

How to solve a programming problem

Programming is more than just writing code. It also includes the analysis of the problem and the design of the solution, and testing to verify that the solution is correct. The example below should help you become better at solving programming problems.

Request

First, you will need to understand what the problem to be solved is. Normally in college, you are given a comprehensive requirement. This is unlikely to happen in the Real World™, and you will often have to struggle with your users, your customer, or your product management to discover what is actually required.

Write a command-line program to validate weather data coming in from a network of unattended weather stations and report the status of each file. The weather data will be presented in some number of text files of any number of records, each record formatted in the following manner:

Field number	Field name	Field Description	Field Format
1	ID	Station ID	0001 - 2999
2	Date	Date of reading	YYYY-MM-DD
3	Time	Timestamp	HH:MM (24-hour clock)
4	Temperature (dry)	Air temperature	-99.9 to +99.9 degrees C
5	Temperature (wet)	Wet-bulb temperature	-99.9 to +99.9 degrees C
6	Pressure	Barometric pressure	0 to 199.9 kiloPascals
7	Wind direction	Direction of wind source	0 to 359 degrees
8	Wind speed	Wind speed	0 to 299 km/h

No field can be omitted. No field can exceed the range specified. Timestamps are 3 minutes apart, and there can be no missing records. Each file will consist of sequences of records for a single station.

Analysis

Now start to consider what needs to be done. Don't yet concern yourself with "how" to accomplish the steps; that comes much later.

Possible validation errors:

- record has missing field(s)
- ID error
- date invalid
- timestamp invalid
- field has wrong format
- field is outside range

Since this is to be a command-line program, the filenames to process will be arguments, so:

- check for at least one filename argument
- make sure the file to be validated exists and can be opened

We should log the reporting to a file, since it could be confusing to see the output from a lot of files only on the screen. Let's write to both:

- make sure a unique output file can be opened
- make sure we don't run out of disk space for the log file

We will also have to identify the file and the record if there's a problem:

- report filename being processed
- report the record that fails its checks

On the bright side, what about a file that's good?

- report when no errors are found

Problem statement

From the material given and your preliminary analysis of it, write a brief Problem Statement as a starting point. Be sure you understand it, asking questions of the problem sponsor until you are confident.

Write a command-line program to validate weather data. Input will come as one or more command-line filenames for files of data as received from a network of unattended weather stations. Each weather text file will consist of any number of records from a single weather station, formatted in the form defined in the Request.

The program will log all output to the screen and to a uniquely-named log file. It will check each record for the correct number and range of fields as shown in the table in the Request. In addition, the timestamps will be checked to ensure that a record is present for each 3-minute interval, taking care that the date is also correct.

For any error detected, the failing record will be reported followed by a descriptive message about the error. Processing of that file will terminate, and the program will move on to the next file in the list, if any. At the end of the list of filenames, the program will terminate. Each file that passes all the validity checks will be reported as clean.

Design

Once it's agreed that you understand the problem, you need quite a number of things before you're ready to start the implementation (writing code). Luckily, some of them are quite short, although some also tend to be interconnected and usually need to be done simultaneously. Here' are lists for the two parts of the Design phase:

External Design

1. *input files — what files will you need or be given, and what do they look like?*
2. *output files — what is the normal output you must produce, and to what files?*
3. *error files — similarly, list all the errors you can think of, and how you will notify the user about them.*
4. *draft user documentation — this will need to be written, so write it now (as a draft, since you may want to change it later).*

5. *first test plan* — you've listed the errors, now list the tests you will need.

Internal Design

1. *rough logic flow* — sometimes called the *Narrative*, this is most often a point-form bulleted list of the steps your eventual coding solution will require to solve the problem as you have now determined.
2. *function descriptions* — functions are the basic building blocks of many programming solutions. From your logic flow, estimate where you can best make use of functions, what they will do, and what their arguments are and what their return value is.
3. *data dictionary* — now you can list your most important variables, along with their purpose, their type, and (if an array) how many elements they need. You will also describe structs, unions, and enums here.
4. *Problem Description Language* — now you are ready to write the formal PDL. This is a complete and detailed description of your proposed solution, written in a strictly logical form largely independent of any particular programming language. This is the **WHAT** and **WHY**, but **not** the **HOW**.
 - a) *WHAT* – the steps you're going to follow, in sequence, including loops and functions calls, both library functions and your own, but avoid describing how you're going to code it;
 - b) *WHY* – describe why you're doing something, not how you plan to code it: for example say, “if 1 or more arguments” rather than “if argc is more than 1”, or “for each item in the list” instead of “for i from 0 to n - 1”;
 - c) *HOW* – this is the specifics of your solution, the code you will write. Don't do this until you're worked through all the logic in your PDL and verified it by walking some samples through it (see first test plan).

