CalcVox - Journal 4

Ty Gillespie

March 11, 2024

Goals:

1. Fix more bugs, primarily related to decimals crashing the calculator when used in exponents and when missing trailing zeros.
2. Research how to integrate menu code into the main Arduino `loop()` function.
3. Work on Journal 4.
4. Fix a few remaining bugs in Evox, the expression evaluator.
5. Research CAS (Computer Algebra Systems), thanks to the suggestion of an advisor.
6. Meet with advisor.
7. Work more on core calculator components.
8. Write a basic outline of a menu system.

My Research and What I Learned: *Fixing bugs in Evox:* Shortly after writing the initial implementation of Evox, our expression evaluator, Grant found a few bugs with it. A good portion of the last couple weeks for me has been tracking them down and fixing them. The bugs were as follows: *Decimals without leading zeros don't work:* The first problem discovered was that trying to evaluate something like $1 + .1$ wouldn't work. $1 + 0.1$ would, however. I sat down and tackled this, and ended up fixing another bug at the same time (decimals in exponents always evaluated to 0). Here was the code:

```
#include "evox.h"

int get_precedence(char op) {
        switch (op) {
        case '^':
                return 3;
        case '*':
        case '/':
                return 2;
        case '+':
        case '-':
                return 1;
        default:
                return 0;
        }
}

double apply_operation(double a, double b, char op) {
        switch (op) {
        case '^':
                return std::pow(a, b);
        case '*':
                return a * b;
        case '/':
                return a / b;
        case '+':
                return a + b;
        case '-':
                return a - b;
        default:
                return 0;
        }
}

double evox(const std::string& expression) {
        std::stack<double> numbers;
        std::stack<char> operators;
        bool reading_number = false;
        double current_number = 0;
        double decimal_place = 10; // To handle decimal digits
        int open_parentheses_count = 0;
        for (char c : expression) {
                if (isdigit(c) || c == '.') { // Allow digits and decimal point
                        if (c == '.') {
                                decimal_place = 0.1;
                        } else {
                                if (reading_number) {
                                        current_number += (c - '0') * decimal_pl
                                        decimal_place *= 0.1;
                                } else {
                                        current_number = current_number * 10 + (
                                }
                        }
                        reading_number = true;
                } else {
                        if (reading_number) {
                                numbers.push(current_number);
                                current_number = 0;
                                decimal_place = 10; // Reset decimal place
                                reading_number = false;
                        }
                        if (c == '(') {
                                operators.push(c);
                                open_parentheses_count++;
                        } else if (c == ')') {
                                if (open_parentheses_count == 0) {
                                        throw std::runtime_error("Mismatched␣par
                                }
                                while (!operators.empty() && operators.top() !=
                                        char op = operators.top();
                                        operators.pop();
                                        double b = numbers.top();
                                        numbers.pop();
                                        double a = numbers.top();
                                        numbers.pop();
                                        numbers.push(apply_operation(a, b, op));
                                }
                                if (!operators.empty()) {
                                        operators.pop();
                                        open_parentheses_count--;
                                } else {
                                        throw std::runtime_error("Mismatched␣par
                                }
                        } else {
                                while (!operators.empty() && get_precedence(oper
                                        char op = operators.top();
                                        operators.pop();
                                        double b = numbers.top();
                                        numbers.pop();
                                        double a = numbers.top();
                                        numbers.pop();
                                        numbers.push(apply_operation(a, b, op));
                                }
                                operators.push(c);
                        }
                }
        }
        if (reading_number) {
                numbers.push(current_number);
        }
```

```
        if (open_parentheses_count > 0) {
                throw std::runtime_error("Mismatched parentheses: Too many openi
        }
        while (!operators.empty()) {
                char op = operators.top();
                operators.pop();
                double b = numbers.top();
                numbers.pop();
                double a = numbers.top();
                numbers.pop();
                numbers.push(apply_operation(a, b, op));
        }
        if (numbers.size() != 1) {
                throw std::runtime_error("Invalid expression");
        }
        return numbers.top();
}
```

