CalcVox - Journal 3

Ty Gillespie

February 12, 2024

Goals:

1. Port the prototype code to use Talkie. Goal changed, now using a standalone speech chip.
2. Assist with firmware of a dedicated speech chip.
3. Begin making the speech chip's protocol conform to the protocol of the Doubletalk LT, a hardware speech synthesizer commonly used by many other blind people.
4. Work on mid-year presentations.
5. Write super basic expression evaluator that supports the basic mathematical functions in C++.
6. Build more components for the higher-level calculator interface (e.g. menus, history, etc.).
7. Work on the next journal.

My Research and What I Learned: *Writing an expression evaluator:* Shortly after moving over to the Teensy, we noticed a problem. The expression evaluation library we were previously using (Tinyexpr), wouldn't compile due to a missing function in the C/C++ runtime. More specifically, it was the _times() function. The initial attempt to fix this involved monkey patching the _times() function. This was only ever mentioned in one place, and it (unsurprisingly) didn't work.

```
extern "C" {
    // define _times to return 0
    int \_times(struct tms* buf) {
        return 0;
    }
}
```

This most definitely didn't work, it made the board crash right as it booted! The next week or so was basically nothing but me looking at the internals of

a lot of libc implementations, attempting to find _times() in a form that could run on our board. I never did manage though, and tinyexpr never did run on CalcVox after H1. However, despite all of this, I was still very motivated to make it work. As such, I started the painstaking process of writing my own expression evaluator. It's a tedious and annoying process, but amazing progress has been made so far. I first started out with something like this. It worked, but didn't support PEMDAS, or expressions like 1+1+1 if not wrapped in parentheses. It looks like:

```
// Evaluates basic math expressions. Currently nothing advanced is supported, it's basic
double evox(const std::string& expression) {
    std::stack<double> numbers;
    std::stack<char> operators;
    for (char c : expression) {
        if (isdigit(c)) {
            // Stupid and weird hack to convert a char to an int without a cast.
            numbers.push(c - '0');
        } else if (c == '+' || c == '-' || c == '*' || c == '/' || c == '\^{}') {
            while (!operators.empty() && operators.top() != '(' && (c == '\^{}' || c ==
                char op = operators.top();
                double operand2 = numbers.top();
                numbers.pop();
                double operand1 = numbers.top();
                numbers.pop();
                switch (op) {
                case '+':
                    numbers.push(operand1 + operand2);
                    break;
                case '-':
                    numbers.push(operand1 - operand2);
                    break;
                case '*':
                    numbers.push(operand1 * operand2);
                    break;
                case '/':
                    numbers.push(operand1 / operand2);
                    break;
                case '\^{}':
                    numbers.push(pow(operand1, operand2));
                    break;
                }
            }
            operators.push(c);
        } else if (c == '(') {
            operators.push(c);
        } else if (c == ')') {
```

```
            while (!operators.empty() && operators.top() != '(') {
                char op = operators.top();
                operators.pop();
                double operand2 = numbers.top();
                numbers.pop();
                double operand1 = numbers.top();
                numbers.pop();
                switch (op) {
                case '+':
                    numbers.push(operand1 + operand2);
                    break;
                case '-':
                    numbers.push(operand1 - operand2);
                    break;
                case '*':
                    numbers.push(operand1 * operand2);
                    break;
                case '/':
                    numbers.push(operand1 / operand2);
                    break;
                case '\^{}':
                    numbers.push(pow(operand1, operand2));
                    break;
                }
            }
            operators.pop();
        }
    }
    while (!operators.empty()) {
        char op = operators.top();
        operators.pop();
        double operand2 = numbers.top();
        numbers.pop();
        double operand1 = numbers.top();
        numbers.pop();
        switch (op) {
        case '+':
            numbers.push(operand1 + operand2);
            break;
        case '-':
            numbers.push(operand1 - operand2);
            break;
        case '*':
            numbers.push(operand1 * operand2);
            break;
        case '/':
```

```
            numbers.push(operand1 / operand2);
            break;
        case '\^{}':
            numbers.push(pow(operand1, operand2));
            break;
        }
    }
    return numbers.top();
}
```

