Patrick Letts

December 4th, 2017

CS 480

NMU F17

Therapist: a Brainf\* interpreter

Brainf\* is a very simple esoteric language created by Urban Müller. Simple beyond the point of usefulness with only eight single-character commands: <>-+,.[]; these commands allow for the navigation and changing of an array and its contents, the unofficial standard being at least 30,000 cells in size, all initialized to zero. ‘<’ and ‘>’ respectively decrement and increment a data pointer to track a program’s point in the array. ‘+’ and ‘–‘ respectively increment and decrement the value in the current cell. ‘,’ and ‘.’ allow for single-byte input and output of the current cell. ‘[‘ and ‘]’ allow for looping. All other characters should be ignored. Even given this simplicity, it is considered Turing-complete.

Here’s a table of C equivalents for each instruction, courtesy of [https://en.wikipedia.org/wiki/Brainf\*](https://en.wikipedia.org/wiki/Brainfuck)

|  |  |
| --- | --- |
| > | ++ptr; |
| < | --ptr; |
| + | ++\*ptr; |
| - | --\*ptr; |
| . | putchar(\*ptr); |
| , | \*ptr=getchar(); |
| [ | while(\*ptr){ |
| ] | } |

Here’s a Hello World, also courtesy of Wikipedia

++++++++[>++++[>++>+++>+++>+<<<]>+>+>→>+[<]]>>.>---.+++++++..+++.>>..<.+++.------.--------.>>+.>++.

**Overview**

For my senior project I created a Brainf\* interpreter. Naturally, this practice is trivial to the casual observer, therefore much more work was required of me. I ran across Brainf\* some time ago and found it, as many do, interesting but ultimately off-putting. I chose Brainf\* as my language of choice mostly because I was interested in learning about interpreters, but had zero experience with them, so the extreme simplicity of it attracted me. I was also drawn in by the opportunity to garner more experience with project management and commonly used tools.

Thanks to the inherent simplicity of Brainf\*, I knew I’d need more complex features, which include a shell and two debuggers. I learned a lot throughout the development of this project, of myself, programming, Brainf\*, and the development process in general. Therapist provides, in essence, a collection of tools for Brainf\* development. The simplest usage of it involves feeding a file with Brainf\* source code to a machine that can recognize it, allowing immediate processing.

After this, a REPL shell is provided, allowing a user to responsively interact with the machine, the interface between the two making use of an environment. This gives the user a level of abstraction through the pairing of names and Brainf\* procedures. Also provided are two debuggers. The first makes use of the C Curses library, and is started with a flag to the interpreter. This debugger provides basic debugging functionality, allowing the user to manually step through code char by char or immediately advance to the next breakpoint (or EOF). The second debugger is written with GNOME’s gtkmm, a C++ wrapper for GTK+. Atop the functionality provided by the previouse debugger, the most notable feature of this debugger is the ability to insert and edit Brainf\* source code while the program is being debugged.

**Learning Objectives**

I had a few particular personal objectives in mind when I started this project, goals which would reflect my learning experience, but couldn’t quite be quantified. These goals involve:

* C++
* Curses
* Project management
* GTK

I’m certainly confident asserting that I’ve met these objectives, learned many new things, particularly with respect to learning whole new libraries. I also learned much more about my own abilities, especially time management, self-direction, and standards.

**Technologies Used**

At this point in my progression through the CS program, C++ is generally my language of choice, though as this semester progressed, it certainly became much less so. Taking CS 322 and 426 at the same time provided much inspiration for some of my design choices, and if I began this same project one semester later, I would have entirely preferred another language, probably Smalltalk.