External Design

1. Input files

stdin

- nothing;

data files

- a list of command-line arguments, minimum one, no maximum:

Field number	Field name	Field Description	Field Format
1	ID	Station ID	0001 - 2999
2	Date	Date of reading	YYYY-MM-DD
3	Time	Timestamp	HH:MM (24-hour clock)
4	Temperature (dry)	Air temperature	-99.9 to +99.9 degrees C
5	Temperature (wet)	Wet-bulb temperature	-99.9 to +99.9 degrees C
6	Pressure	Barometric pressure	0 to 199.9 kiloPascals
7	Wind direction	Direction of wind source	0 to 359 degrees
8	Wind speed	Wind speed	0 to 299 km/h

2. Output files

stdout

- report current file being processed: file <filename> start
- report current file passed validation tests: file <filename> passed
- report current file failed validation tests: file <filename> failed

log file

- duplicate of all stdout and stderr output

3. Error files

stderr

- arguments: must have at least one filename
- no input file: file <filename> does not exist
- no access to input file: file <filename> cannot be read
- log file creation failure: log file <logname> create failure
- log file output error: log file <logname> write failed

All messages below are immediately preceded by the display of the current record being processed:

- record has missing field(s): incomplete record
- ID error: station ID change
- ID form error: station ID error

- invalid date: date invalid
- invalid timestamp: time invalid
- missing record: out of time sequence
- temperature (dry) error: temperature (dry) error
- temperature (wet) error: temperature (wet) error
- pressure error: pressure error
- wind direction error: wind direction error
- wind speed error: wind speed error

log file

- duplicate of all stdout and stderr output

4. Draft User Documentation

The v8 (validate) program performs command-line validation under Linux for the raw weather station data files. To run it, ensure that the program is in a directory on your PATH. Move to a directory containing suitable data files and enter a line like:

```
v8 filename1 [filename2 ...]
```

That is, there must be a minimum of one filename on the command line, but you may have as many as you wish. Wild-card file specification will behave correctly.

The validate program will create a log file with a name like

```
v8-99999-yymmddhhmm.log
```

where 99999 represents a number that will be different for every log file, and yymmddhhmm represents the time of the start of the program. The log file will contain a copy of every message sent to the screen while this copy of v8 is running (you can run many copies of v8 at once).

If a data file cannot be found, cannot be read, or contains a data error, that file will stop being processed and v8 will carry on with the next program in its argument list. Each data error will print the line in error followed by a brief diagnostic message. The data file will have to be corrected manually and re-validated before use.

5. First Test Plan

Test	Input	Purpose	Expected result
1	v8	no file arguments	must have at least one filename
2	v8 no-file	file not found	file no-file does not exist
3	v8 bad-file	no access to file	file bad-file cannot be read
4	v8 empty-file	no contents	file empty-file start file empty-file passed file file1 start file file1 passed file file2 start file file2 passed file file3 start file file3 passed
5	v8 file1 file2 file3	multiple files	file bad-record start 0001 2009-01-01 12:00 -10.0 -12.0 101.3 5.6 incomplete record file bad-record failed file id-change start 9999 2009-01-01 12:03 -10.0 -12.0 101.3 5.6 19 station ID changed file id-change failed file id-range start 99990 2009-01-01 12:00 -10.0 -12.0 101.3 5.6 19 station ID error file id-range failed
6	v8 bad-record	malformed record	
7	v8 id-change	multiple IDs	
8	v8 id-range	ID out of range	

