COMP 348: Principles of Programming Languages
Assignment 1

Summer 2023, sections AA and AB

May 25, 2023

Procedural Programming with C

Date posted: Friday May 19th, 2021.

Date due: Monday June 5th, 2021, by 23:59.

Weight: $8\frac{1}{3}$% of the overall grade.

IMPORTANT: Make sure the C++ compiler option is turned OFF and your code is compiled and run under gcc on ubuntu.

Overview

In this assignment you write a program called `mathpipe` that performs simple operations on some input numbers. The program reads a jagged-array[1] from the standard input and applies the requested operation on each row and prints the result to the standard output. Upon processing a row, the output will be printed to the standard output. In case of any processing error, the error message is reported to the `stderr` followed by the immediate *abnormal* termination of the program. In case of no error, the program *successfully* terminates. Examples of processing errors are invalid function name, a bad number format, memory allocation error, etc.

[1]A jagged-array is an array whose rows may contain different number of cells.

`mathpipe` supports 6 *aggregate* as well as 3 *singular* operations. They are as follows.

Aggregate and Singular Operations

The specifications of the six "aggregate" functions are given in the following.

- "COUNT": prints the number of the elements in a given array.
- "MIN": applies the function $\min(\{a_i\})$, that returns the minimum of the elements in a given array.
- "MAX": applies the function $\max(\{a_i\})$, that returns the maximum of the elements in a given array.
- "SUM": applies the function $\mathrm{sum}(\{a_i\})$, that returns the sum of the elements in a given array.
- "AVG": applies the function $\mathrm{avg}(\{a_i\})$, that returns the average of the elements in a given array.
- "PAVG": applies the function $\mathrm{pseudo\_avg}(\{a_i\}) = (\min(\{a_i\}) + \max(\{a_i\}))/2$.

The three "individual" operations are as follows:

- "PRINT": simply prints the number to the standard output, no change in value.
- "FILTER" `<TYPE> nnn.nn`: prints[2] the number(s) if they pass the filter defined by <TYPE>. Supported types are: `EQ`, `NEQ`, `GEQ`, `LEQ`, `LESS`, `GREATER`. `nnn` is the threshold in floating point.
- "SHIFT" `nnn.nn`: adds `nnn.nn` to each element of the array.

Command-Line Arguments

While the input matrix is read form the standard input, `mathpipe` receives the parameters through command line arguments. No parameter must be hard-coded or entered by the user. `mathpipe` receives one mandatory parameter which is the function name and two optional parameters: size, and precision which are specified by `‘‘-size=nnn’’` and `‘‘-prec=n’’`, respectively, where `nnn` represents a decimal number and `‘‘n’’` is a decimal digit, representing

[2]See the `filter()` function later in this document.

the number of digits to be printed after the decimal point[3]. There is no predefined order for the parameters and any of the three may appear in any position. Some functions (i.e. FILTER) may have additional parameters that are given immediately after the name. See the usage examples.

Usage Examples

```
cat sample.txt | mathpipe sum -size=2             # prints 3.50000 6.0000 25.0000
cat sample.txt | mathpipe -size=3 PAVG -prec=1      # prints 2.5 2 12.5
cat sample.txt | mathpipe -size=3 PRINT -prec=0     # prints:
                                                    #   1 3 4
                                                    #   2
                                                    #   12 13
cat sample.txt | mathpipe -size=1 FILTER GEQ 10 | mathpipe SUM
                                                    # prints:
                                                    #   12 13
cat sample.txt | mathpipe SHIFT 5.5 -prec=1        # prints:
                                                    #   6.5 8 9.5 7.5
                                                    #   17.5 18.5
```

sample.txt[4]:

```
1 2.5 4 2
12 13
```

The operations names must be precisely defined as listed above. However, the user may use them in small letters (or mixed case).

Implementation Requirements

- Your program reads each row in a single dimensional array, whose size may be specified by the user.
- Your program must read the inputs row by row until it reaches the `EOF`.

[3]See format specifiers in `printf()`.

[4]The third example in the above generates `1 2 4` (as apposed to `1 3 5`), due to arithmetic errors in floating points, since 2.5 is stored as 2.49999... in binary.

- All array elements are `double`. You may use `sscanf` to read individual numbers from a line that is already read in memory.
- If no size is specified, the program uses a default size as the number of columns in the first row (line) which cannot exceed the system default size 256[5].
- If an input row has less columns than the default size, the number of columns would be considered as the column size for the row.
- If an input row has more columns than the default size, the exceeding values are automatically considered to be in a new row.
- The output of aggregate operations are separated by a single white space, all together followed by a new line character.
- The output of singular operations are separated by a single white space on each row, and each row is terminated by a new line character.
- If a row is empty, it will simply be skipped. No empty line will be reported to the output.
- Processing of each row must be done in a dynamically allocated array, which must be properly freed up, upon termination.
- While aggregate operations return a singular value, applying singular operations on the input arrays are done in-place. See the examples in the next sections.

File Structure

Your program consists of 3 modules (source / include files):

- `aggregate.h/c`: that implements the aggregate operations.
- `singular.h/c`: that includes non-aggregate operations.
- `mathpipe.c`: that contains your main code.

