[**Virtual machine in C**](https://web.archive.org/web/20200121100942/https:/blog.felixangell.com/virtual-machine-in-c)

**Hacker News Dicussion**  
[Article discussion on Hacker News](https://web.archive.org/web/20200121100942/https:/news.ycombinator.com/item?id=9762054)

**Introduction**[**#**](https://web.archive.org/web/20200121100942/https:/blog.felixangell.com/virtual-machine-in-c/#introduction_2)

Here’s the GitHub to show what we’ll be making. You can also compare your code to this repository in case you have any errors: [GitHub Repository](https://web.archive.org/web/20200121100942/https:/www.github.com/felixangell/mac)

I felt like writing an article about building your very own virtual machine in the C programming language. I love working on lower level applications e.g. compilers, interpreters, parsers, virtual machines, etc. So I thought I’d write this article as learning how virtual machines work is a great way to introduce yourself into the general realm of lower level programming!

**Prerequisites & Notices**[**#**](https://web.archive.org/web/20200121100942/https:/blog.felixangell.com/virtual-machine-in-c/#prerequisites-amp-notices_2)

There are a few things that you need before we can continue:

* GCC/Clang/.. — I’m using clang, but you can use any modern compiler;
* Text Editor — I would suggest a text editor over an IDE (when writing C), I’ll be using Emacs;
* Basic programming knowledge — Just the basics: variables, flow control, functions, structures, etc; and
* GNU Make — A build system so we aren’t writing the same commands in the terminal over and over to compile our code

**Why should I write a Virtual Machine?**[**#**](https://web.archive.org/web/20200121100942/https:/blog.felixangell.com/virtual-machine-in-c/#why-should-i-write-a-virtual-machine_2)

Here are some reasons why **you** should write a virtual machine:

* You want a deeper understanding of how computers work. This article will help you understand what your computer does at a lower level, a virtual machine provides a nice simpler layer of abstraction. And there’s no better way to learn than build one, ey?
* You just want to learn about virtual machines because it’s fun.
* You want to learn more about how some programming languages work. For instance, various languages nowadays target virtual machines - usually written specifically for the language. Examples include the JVM, Lua’s VM, Facebook’s Hip-Hop VM (PHP/Hack), etc. There’s also quite a large abstraction, from say a C++ program to assembly for your machine. When you think about it, we take a lot for granted when we write our programs in our fancy OOP paradigm language with garbage collection and all these nice features.

**Instruction Set**[**#**](https://web.archive.org/web/20200121100942/https:/blog.felixangell.com/virtual-machine-in-c/#instruction-set_3)

We’ll be implementing our own instruction set, it will be relatively simple. I’ll briefly cover instructions like moving values from registers, or jumping to other instructions, but hopefully you’ll figure this out after you’ve read the article.

Our virtual machine will have a set of register: A, B, C, D, E, and F. These are general purpose registers, which means that they can be used for storing anything. This is as opposed to say special purpose registers, for example on x86, e.g: ip, flag, ds, …

A program will be a read-only sequence of instructions. The virtual machine is a stack-based virtual machine, which means that it has a stack we can push and pop values to, and a few registers we can use too. These are also a lot more simpler to implement than a register-based virtual machine.

Without further ado, here’s an example of the instruction set we’re going to be implementing in action. The semi-colons are comments on what the line will do.

PSH 5 ; pushes 5 to the stack

PSH 10 ; pushes 10 to the stack

ADD ; pops two values on top of the stack, adds them pushes to stack

POP ; pops the value on the stack, will also print it for debugging

SET A 0 ; sets register A to 0

HLT ; stop the program

That’s our instruction set, note that the POP instruction will print the instruction we popped, this is more of a debugging thing (ADD will push a result to the stack, so we can POP the value from the stack to verify it is there).  
I’ve also included a SET instruction, this is so you understand how registers can be accessed and written to. You can also try your hand at implementing instructions like MOV A, B (move the value A to B). HLT is the instruction to show we’ve finished executing the program.

**How does a Virtual Machine work?**[**#**](https://web.archive.org/web/20200121100942/https:/blog.felixangell.com/virtual-machine-in-c/#how-does-a-virtual-machine-work_2)

Virtual Machines are more simple than you think, they follow a simple pattern, the “instruction cycle”, which is: fetch; decode; and execute.  
First we fetch the next instruction in the instruction list or code, we then decode the instruction and execute the decoded instruction.

