Informatik - AS19

Exercise 6: Methods & Stepwise refinement

Handout: 21. Okt. 2019 06:00

Due: 28. Okt. 2019 18:00

Task 1: Perpetual calendar

Open Task

Task

A perpetual calendar can be used to determine the weekday (Monday, ..., Sunday) of any given date. You may for example know that the Berlin wall came down on November 9, 1989, but what was the weekday? It was a Thursday. Or what is the weekday of the 1000th anniversary of the Swiss confederation, to be celebrated on August 1, 2291? It will be a Saturday. The task of this exercise is to write a program that outputs the weekday of a given input date.

Your program will read the date from the input. The input is given as three unsigned integer values in the following order: Day, month, year. First the program must validate the input. Pay attention to special cases of February (leap years!) also, the date must be greater or equal to the reference date. If a date is invalid output `invalid date` and exit the program. If the date is valid, calculate the weekday of this day. This can be done by calculating how many days lie between the date in question and a reference date whose weekday is known. As reference date use *Monday, 1th January 1900*. Finally, output the weekday as one word in English.

Approach: The goal of this exercise is to learn the usage of functions. For that, we split the program into the following sub tasks given as function declarations. Your task is to implement these functions so that they perform the action that is specified in their post condition.

Important: There is a well-known mathematical function to calculate the weekday of a date. Using this function is an incorrect solution as it defeats the purpose of this exercise.

```
// PRE:  a year greater or equal than 1900
// POST: returns whether that year was a leap year
```

```
bool is_leap_year(unsigned int year);

// PRE:  a year greater or equal than 1900
// POST: returns the number of days in that year
unsigned int count_days_in_year(unsigned int year);

// PRE:  a month between 1 and 12 and a year greater or equal tha
// POST: returns the number of days in the month of that year
unsigned int count_days_in_month(unsigned int month, unsigned int

// PRE:  n/a
// POST: returns whether the given values represent a valid date
bool is_valid_date(unsigned int day, unsigned int month, unsigned

// PRE:  the given values represent a valid date
// POST: returns the number of days between January 1, 1900 and t
unsigned int count_days(unsigned int day, unsigned int month, uns
```

Additional notes:

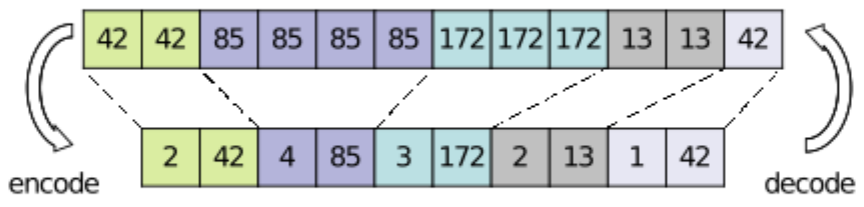
1. For function arguments that do **not** fulfill the precondition, the behavior is undefined.
2. There are a few opportunities here to use switch statements. Use them if they result in better readable code.

Task 2a: Run-length encoding

Open Task

Task

Run-length encoding is a simple data compression technique that represents N consecutive identical values W (a run) by a tuple (N, W). This method is applied for image compression, for instance. Example:



Write a program that implements run-length encoding and decoding of a byte sequence as described above. By a byte, we mean an integer value in the range $[0; 255]$. Use the stepwise refinement method to implement the program. **Your**

solution must consist of at least three functions (i.e., `main` and two other functions).

The input is structured as follows:

1. One integer that determines whether to encode: 0 or decode: 1.
2. Byte sequence to be encoded or decoded (of arbitrary length). If a value outside the range $[0; 255]$ (except -1) is entered, output `error` and stop the en- or decoding.
3. Integer -1 signaling the end of the byte sequence. Any extra input should be ignored.

For the example above, the inputs are:

Encode:

```
0 42 42 85 85 85 85 172 172 172 13 13 42 -1
```

Decode:

```
1 2 42 4 85 3 172 2 13 1 42 -1
```

The output is expected on a single line in the following format:

1. A start value to indicate the begin of the sequence: either 0 for decoded values or 1 for encoded values.
2. The values that make up the encoded or decoded byte sequence.
3. The value -1 to indicate the end of the sequence.

I.e., you can 'reuse' the output as the input.

Note: As the encoded sequence must be a *byte* sequence, runs of length 256 or longer need to be split into multiple runs of length 255 at most.

Special cases: While decoding a byte sequence two special cases can occur. These must be handled as follows:

1. If a byte sequence ends in the middle of a tuple, output `error` and stop the en- or decoding.
2. Tuples of run-length 0 are possible. For such tuples, output nothing.

Hint: You can enter multiple numbers at once separated with a space, e.g, you can copy and paste the above examples, and sequentially read them using multiple `std::cin >> _var_` calls.

Also note that, even though the program's output and the user input are both shown in the same console, they are processed separately internally, so you do not have to worry that the two will mix: if your program outputs something to the console before reading another value, it will only read the user input and not the previous output value. See the Calculator code in the Lecture 4 handout for an example of reading input and producing output in a loop.

Restrictions: storing the read sequence in a vector or a similar data structure is not allowed.

Task 2b: Run-length encoding: pre- and post-conditions

Open Task

This task is a text based task. You do not need to write any program/C++ file: the answer should be written in main.md *(and might include code fragments if questions ask for them).*

Consider the functions implemented in the previous task (*Task 2a*). For each function, write its declaration with proper pre- and post-conditions.

For example, for function

```
double f1_score(double p, double r) {
  return 2 * p * r / (p + r);
}
```

you would write

```
//PRE:  p and r are values between 0.0 and 1.0
//      p + r is different than zero
//POST: returns the f1-score
double f1_score(double p, double r);
```

If no pre-condition is needed, you can simply write `"n/a"`. Function `main()` needs no pre- and post-conditions.

Task 3 (optional): Optimized Run-Length Encoding

Open Task

Task:

Extend task 2 “run-length encoding” by the following optimization:

You may notice that run-length encoding is inefficient for tuples with run-length of $N = 1$ because a single 8-bit value (value range: $[0; 255]$) is represented by two bytes (run-length and value). This can be mitigated by the the following optimization: Allow certain tuples of length $N = 1$ to be represented as single byte value. For that, the most significant bit of a byte is used as a marker whether the byte represents a single value or if it is the first byte of a tuple.

There are now two possible groups of data to consider: tuples, which have the same meaning as in task 2, and single-element shortcut W, which have the same meaning as $(1, W)$.

When attempting to decode a group, if the most signficant bit of the leading byte:

- is set, then the byte initiates a tuple group. The rest of the byte (without most significant bit) represents the run-length N and the next byte that follows provides the value W.
- is not set, the byte represents a single value (a shortcut group).

There is still one issue: if the most significant bit is used as a flag to indicate the type of encoding, we have now only 7 bits to represent our values and run-lengths (eg. the single value 130 (`= 0b10000010`) would be interpreted as a run-length of 2). When 7 bits are not enough you can fallback to the normal tuple encoding, making sure that the run-length stays in the correct range too.