Test	Input	Purpose	Expected result
9	v8 date error	malformed date	file date-error start 0001 2009-13-01 12:00 -10.0 -12.0 101.3 5.6 19 date invalid file date-error failed
10	v8 time-error	malformed time	file time-error start 0001 2009-01-01 25:00 -10.0 -12.0 101.3 5.6 19 time invalid file time-error failed
11	v8 sequence-error	omitted record	file sequence-error start 0001 2009-02-01 12:06 -10.0 -12.0 101.3 5.6 19 out of time sequence file sequence-error failed
12	v8 temp-dry-error	malformed dry temp	file temp-dry-error start 0001 2009-01-01 12:00 -111.0 -12.0 101.3 5.6 19 temperature (dry) error file temp-dry-error failed
13	v8 temp-wet-error	malformed wet temp	file temp-wet-error start 0001 2009-01-01 12:00 -10.0 -122.0 101.3 5.6 19 temperature (wet) error file temp-wet-error failed
14	v8 pressure-error	malformed pressure	file pressure-error start 0001 2009-01-01 12:00 -10.0 -12.0 222.3 5.6 19 pressure error file pressure-error failed

Test	Input	Purpose	Expected result
15	v8 wind-dir-error	malformed wind dir	file wind-dir-error start 0001 2009-01-01 12:00 -10.0 -12.0 101.3 445.6 19 wind direction error file wind-dir-error failed
16	v8 wind-speed-error	malformed wind speed	file wind-speed-error start 0001 2009-01-01 12:00 -10.0 -12.0 101.3 5.6 -19 wind speed error file wind-speed-error failed

Test files

bad-file

[file has no read permissions, content irrelevant]

empty-file

[file empty, no content]

file1

0001 2009-01-01 12:00 -10.0 -12.0 101.3 5.6 19

file2

0002 2009-01-01 12:00 -10.0 -12.0 101.3 5.6 19

file3

0003 2009-01-01 12:00 -10.0 -12.0 101.3 5.6 19

bad-record

0001 2009-01-01 12:00 -10.0 -12.0 101.3 5.6

id-change

0001 2009-01-01 12:00 -10.0 -12.0 101.3 5.6 19

9999 2009-01-01 12:03 -10.0 -12.0 101.3 5.6 19

id-range

99990 2009-01-01 12:00 -10.0 -12.0 101.3 5.6 19

date error

0001 2009-13-01 12:00 -10.0 -12.0 101.3 5.6 19

time-error

0001 2009-01-01 25:00 -10.0 -12.0 101.3 5.6 19

sequence-error

0001 2009-01-01 12:00 -10.0 -12.0 101.3 5.6 19

0001 2009-01-01 12:06 -10.0 -12.0 101.3 5.6 19

temp-dry-error

0001 2009-01-01 12:00 -111.0 -12.0 101.3 5.6 19

temp-wet-error

0001 2009-01-01 12:00 -10.0 -122.0 101.3 5.6 19

pressure-error

0001 2009-01-01 12:00 -10.0 -12.0 222.3 5.6 19

wind-dir-error

0001 2009-01-01 12:00 -10.0 -12.0 101.3 445.6 19

wind-speed-error

0001 2009-01-01 12:00 -10.0 -12.0 101.3 5.6 -19

External Design

1. Rough Logic Flow

Just what it says – lay out the steps you expect your program to follow in the correct order. Do not worry about details, number of functions, variables, or anything else but the logic and the flow.

```
check for correct arguments
prepare the log file
for each file in the argument list
    open the file, else report error
    for each line in the file
        check for 6 values
        verify correct station id
        verify correct timestamp
        for each value
            make sure it's valid
        end value for
    end line for
end file for
```

2. Function Descriptions

By examining the logic flow, decide what can most usefully be placed into functions. Determine what each one will do, what information it needs in order to be able to do that, and what its result should be.

Note that functions do not normally issue either error messages or regular output, except for a function expressly designed to do so like `write_line()` below. Instead, return a value to the caller and let it decide.

```
int check_range(char *value, int min, int max, int flag);
```

- check the given value for float or int (flag is 1 for float, else 0) and ensure it lies within the range given, returning 1 for true, 0 for false.

```
int time_check(char *time_fields, char *prev_time);
```

- verify that the date and time is valid and 3 minutes after the previous timestamp (or the first one if `prev_time` is empty); returns 1 for true.

```
int write_line(FILE *stream, FILE *log, char *message);
```

- write the message provided to both the log file and to stdout or stderr, returning 0 if OK or 1 for an error.

3. Data Dictionary

For each function you're going to write, including `main()`, determine what key (that is, important) variables you will need. Exclude casual variables like loop

controls (i, j, etc.), and do not include the function arguments. Give the name and type for each such variable, the number of elements of each, and a brief description of the purpose. Add new entries during the PDL writing and the coding when you come across them. If you are planning to use struct, union, or enum declarations, put them here as well. Global variables (file scope or greater) should have their own section, but also justify each one in detail as to why it cannot be local to one of your functions.