The fix was pretty simple, I just had to do a bit of extra logic when parsing decimals. It made the code bigger, but not bloated so, and fixed two problems at once. Overall, pretty successful. *Fixing mismatched parentheses:* The next big thing was that mismatched parentheses caused the test program (and as a result the calculator hardware) to freeze. My solution to this looks like the following:

```
#include "evox.h"

int get_precedence(char op) {
        switch (op) {
        case '^':
                return 3;
        case '*':
        case '/':
                return 2;
        case '+':
        case '-':
                return 1;
        default:
                return 0;
        }
}

double apply_operation(double a, double b, char op) {
        switch (op) {
        case '^':
                return std::pow(a, b);
        case '*':
                return a * b;
        case '/':
                return a / b;
        case '+':
                return a + b;
        case '-':
                return a - b;
        default:
                return 0;
        }
}

double evox(const std::string& expression) {
        std::stack<double> numbers;
        std::stack<char> operators;
        bool reading_number = false;
        double current_number = 0;
        double decimal_place = 10; // To handle decimal digits
        int open_parentheses_count = 0;
        for (char c : expression) {
                if (isdigit(c) || c == '.') { // Allow digits and decimal point
                        if (c == '.') {
                                decimal_place = 0.1;
                        } else {
                                if (reading_number) {
                                        current_number += (c - '0') * decimal_pl
                                        decimal_place *= 0.1;
                                } else {
                                        current_number = current_number * 10 + (
                                }
                        }
                        reading_number = true;
                } else {
                        if (reading_number) {
                                numbers.push(current_number);
                                current_number = 0;
                                decimal_place = 10; // Reset decimal place
                                reading_number = false;
                        }
                        if (c == '(') {
                                operators.push(c);
                                open_parentheses_count++;
                        } else if (c == ')') {
                                if (open_parentheses_count == 0) {
                                        throw std::runtime_error("Mismatched par
                                }
                        while (!operators.empty() && operators.top() !=
                                char op = operators.top();
                                operators.pop();
                                double b = numbers.top();
                                numbers.pop();
                                double a = numbers.top();
                                numbers.pop();
                                numbers.push(apply_operation(a, b, op));
                        }
                        if (!operators.empty()) {
                                operators.pop();
                                open_parentheses_count--;
                        } else {
                                throw std::runtime_error("Mismatched␣par
                        }
                } else {
                        while (!operators.empty() && get_precedence(oper
                                char op = operators.top();
                                operators.pop();
                                double b = numbers.top();
                                numbers.pop();
                                double a = numbers.top();
                                numbers.pop();
                                numbers.push(apply_operation(a, b, op));
                        }
                        operators.push(c);
                }
        }
}
if (reading_number) {
        numbers.push(current_number);
}
if (open_parentheses_count > 0) {
        throw std::runtime_error("Mismatched␣parentheses:␣Too␣many␣openi
}
while (!operators.empty()) {
        char op = operators.top();
        operators.pop();
        double b = numbers.top();
        numbers.pop();
        double a = numbers.top();
        numbers.pop();
        numbers.push(apply_operation(a, b, op));
}
if (numbers.size() != 1) {
        throw std::runtime_error("Invalid␣expression");
        }
        return numbers.top();
}
```

This adds a bit more syntactical and conditional logic, checking for mismatched parentheses and throwing a C++ exception if they're mismatched. This is quite nice actually, it makes it so the user can do something like:

```
#include <iostream>
#include <string>
#include "evox.h"

int main(int argc, char **argv) {
        if (argc != 1) {
                try {
                        std::string expression = argv[1];
                        double result = evox(expression);
                        std::cout << result << std::endl;
                } catch (const std::exception& e) {
                        std::cerr << "Error: " << e.what() << std::endl;
                        return 1;
                }
        }
        else {
                while (true) {
                        try {
                                std::string expression;
                                std::cout << "Enter expression (or press Enter t
                                std::getline(std::cin, expression);
                                if (expression.empty()) break;
                                std::cout << evox(expression) << std::endl;
                        } catch (const std::exception& e) {
                                std::cerr << "Error: " << e.what() << std::endl;
                        }
                }
                return 0;
        }
}
```