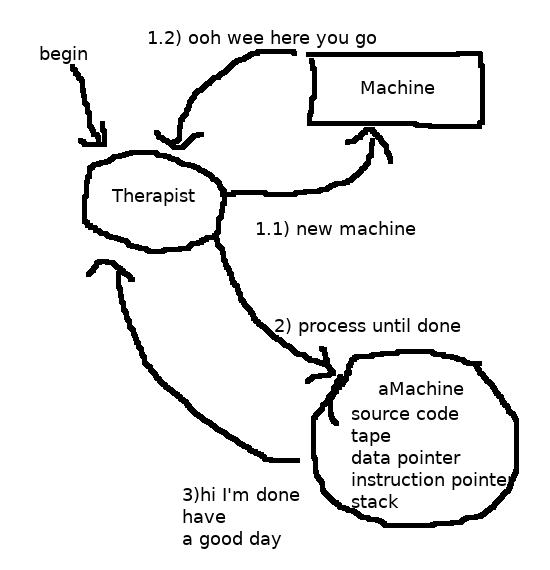
Particular tools that I used in development of my project include Curses, GTK, Git, and to an extent, the tools that I even wrote for this. Naturally, when writing an interpreter, a grasp of the language at hand is necessary, and actively using the tools I created for this helped me understand the subtleties behind Brainf\* and common programming practices with it.

**Organization, Structure, and Design Goals**

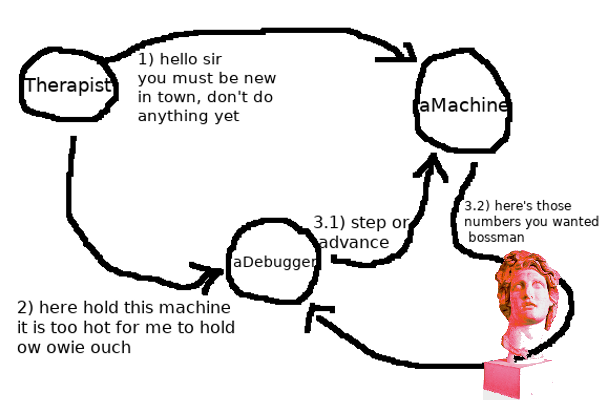
As soon as I began to properly work on this project, I knew good design strategies would be fundamental to my development process, especially regarding the core of the interpreter.

The core functionality of this project revolves around the machine class; in the end, nearly everything else is just an interface to it. Each tool I’ve written follows a similar path to working with Brainf\* code; the tool at hand will create a new Brainf\* machine object, to which messages are passed. The machine may be provided with a number of arguments for customization, including ASCII IO, tape length, and a string of Brainf\* source, though defaults are provided otherwise. Now for some elaboration.

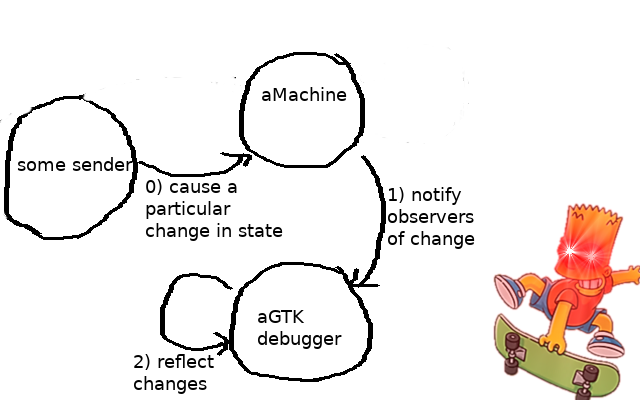
To start with the basic form of the interpreter, the Therapist class is provided the name of a Brainf\* source file, and a number of flags to tailor some qualities of the machine to a particular program. After this, the interface will send a process message to the machine, and immediately evaluate the provided source code. Here’s a simple diagram of this interaction:



We can also make use of the Curses debugger from this system. The Curses debugger is started as a flag to the Therapist, which begins as previously described, but then passes on all control of the newly instantiated machine. From here, the user can interact with the debugger, which handles message passing to the machine. Generally speaking, the most common action in this model is telling the machine to process only the next operation, and retrieve changes in the machine’s state to update the screen. Something like this, sans previously described actions:

****

Before I began working with GTK, I expected the second debugger’s control flow to resemble the curses debugger pretty closely, although that assumption couldn’t be further from the truth, certainly a good thing. Before this semester, I was only vaguely aware of the term ‘design patterns’; later this semester the application of them was an ambition of mine. Unfortunately, my inexperience coupled with some aspects of C++ had quite expertly dashed my hopes with regard to this; I wanted to employ some more interesting code in the machine class especially, and I’ll concede that it wasn’t good enough, and there’s work in there I’m not very proud of. Thankfully, GTK naturally requires usage of the observer pattern, manifesting in signal handlers, allowing widgets to be connected to a callback function; the ‘subject’ class emits a signal to all its observers upon particular events, which is followed by the observers calling the functions they’ve been connected to. GTK uses this in myriad applications, mostly to allow widgets some more functionality; certainly one of the most valuable things a programmer can make use of in GTK. Upon creation of the GTK application, widgets are connected to callback functions, and then the application just has to listen for updates from its subjects. I recognized the importance of this power, and immediately began using it to implement some custom signals of my own; the machine class contains a few particular points where some of the most important changes to itself occur, these are where observers are notified of these changes the moment they happen, allowing the debugger to know when to make requests of the machine. Here’s a simplified visual representation of the process:



Development of my REPL shell was, to be honest, a decently straightforward task, mostly. At this point in my development process, I was lucky enough to have been taking CS 426 at the same time, concerning design choices. Thanks to CS426, I had already been familiar with implementing the features of readline and history, allowing my shell to be much more than just looping getline().

I knew a plain shell would be a bit boring, and too easy; I could already have a machine process code, and all I really needed was to make some small changes to the machine class to allow for repeated keyboard input. The first thing that sprang to mind was abstraction; it’s an essential concept, and Brainf\* offers just about none of it. To allow for this, the shell uses an intermediate environment class. This class’s main purpose is to recognize and handle attempts to bind a name to a string of Brainf\* code, as well as recognize usage of an existing binding so it may be expanded before being sent to the machine for processing. I’ll spare you the drawing.

**Challenges**

This project presented numerous challenges throughout the development process. Brainf\* itself being the first, thanks to its inherent difficulty; how can I tell a computer what a program does when I don’t know either? In this sense, I spent a healthy amount of time working with the core interpreter, knowing the biggest challenge there would be loops, which are certainly the most difficult concepts in the language itself to implement. In Brainf\*, loops are enabled by the square bracket characters. If the tape cell currently pointed at is non-zero, the current instruction pointer is saved to a stack, and code execution is continued until the matching ‘]’ is found. Its C equivalent could be presented as:

while(\*ptr){ }

The challenge presented by this implementation is skipping loops altogether; it’s entirely possible for a particular loop to never be entered, which provides another challenge in an efficient bracket challenge algorithm, which took me quite some time to work out. I found this especially intimidating at the beginning of this semester, causing my own interpreter to actually implement the equivalent of a C do-while loop for a little while.

Many of the other challenges I faced involved the libraries I was using. I have had some previous experience with the Curses library, though it was minimal and I look back on that code with a degree of shame; a particular roadblock that I look back on fondly when creating my curses debugger was working with IO. My BF machine had initially implemented rather plain IO functionality, not allowing for IO outside of the terminal it was running in. While working on the debugger, I quickly realized this wouldn’t do, so I had to modify the machine for some smart ~~workarounds~~ solutions to this.

Regarding work with my GTK debugger, many major challenges involved just learning GTK. Honestly, after I got a basic grasp of its usage, I frequently found myself overwhelmed by just how much functionality it provides; this paused my progress for just about every feature of the debugger, having to decide exactly which widget I wanted for a particular UI element, and then choosing an appropriate container type for it. I was also frequently thinking of a particular functionality that wasn’t important but I would love to have, often preventing progress on more important features; there’s just too much cool stuff. After this, my first goal was to copy the functionality of the curses debugger, providing more unique challenges, again forcing me to make adjustments to my machine, though ultimately it allowed for much better design of the GTK debugger.