Function: main()

Name	Type	Num.	Purpose
filename	char*	1	current filename in argv
logfile	FILE*	1	stream for log file
in_file	FILE*	1	stream for input file
line	char[]	100	line buffer
value	char*	1	pointer to current value in line
prev_time	char[]	17	copy of previous timestamp, if any
station_id	int	1	value of station id, or 0 for first
message	char[]	100	buffer for message creation
ranges	int	5 by 3	ranges for 5 fields
error_msgs	char*	5	range error text for 5 fields

Function: check_range()

Name	Type	Num.	Purpose
f_val	float	1	value if float
i_val	int	1	value if int
rc	int	1	return value
end_ptr	char*	1	pointer for strtod()/strtol()

Function: time_check()

Name	Type	Num.	Purpose
year	int	2	value for year, previous/now
month	int	2	value for month, previous/now
day	int	2	value for day, previous/now
hour	int	2	value for hour, previous/now
min	min	2	value for minute, previous/now

Function: write_line()

Name	Type	Num.	Purpose
= none =			

4. Program Description Language

You now have enough information that you can write formal PDL for each of your functions. It is a description of the WHAT and WHY of your program. Do not attempt to explain HOW something is to be done, except perhaps a brief and separate explanation of some non-obvious algorithm or formula.

Consider PDL for both of two purposes. First for the code writer, it's a description of what needs to be done. You should be able to hand your PDL, plus the other material above, to another coder and have him or her implement the program. Indeed, some development departments do exactly that. Second, for the code maintainer, it's how he or she will determine how to add a feature or fix a bug in the code, because the PDL (along with the comments in your code) is an aid to understanding what the program does and why it does it.

The purpose of the PDL is to supplement the code, not duplicate it.

```
BEGIN main()
  IF there is not at least one filename argument
    PUT "must have at least one filename" to stderr
    RETURN error exit
  ENDIF
  IF the log file cannot be opened
    PUT "log file <logname> create failure" to stderr
    RETURN error exit
  ENDIF
  FOR each name in the argument list
    IF the file cannot be opened
      CALL write_message "file <filename> does not exist"
      CONTINUE with next filename
    ENDIF
    FOR each line read from the file
      IF end-of-file found
        CONTINUE with next filename
      ELSE if read error
        CALL write_message "file <filename> cannot be read"
        CONTINUE with next filename
      ENDIF
      IF line does not match required format
        CALL write_message "incomplete record"
        CONTINUE with next filename
      ENDIF
      IF station_id is not 0 AND it's different from this one
        CALL write_message "station ID change"
        CONTINUE with next filename
      ELSE if station_id is zero
        station_id = this station id
      ENDIF
      CALL check_time with current and previous timestamps
      IF timestamp is not OK
        CALL write_message "out of time sequence"
        CONTINUE with next filename
      ENDIF
```

```

        FOR fields 4 to 8
            CALL check_range with required range for the field
            IF field is not valid
                CALL write_message with error message
                CONTINUE with next filename
            ENDIF
        END "fields" FOR
    END "line" FOR
END "file" FOR
RETURN successful end
END main()

```

That's the first draft of the PDL for the main() function. Now you can complete this task by writing the PDL for the three functions, and then writing the code.

Write the code in chunks, a few lines at a time. Then compile it and fix any syntax or typing errors that are found. Leave out loop bodies (for example) and function calls until you have the framework of the logic compiled and tested alone. Once there is enough code written you can start to run your test plan against the partial program.

You can also use "stubs" for your functions at this point – a function with the correct name and arguments that only returns a test result and does nothing else, such as:

```

int time_check(char *time_fields, char *prev_time)
{
    return 1;
}

```

Once the rest of the program has run satisfactorily with this stub function, you can replace it when you are ready to write and test the complete function.

Why do it this way? Because it's a lot easier to find and fix a handful of syntax or typing errors at a time than it is to address screens and screens full of them when you attempt to compile all the code at once for the first time. You will find that a reasonably simple typo can result in many cascading errors, since it can affect all the code below that point; try omitting a single semicolon sometime!

Testing with stubs has a similar advantage. By leaving out some of the code, you can get the rest to work correctly. Then you have known good code as a framework to support each chunk of new code that you are about to test.