The code I just showed is actually what I use to test the Evox library. CMake, my C/C++ build tool of choice, does have a native test runner, and I decided to take this opportunity to sit down and learn how the test runner worked, and also to make my library as good as possible. My CMake configuration looks like the following (calcvox/evox/CMakeLists.txt):

```
cmake_minimum_required(VERSION 3.15.0)

project(evox LANGUAGES CXX)
```

```
add_executable(tester tester.cpp evox.cpp)

enable_testing()
add_test(NAME TestMismatchedParentheses COMMAND tester "(2+3)*4)")
add_test(NAME TestMismatchedParentheses2 COMMAND tester "((2+3)*4")
add_test(NAME TestOrderOfOperations COMMAND tester "2+3*4")
add_test(NAME TestOrderOfOperations2 COMMAND tester "2*3+4")
add_test(NAME TestDecimalWithoutTrailingZeros COMMAND tester "0.5*2")
add_test(NAME TestDecimalExponents COMMAND tester "2^0.5")
set_tests_properties(TestMismatchedParentheses PROPERTIES PASS_REGULAR_EXPRESSIO
set_tests_properties(TestMismatchedParentheses2 PROPERTIES PASS_REGULAR_EXPRESSIO
set_tests_properties(TestOrderOfOperations PROPERTIES PASS_REGULAR_EXPRESSION "1
set_tests_properties(TestOrderOfOperations2 PROPERTIES PASS_REGULAR_EXPRESSION "
set_tests_properties(TestDecimalWithoutTrailingZeros PROPERTIES PASS_REGULAR_EXP
set_tests_properties(TestDecimalExponents PROPERTIES PASS_REGULAR_EXPRESSION "1.
```

It's not super complicated, I just needed to make my tester program take the proper command line arguments, while retaining the existing functionality (i.e. an interactive, shell-like calculator experience on Windows). *Menus:* Another big thing I worked on over the past few weeks is menus. Menus are going to be used everywhere in this calculator, for everything from changing angle measurements to statistics, so they're important to get right. So far, I've made pretty good progress. The menus can (optionally) wrap, specify a custom click sound (i.e., a sound when the user moves through the menu), set the starting position, speak an intro message, choose if they're horizontal, vertical, or both, bind custom callbacks, and many more features. It works fully when tested on Windows. The current obstacle is that I can't get it to run on the microcontroller. I need to find a way to integrate it with the existing `loop()` function. My thought on how to do this is to have a `std::vector<uint> keys_pressed;` and just make a `bool key_pressed(uint key)` function. That should make checking for key presses globally way easier, thus making the menu able to work.

Accomplishments:

1. Wrote a basic expression evaluator in idiomatic C++, supporting all the basic functions of a calculator, as well as PEMDAS.
2. Began working on a new method of speech output using a second microcontroller.
3. Wrote some software for the new speech chip, primarily to make it compatible with the DoubleTalk protocol.
4. Completed and presented my mid-year presentation.
5. Built some components for the final calculator. Most notably, started on a super extensible menu system.
6. Squashed bugs in the expression evaluator, as well as in the software as a whole.

Progress on Timeline: Despite the setbacks I've faced in recent weeks, especially as far as bugs in the expression evaluator go as well as trying to integrate the menu code with the main `loop()` function, I still think I'm quite on track to finish the project. I deliberately gave myself ample time during this part, just because I knew I'd most likely hit some issues.

Sources:

- `https://github.com/codeplea/tinyexpr`
- `https://github.com/christian-vigh/eval`
- `https://github.com/Blake-Madden/tinyexpr-plusplus`
- `https://stackoverflow.com/questions/5115872/what-is-the-best-way-to-evaluate-mathematic`
- `https://codeplea.com/tinyexpr`
- `https://www.codespeedy.com/expression-evaluation-in-cpp/`
- `https://www.geeksforgeeks.org/expression-evaluation/`
- `https://www.cppstories.com/2021/evaluation-order-cpp17/`
- `https://stackoverflow.com/questions/35845566/operator-precedence-in-python-pemdas`
- `https://en.cppreference.com/w/cpp/container/stack`