ENGR151 — Accelerated Introduction to Computer and Programming

Lab 5

Manuel — UM-JI (Fall 2021)

Goals of the lab

- Write header clean files
- Use of conditional statements
- Practice online search for C

1 Background

Haruka, Kana, and Chiaki are busy with many many midterms while Jane struggles with her own projects and TA work. In this intense context the three sisters still want to play with new programs but are a bit limited in what they have time to do. At the end of a recent VG101 lecture Hakura realised that C is not very user friendly when it comes to playing with numbers. While MATLAB could be used as a simple calculator out of the box, in C even basic things like rounding to the nearest integer is complicated.

After a brief discussion the three sisters decide to write their own function for calculating basic arithmetic operations, rather than directly using the C operators.

2 Tasks

As Haruka takes the lead in this mini-project she prepares a header file. Her idea consists in using a single function to deal with all the arithmetical operations that she wants to override.

calculator.h

```c
// evaluate takes 2 integer and an operator as arguments and returns a double.
// The behavior of the function depend on the operator, as described below.
// Note that the result should never be rounded.
// operator == '+': return operand1 + operand2. (optional)
// operator == '-': return operand1 - operand2. (optional)
// operator == '*': return operand1 * operand2.
// operator == '/': return operand1 / operand2.
// operator == '^': return operand1 ^ operand2, where operand2 is the exponent.
// operator == 'l': return log(operand1, operand2), where operand1 is the base.
// operator == 'm': return r, which satisfies operand1 = q * operand2 + r,
// where q and r are integers, r * operand2 >= 0, and abs(r) < abs(operand2).
double evaluate(int operand1, char operator, int operand2);
```

Mandatory part

2.1 Interface and implementation

As soon as the two other sisters look at Haruka proposal they simply ask her: "Are you sure to comply with the code quality requirements?" Immediately Haruka realises her mistake, deeply apologizes, and

fixes it. Now that the interface is properly set, they are ready to work on the actual implementation of the `evaluate` function.

Help the three sisters to

- Fix Haruka's mistakes when writing the header file; (2 min – 2 students)
- Decide whether you use `if` or `switch` when implementing `evaluate`; (2 min – 2 students)
- Write the code of `evaluate` for `+`, `-`, `*`, and `/`; (6 min – 2 students)
- Search reference websites to find out how to compute `^`, `l`, and `m` in C; (6 min – 2 students)
- Write the code of `evaluate` for `^`, `l`, and `m` in C; (6 min – 2 students)

Present and demonstrate to the three sisters the result of your work. (5 groups – 3 min each)

Optional part

When the three sisters are done with the evaluate function they realise their file fails to compile. However they quickly realise this is not surprising since they have not written any `main()` function. At that point Kana says "But wait! Based on what we learnt in the lectures it means we have a library?"

2.2 Libraries

To the sisters, writing their own library sounds very exciting. They immediately search online to learn more about that and understand how to build a library using `gcc`. They rapidly find helpful resources but not everything is as clear and simple as they initially expected it to be.

Libraries

In computer science, a *library* is a collection of non-volatile resources used by computer programs, often for software development. These may include configuration data, documentation, help data, message templates, pre-written code and subroutines, classes, values or type specifications.

For a large project, it may contain millions of lines of code. It will take several hours to build only part of the whole project. So it is very important to separate different parts of the project into libraries.

A *static library* or *statically-linked library* is a set of routines, external functions and variables which are resolved in a caller at compile-time and copied into a target application by a compiler, linker, or binder, producing an object file and a stand-alone executable.

A *shared library* or *shared object* is a file that is intended to be shared by executable files and further shared object files. Modules used by a program are loaded from individual shared objects into memory at load time or run time, rather than being copied by a linker when it creates a single monolithic executable file for the program.

Help the three sisters to

- Clarify and fully understand all the words and ideas presented in the summary box they read;
- Explain in simple terms the difference between shared and a static libraries;
- Understand the full compilation process, i.e. pre-processing, building, and linking stages;
- Build both a shared and a static library out of the `evaluate` function;

2.3 A CLI Calculator

Kana, Haruka, and Chiaki are rather happy with their library, but they realise a library alone is rather useless. A `main()` function is necessary to test their work and run a calculation. However Chiaki objects that editing the `main()` function each time they need to run a task is really not convenient.

Filling stuck the three sisters decide to take a break and go out for dinner. When they return to their dorm they want to give up on their idea and focus on the next homework. This is at this very moment that Haruka jumps off her chair and says "Eureka! We should have a `main()` function similar to the one in the homework template." She then further explains that they could write a small program that would take some input parameters from the command line, a bit like `gcc` or the homework template. As this ideas makes sense to all of them they decide to study the template in more details.

After playing with the `main.c` file they fully understand how it works and decide to bring this idea to write a CLI calculator. A few minutes later they have already agreed on the functionalities and command lines arguments to use. The only thing left for them to consider and precisely define is the format of the input expression.

```
sh $ ./calculator --help
usage: ./calculator [option | file | -]

Options:
-h      : print this help message and exit (also --help)
-c str  : evaluate str and print the result to stdout

Modes:
-       : read expressions from stdin and print results interactively (default)
file    : read expressions from file and print all results
```

Help the three sisters to

- Fully grasp how the `main.c` file from the homework template works;
- Write a `main.c` file that would allow their calculator to run as expected;
- Precisely define a simple expression format to input operations from the terminal;

2.4 A better library

The three sisters are very proud of their achievement and as Jane is a bit less busy they decide to show her their work. She is very impressed by what they have learnt and done without needing her help.

Jane also points out a few aspects of their work that can be improved. She explains that the `pow()` function is not optimized for integers, and as such can sometimes be slow or even inaccurate. Another limitation of their work is the difficulty to chain operations: integers are expected as input but their function outputs a double, so the result cannot be used as an input.

Help the three sisters to

- Write their own `pow()`, `l`, `m`, and `^` functions;
- Improve their `evaluate` function such that operations can be chained;