CalcVox - Journal 5}

Goals:

1. Rewrite input layer to allow for more modular code, and more systems.
2. Majorly clean up the code and attack some nasty technical debt.
3. Polish up the new input layer code, and integrate menus.
4. Write a complete history system, complete with searching, indexing, and using modern C++.
5. Start on a project wiki, to document all aspects of the project.
6. Add repeat key functionality, and reorganize/optimize code.
7. Work on presentation.
8. Work on journal 5.

My research and what I learned:

.ino vs .cpp files

For many months now, whenever I tried to make the project have multiple source files (.cpp), we hit many errors that I simply could not figure out. For a while the code was small enough that this didn't really matter, but eventually (when implementing history) it became such an issue that I did a deep dive on the Teensy forums, and it turns out that .cpp files are compiled separately, while .ino files are all compiled at the same time. Renaming all the .cpp files to .ino fixed all the issues!

Implementing history

A vital portion of any calculator is the history (i.e. being able to review previous equations, insert portions of previous equations etc.). After a bit of tinkering, I managed to come up with something I quite like:

history.ino:

```
#include "history.h"

history::history()
    : pos(0) {}

void history::clear(bool silent) {
    buff.clear();
    pos = 0;
    if (!silent)
     speak("History cleared");
}

// Adds an item to the history given an equation and result
void history::add_item(const std::string& equation, const std::string& result) {
    buff.push_back({equation, result});
    pos = buff.size() - 1;
}

bool history::scroll_down() {
    if (buff.empty() || pos >= (buff.size() - 1)) return false;
    pos++;
    std::string message = buff[pos].equation + " = " + buff[pos].result;
    speak(message);
    return true;
}

bool history::scroll_up() {
    if (buff.empty() || pos <= 0) return false;
    pos--;
    std::string message = buff[pos].equation + " = " + buff[pos].result;
    speak(message);
    return true;
}

history_item* history::operator[](size_t index) {
    return &buff[index];
}

bool history::set_pos(int new_pos, bool silent) {
    if (pos < 0 || pos >= (buff.size() - 1)) return false;
    pos = new_pos;
    if (!silent) {
     std::string message = buff[pos].equation + " = " + buff[pos].result;
     speak(message);
    }
    return true;
}

// Moves left, to the previous type of item in the history
void history::move_left() {
    if (type == RESULT) {
     type = EQUATION;
     speak("Equation");
    }
}

// Moves right, to the next type of item in the history
void history::move_right() {
    if (type == EQUATION) {
     type = RESULT;
     speak("Result");
    }
}

// Sets the type of the current item in the history
void history::set_type(history_item_type new_type, bool silent) {
    type = new_type;
    if (!silent) {
     std::string type_str = new_type == EQUATION ? "Equation" : "Result";
     speak(type_str);
    }
}

// Inserts a portion of the focused item into the current equation, depending on the type s
void history::insert_item() {
    if (type == EQUATION) {
     current_equation += buff[pos].equation;
    } else {
     current_equation += buff[pos].result;
    }
    speak("Inserted");
    pos = buff.size() - 1;
}
```

and the header, history.h:

```
#pragma once

#include "calcvox.h"
#include <string>
#include <vector>

// Represents the types of items to be inserted into the history.
enum history_item_type {
    EQUATION,
    RESULT
};

// Represents a single item in the calculation history.
struct history_item {
    std::string equation;
    std::string result;
};

class history {
public:
    history();
    void clear(bool silent = false);
    void add_item(const std::string& equation, const std::string& result);
    bool scroll_up();
    bool scroll_down();
    history_item* operator[](size_t index);
    bool set_pos(int new_pos, bool silent = false);
    void move_left();
    void move_right();
    void set_type(history_item_type new_type, bool silent = false);
    void insert_item();

private:
    std::vector<history_item> buff;
    // Current position in the buffer.
    std::vector<history_item>::size_type pos;
    history_item_type type;
};
```

There's a lot going on there, but the basics are: * The history maintains a list of history_items, which contain info about the given equation, and the computed result (cached for speed). * There are functions for going to the previous item, the next item, or back and forth between what you want to insert. * You can insert either the current equation, or the result of said equation. Switch between them with the left and right buttons. * I also used my C++ knowledge to provide a bit of syntax sugar, allowing you to index into the history directly as if it were a standard C++ array/vector/etc. * There are also methods to clear the history, jump to a particular position, and to insert items.

Overall, I'm incredibly happy with this history system. I imagine some more work will be done in the future, for example to make history persist across device reboots, but it works incredibly well in its current state.

I learned something incredibly interesting while making this history system. I initially started out speaking the current equation and the result concatenated together using `TalkSerial.printf()`. However, I noticed that the history was painfully slow when scrolling, and just could not figure out why. Eventually, I switched to using an std::string and passing it to speak directly, and that sped it up significantly! I guess the serial's print formatter is super intensive :(

Handling input

When handling input, I wanted a way to detect if keys are pressed in more than just the loop() function. Having everything there worked for a bit, but it very quickly fell apart when I started porting my C++ menu system (originally written for Windows) to the calculator.

I initially started out with a function, called key_pressed, that would call `keypad.getKey()`, and return true or false depending on if the specified key was pressed or not. However, this didn't work, for some reason we only ever got key down events, so the firmware thought that the key was constantly held down!

After a bit more research, we discovered a function in the keypad library, called `keypad.isPressed()`. You just pass a char to it, and it tells you if that key was pressed or not. Exactly what we wanted.

Accomplishments:

1. Wrote a full history system!
2. Majorly cleaned up the code, making it much easier to maintain, especially long-term.
3. Started on a wiki, making the project easier to pick up for other contributors.
4. Rewrote the input layer to allow for more flexible code.
5. Fixed the repeat key's function, and made it actually useful.

Progress on Timeline:

We're incredibly close to presentations now, but I feel ready. We've had many setbacks and many successes, and I feel ready to show it to the world.

Sources

- https://www.pjrc.com/teensy/troubleshoot.html
- https://forum.arduino.cc/t/solved-i-think-strange-interrupt-behavior-on-teensy-4-1/1001552
- https://en.cppreference.com/w/cpp/language/operators
- https://stackoverflow.com/questions/4421706/what-are-the-basic-rules-and-idioms-for-operator-overloading
- https://isocpp.org/wiki/faq/operator-overloading
- https://en.cppreference.com/w/cpp/language/default_arguments
- https://www.arduino.cc/reference/en/libraries/keypad/