Jan 0001
Fri 31 Dec 9999
```

Hints

You may find these functions useful. Use them if you wish.

- `stoi()`
- `mktime()`
- `localtime()`
- `ctime()`
- `strftime()`

Beware of unfortunate conventions in `localtime()` & `mktime()`, where months are 0…11 and years are 1900-based.

Gregorian Calendar

Year	Leap?
2019	no
2020	yes
1900	no
2100	no
2000	yes

This assignment uses the current calendar system common in the USA, called the Gregorian calendar. In the Gregorian calendar, leap years (years with February 29) occur in years divisible by four, unless the year is divisible by 100 (no leap year), unless it's divisible by 400 (leap year).

Use the Gregorian calendar for all dates in this assignment, even if they occur before the Gregorian calendar was devised.

Requirements

- Input format:
  - The input may consist of any number of lines.
  - Each input line may be arbitrarily long.
  - I didn't really need to specify the previous two requirements. When an assignment doesn't specify a limit, don't create your own limits.
- Output format:
  - A weekday is one of: Sun Mon Tue Wed Thu Fri Sat
  - A day-of-month number is exactly two digits.
  - A month is one of: Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
  - A year is exactly four digits.
  - Newlines do not merely separate lines—newlines *terminate* lines. Therefore, **every** line ends with a newline, including the last one.
- Creativity is a wonderful thing, but your output format is *not* the place for it. Your output should look ***exactly*** like the output shown above.
  - UPPERCASE/lowercase matters.
  - Spaces matter.
  - Blank lines matter.
  - Extra output matters.
- Error messages:
  - go to standard error.
  - include the program name, no matter how it was compiled.
  - include the offending data, if applicable
- Produce an error message and stop the program if:
  - an input line is not of the proper format: *number.number*
  - an input date has an invalid number (e.g., **`2020.400`** or **`12345.123`**)
  - an input year is not in the range $1 \le year \le 9999$.
- You may wonder: does a problem with line three of the input mean that your program shouldn't produce *any* standard output, or should it produce standard output only for the first two lines?
  - The assignment doesn't specify either of these two reasonable behaviors, so either is acceptable.
  - Which is easier to implement?
- You may not use any external programs via `system()`, `fork()`, `popen()`, `execl()`, `execv()`, etc.
- You may not use C-style I/O such as `printf()`, `scanf()`, `fopen()`, and `getchar()`.
  - Instead, use C++ facilities such as `cout`, `cerr`, and `ifstream`.
- You may not use dynamic memory via `new`, `delete`, `malloc()`, `calloc()`, `realloc()`, `free()`, `strdup()`, etc.
  - It's ok to *implicitly* use dynamic memory via containers such as `string` or `vector`.
- You may not use the `istream::eof()` method.
- No global variables.
  - Except for an optional single global string containing `argv[0]`.
- For readability, don't use ASCII `int` constants (`65`) instead of `char` constants (`'A'`) for printable characters.
- We will compile your program like this: **`cmake . && make`**
  - If that generates warnings, you will lose a point.
  - If that generates errors, you will lose *all* points.
- There is no automated testing/pre-grading/re-grading.
  - Test your code yourself. It's your job.
  - Test with the CSU compilers, not just your laptop's compiler.
  - Even if you only change it a little bit.
  - Even if all you do is add a comment.

If you have any questions about the requirements, **ask**. In the real world, your programming tasks will almost always be vague and incompletely specified. Same here.

Tar file

- For each assignment this semester, you will create a tar file, and turn it in.
- The tar file for this assignment must be called: **`hw1.tar`**
- It must contain:
  - source files (**`*.cc`**)
  - header files (**`*.h`**) (if any)
  - **`CMakeLists.txt`**
- This command must produce the program **`hw1`** (note the dot):

  ```
  cmake . && make
  ```

- *At least* **`-Wall`** must be used every time `g++` runs.

How to submit your homework:

- Use web checkin, or Linux checkin:

  ```
  ~cs253/bin/checkin HW1 hw1.tar
  ```

How to receive *negative* points:

Turn in someone else's work.