As you can see, it worked, but the code had a lot of the same lines repeating, and it also had the problems I listed above. I took this base, and managed to get basically all expressions to not crash. It looked like:

```
#include "evox.h"

double evox(const std::string& expression) {
    std::stack<double> numbers;
    std::stack<char> operators;
    bool reading_number = false;
    double current_number = 0;
    for (char c : expression) {
        if (isdigit(c)) {
            if (reading_number) {
                current_number = current_number * 10 + (c - '0');
            } else {
                current_number = c - '0';
                reading_number = true;
            }
        } else {
            if (reading_number) {
                numbers.push(current_number);
                current_number = 0;
                reading_number = false;
            }
            if (c == '+' || c == '-' || c == '*' || c == '/' || c == '\^{}') {
                while (!operators.empty() && operators.top() != '(' && (c == '\^{}' || c
                    char op = operators.top();
                    double operand2 = numbers.top();
                    numbers.pop();
                    double operand1 = numbers.top();
                    numbers.pop();
                    switch (op) {
                    case '+':
                        numbers.push(operand1 + operand2);
                        break;
                    case '-':
```

```
                numbers.push(operand1 - operand2);
                break;
            case '*':
                numbers.push(operand1 * operand2);
                break;
            case '/':
                numbers.push(operand1 / operand2);
                break;
            case '\^{}':
                numbers.push(pow(operand1, operand2));
                break;
            }
            operators.pop();
        }
        operators.push(c);
    } else if (c == '(') {
        operators.push(c);
    } else if (c == ')') {
        while (!operators.empty() && operators.top() != '(') {
            char op = operators.top();
            operators.pop();
            double operand2 = numbers.top();
            numbers.pop();
            double operand1 = numbers.top();
            numbers.pop();
            switch (op) {
            case '+':
                numbers.push(operand1 + operand2);
                break;
            case '-':
                numbers.push(operand1 - operand2);
                break;
            case '*':
                numbers.push(operand1 * operand2);
                break;
            case '/':
                numbers.push(operand1 / operand2);
                break;
            case '\^{}':
                numbers.push(pow(operand1, operand2));
                break;
            }
        }
        operators.pop();
    }
}
```

```
    }
    if (reading_number) {
        numbers.push(current_number);
    }
    while (!operators.empty()) {
        char op = operators.top();
        operators.pop();
        double operand2 = numbers.top();
        numbers.pop();
        double operand1 = numbers.top();
        numbers.pop();
        switch (op) {
        case '+':
            numbers.push(operand1 + operand2);
            break;
        case '-':
            numbers.push(operand1 - operand2);
            break;
        case '*':
            numbers.push(operand1 * operand2);
            break;
        case '/':
            numbers.push(operand1 / operand2);
            break;
        case '\^{}':
            numbers.push(pow(operand1, operand2));
            break;
        }
    }
    return numbers.top();
}
```

As you can see, there were changes, but they were mostly towards the end, and just adding and adding more lines. PEMDAS still didn't work. As such, I did an absolutely massive refactor, and ended up with this:

```
#include "evox.h"

// Utility function to get the precedence of a character according to PEMDAS.
int get_precedence(char op) {
    switch (op) {
        case '\^{}':
            return 3;
        case '*':
        case '/':
            return 2;
        case '+':
```

```
        case '-':
            return 1;
        default:
            return 0;
    }
}

// Tiny utility function to apply operations to operands. This mainly exists so I don't l
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;
    for (char c : expression) {
        if (isdigit(c)) {
            double num = c - '0';
            numbers.push(num);
        } else {
            if (c == '(') {
                operators.push(c);
            } else if (c == ')') {
                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));
                }
                operators.pop(); // Remove '('
```

```
            } else {
                while (!operators.empty() && get_precedence(operators.top()) >= get_prec
                    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);
            }
        }
    }

    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));
    }
    return numbers.top();
}
```

This actually complicated things a great bit. I added a function to get the order of operations for a particular operand, as well as to get rid of all the copy/pasted code for all the operations. I also changed how it handles parentheses to handle PEMDAS. It works! I thought. And it did work. For a while. Until I realized that long numbers (i.e. numbers greater than one character like 39) would get taken to mean 3, and 9. As such, I had to get a bit creative. I basically ended up implementing a portion of a programming language's lexer while sitting at the lunch table with my friends. The final result looked like this:

```
#include "evox.h"

// Utility function to get the precedence of a character according to PEMDAS.
int get_precedence(char op) {
    switch (op) {
    case '\^{}':
        return 3;
    case '*':
    case '/':
        return 2;
    case '+':
```

```
    case '-':
        return 1;
    default:
        return 0;
    }
}

// Tiny utility function to apply operations to operands. This mainly exists so I don't h
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;
    for (char c : expression) {
        if (isdigit(c)) {
            if (reading_number) {
                current_number = current_number * 10 + (c - '0');
            } else {
                current_number = c - '0';
                reading_number = true;
            }
        } else {
            if (reading_number) {
                numbers.push(current_number);
                current_number = 0;
                reading_number = false;
            }
            if (c == '(') {
                operators.push(c);
```

```
            } else if (c == ')') {
                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));
                }
                operators.pop();
            } else {
                while (!operators.empty() && get_precedence(operators.top()) >= get_prec
                    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);
    }
    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));
    }
    return numbers.top();
}
```

I implemented a boolean flag to tell if we're currently reading a number or not. If we are, we tokenize the input, and combine all the numbers. And that did the trick!

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.

Progress on Timeline: Although my timeline has changed greatly, I'm now incredibly happy with how it turned out. It's now looking like we're going to be able to implement a scientific calculator by the deadline, that does everything we want! And, because of how it's going to be designed, we'll be able to expand it super easily. Doing things like writing my own expression evaluator makes things like this even easier, albeit take more time.

Sources:

- https://github.com/codeplea/tinyexpr
- https://github.com/Blake-Madden/tinyexpr-plusplus
- https://codeplea.com/tinyexpr
- https://www.codespeedy.com/expression-evaluation-in-cpp/
- https://www.geeksforgeeks.org/expression-evaluation/
- https://stackoverflow.com/questions/5115872/what-is-the-best-way-to-evaluate-mathematical-expressions-in-c
- https://en.cppreference.com/w/cpp/language/eval$_o rderhttps://stackoverflow.com/questions/358 precedence - in - python - pemdas$

- https://en.cppreference.com/w/cpp/container/stack