[5] All constants and parameters must be defined using `#define` directive in the code.

aggregate.h The header file contains the declaration of the only externally visible functions:

```
double aggregate(const char* func, double* arr, int size);
/* usage example
static double arr[] = {1, 4, 5, 6, -1};
double m = aggregate("MIN", arr, 5);
printf("%f", m); // -1.000000 */
```

Note: that the function receives the array size as a parameter.
In case the input array is `NULL` or the size is non-positive, the function prints "`FATAL ERROR in line ...`" where "..." represents the line number and the program is aborted (see Ref 1).

aggregate.c The implementation file for the aggregate function.
The individual aggregate functions (to be called by the `aggregate()` method above) must be implemented individually in the same file. Make sure they are hidden and not externally visible.
An example of the signature of an individual aggregate function is provided in the following:

```
static double _min(double* arr, int size);
```

The link to the individual aggregate functions must be done via an internal array. To do use two internal arrays: one for the operation names, and one for the pointer to the operation function. Note that all operations use same signature.
Examples of array of pointers are given below.

```
// Given the following function prototype
static void f(void);
static void f2(void);
// The following declares an array of pointer
static void (*farray[2])(void) = { &f, &f2 };
// The following declares an array of strings
static const char* funcnames[] = { "F", "F2" };
```

Similar to above, define the corresponding arrays in the code file[6]. As a result, the `aggregate()` method looks for the function name in the array of names, and uses the index to find the address of the corresponding method. For instance, in the above code, function is "F" appears on index 0, therefore the corresponding routine may be accessed via `farray[0]`.

In addition to the above, revise the declarations and use typedef[7] for function pointers[8]. The search for method name must be case-insensitive. In case the function is not found, a similar FATAL error may be generated and the program is aborted.

singular.h Unlike the above, the singular operations are individually defined in this header file. The header 3 methods are all externally visible:

```
enum filter_type { EQ = 0, NEQ = 1, GEQ = 2, LEQ = 3, LESS = 4, GREATER = 5 };
void print(double a[], size_t size);
void shift(double a[], size_t size, double by);
size_t filter(double a[], size_t count, enum filter_type t, double threshold);
```

singular.c The implementation of the above methods are done in this file.
The print function prints a single array, as specified earlier in this document; the shift function applies an in-place shift operation to the input array; and the filter function performs an in-place filter in the same array. Note that the filter function does **NOT** print the elements. It simply uses an in-place shift operation (if necessary) to make sure the resulting array contains the elements that passes the *filter* criteria. You program, however, prints them. That can be achieved by calling the `print()` function on the resulting array that is prepared by the `filter()` function.
Some extreme-cases are illustrated below:

```
// static double arr[] = {4, 5, 6, 1, -1, 8, 2};
print(arr, 1); // prints 4 followed by a new-line
print(arr, 0); // nothing is printed (no new-line)
shift(arr, 4, .5);
// {4.5, 5.5, 6.5, 1.5, -1, 8, 2};
```

[6]Why not header file?
[7]A *typedef* in C lets the programmer create an additional name (alias) for another data type.
[8]For instance, use `aggregate_func` as the type name for the pointer to the individual aggregate function

```
size_t c = filter(arr, 5, LEQ, 6); // returns 4, prints nothing.
// {4.5, 5.5, 1.5, -1, -1, 8, 2}; // last three elements remain untouched
```

The above filter call scans the array and checks all first 5 elements and keeps the only elements in the array only if they are "less than 6". The search stops when the 5th element is checked. The remaining elements of the array will be untouched.

mathpipe.c This file includes the main method. You may create any number of subroutines / functions as you wish to address the requirements of the assignment. You must dynamically allocated all arrays you use (however, only one would be sufficient). Processing the elements in the arrays as well as printing them must be strictly done via the above two modules.

Submission

IMPORTANT: All files (`.c` and `.h`) must be prepared in the same folder so that they can be compiled from the command line using the following simple `gcc` command:

```
gcc -Wall aggregate.c singular.c mathpipe.c
```

Before submitting, you must test your code with this command to make sure that it works (this particular command will produce a compiled executable called `a.out`). Once you are ready to submit, compress all .c/.h files (and ONLY these files) into a zip file. The name of the zip file will consist of "a1" + last name + first name + student ID + ".zip", using the underscore character "_" as the separator. For example, if your name is John Smith and your ID is "123456", then your zip file would be combined into a file called "`a1_Smith_John_123456.zip`". The final zip file will be submitted through the course web site on Moodle. You simply upload the file using the link on the assignment web page.
Please note that it is your responsibility to check that the proper files have been uploaded. No additional or updated files will be accepted after the deadlines. You can not say that you accidentally submitted an "early version" to Moodle. You are graded only on what you upload.
Note that assignments must be submitted on time in order to receive full value. Appropriate late penalties ($<$ 1 hour = 10%, $<$ 24 hours = 25%) will be applied, as appropriate.

References

1. C Standard Predefined Macros:
   `https://gcc.gnu.org/onlinedocs/cpp/Standard-Predefined-Macros.html`
2. Aggregate Function: `https://en.wikipedia.org/wiki/Aggregate_function`