**Grading**

Here’s my original grading scale, highlighting goals I met in yellow:

* **Interpreter – 10 points**
  + Basic interpreter – 5
  + Object-oriented design – 3
  + Test cases – 2
* **Debuggers** – **10 points**
  + Curses library – 4 points
    - Basic stepping through input code – 2 points
    - Display and update BF code, tape, and stack – 2 points
  + GTK library – 6 points
    - Basic stepping through input code – 2 points
    - Directly manipulate cells on the tape – 1 point
    - Interface for more powerful stepping, allowing for breakpoints – 3 points
* **REPL – 10 points**
  + Repeatedly accept and evaluate BF code from the keyboard – 3 points
  + Show tape and update as necessary – 2 points
  + Accept multiple lines before evaluating when a loop is opened – 2 points
  + Allow user to define, name, and call procedures from sections of BF code – 3 points
* **Automatic compilation and installation** – **6 points**
  + Provide a Makefile with interpreter source code – 2 points
  + Provide an entire package for installation, package manager TBD – 4 points
* **Man page – 4 points**
  + Provide a detailed man page describing the features and usage of the interpreter – 4 points
* **Command line flags – 10 points**
  + Toggle ASCII I/O – 4 points
  + Toggle un/signed cells – 4 points
  + Define tape length – 2 points

According to this scale, I’d have 43/50 points, putting me at a healthy B. Looking at my scale and reflecting on the actual difficulty of the work that I did, I’d absolutely reallocate more points to the debuggers; they were certainly the most difficult, and admittedly, I gave too many points to goals like the man pages – goals that weren’t integral to the code. To juxtapose my work on man pages with the Curses debugger, to which I allocated the same number of points, I absolutely overestimated difficulty of the former.

I did enjoy writing the man pages though, and they did provide a unique challenge; the process of having to describe the usage of my programs forced me to ask myself if I really knew what was going on in my code. It also allowed me to really reflect on what I had done, and gave me an opportunity to consider new features, or rework existing ones. It forced me to look at my programs as a user, not the person who was familiar with every nook and cranny of the code.

­

**Conclusion and Final Notes**

In the end, I enjoyed working on this and I’m happy with how it turned out. Unfortunately, I wasn’t able to meet all of my goals, but I feel it’s made up for by exceeding what I expected of myself for some others. My ultimate personal objective was to prove to myself that I have the capacity to follow through with a long term goal, supported by (more or less) only my own self direction, and I certainly feel I’ve done that.

I managed to squeeze in some extra features not included on my original scale. Those are highlighted in green, while original goals are highlighted in yellow. I also managed to meet a couple more original goals in the last few days. This would bring my final score up to a 50/50.

* **Interpreter – 10 points**
  + Basic interpreter – 5
  + Object-oriented design – 3
  + Test cases – 2
* **Debuggers** – **10 points**
  + Curses library – 4 points
    - Basic stepping through input code – 2 points
    - Display and update BF code, tape, and stack – 2 points
    - Breakpoint support – 2 points
  + GTK library – 6 points
    - Basic stepping through input code – 2 points
    - Directly manipulate cells on the tape – 1 point
    - Interface for more powerful stepping, allowing for breakpoints – 3 points
    - Modification of running BF code – 2 points
* **REPL – 10 points**
  + Repeatedly accept and evaluate BF code from the keyboard – 3 points
  + Show tape and update as necessary – 2 points
  + Accept multiple lines before evaluating when a loop is opened – 2 points
  + Allow user to define, name, and call procedures from sections of BF code – 3 points
  + History support – 1 point
  + Multiline input – 1 point
* **Automatic compilation and installation** – **6 points**
  + Provide a Makefile with interpreter source code – 2 points
  + Provide an entire package for installation, package manager TBD – 4 points
* **Man page – 4 points**
  + Provide a detailed man page describing the features and usage of the interpreter – 4 points
* **Command line flags – 10 points**
  + Toggle ASCII I/O – 4 points
  + Toggle un/signed cells – 4 points
  + Define tape length – 2 points