**Project Structure**[**#**](https://web.archive.org/web/20200121100942/https:/blog.felixangell.com/virtual-machine-in-c/#project-structure_2)

Before we start programming, we need to set-up our project. We need a folder where our project will be located, I like to keep my projects under ~/dev. Here’s how we would set up our project in the terminal. This is assuming you already have a ~/dev/ directory, but you can cd into anywhere you want your project to be.

$ cd ~/dev/

$ mkdir mac

$ cd mac

$ mkdir src

Above we cd into our ~/dev directory, or wherever you want your project to be, we make a directory (I’m calling this VM “mac”). We then cd into that directory and make our src directory, which is where our code will be located.

**Makefile**[**#**](https://web.archive.org/web/20200121100942/https:/blog.felixangell.com/virtual-machine-in-c/#makefile_2)

Our makefile is relatively straight-forward, we won’t need to separate anything into multiple files, and we won’t be including anything so we just need to compile the file with some flags:

SRC\_FILES = main.c

CC\_FLAGS = -Wall -Wextra -g -std=c11

CC = clang

all:

${CC} ${SRC\_FILES} ${CC\_FLAGS} -o mac

That should suffice for now, you can always improve it later on, but as long as it does the job, we should be fine.

**Program Instructions**[**#**](https://web.archive.org/web/20200121100942/https:/blog.felixangell.com/virtual-machine-in-c/#program-instructions_2)

Now for the Virtual Machines code. First we need to define the instructions for our program. For this, we can just use an enum, since our instructions are basically numbers from 0 - X.  
In fact, an assembler program will take your assembly files and (sort of) translate all of the ops into their number counterparts. For example, if you wrote an assembler for mac, it would translate all MOV ops into the number 0, and so on.

typedef enum {

PSH,

ADD,

POP,

SET,

HLT

} InstructionSet;

Now we can store a test program as an array. So for a test, we’ll write a simple program to add the values 5 and 6, then print them out.

Note: When I say print them out, really I’ll just make it so that when we call “pop” our virtual machine will printf the value that we pop. In reality you wouldn’t want to do this unless you’re debugging or something.

The instructions should be stored as an array, I’ll define it somewhere at the top of the document; you could probably throw it in a header file. Here’s our test program:

const int program[] = {

PSH, 5,

PSH, 6,

ADD,

POP,

HLT

};

The above program will push 5 and 6 to the stack, execute the add instruction which will pop the two values that are on the stack, add them together and push the result back on the stack. We then pop the result since our pop instruction will print the value (for debugging purposes).

Finally, the HLT instruction means terminate the program. This is used so that if we had control flow we can terminate the program whenever. Our virtual machine will terminate itself naturally if we had no instructions left, though.

Now we have to implement the instruction cycle (fetch, decode, execute). Technically we don’t really have to decode anything. This will make more sense later.

**Fetching the current instruction**[**#**](https://web.archive.org/web/20200121100942/https:/blog.felixangell.com/virtual-machine-in-c/#fetching-the-current-instruction_2)

Because we have stored our program as an array, it’s simple to fetch the current instruction. A virtual machine has a counter, typically called a Program Counter, Instruction Pointer, … these names are synonymous and your choice is personal preference. Usually they are shortened to PC or IP respectively.

If you remember before, I said that we would store the program counter as a register… we will do that, but later on. For now, we’ll just create a variable at the top of our code called ip, and set that to 0:

int ip = 0;

This ip stands for instruction pointer. The program itself is stored as an array of integers. The ip variable serves as an index in the array as to which instruction is currently being executed.

int ip = 0;

int main() {

int instr = program[ip];

return 0;

}

If we were to printf the instr variable, it should give us PSH (or 0). We can write this as a fetch function like so:

int fetch() {

return program[ip];

}

This function will return the current instruction when called. So, what if we want the next instruction? We just increment the instruction pointer:

int main() {

int x = fetch(); // PSH

ip++; // increment instruction pointer

int y = fetch(); // 5

}

So how do we automate this? Well we know that a program runs until it is halted via the HLT instruction. So we just have an infinite loop that will keep looping until the current instruction is HLT.

#include <stdbool.h>

bool running = true;

int main() {

while (running) {

int x = fetch();

if (x == HLT) running = false;

ip++;

}

}

This will work perfectly fine, but it’s kind of messy. What we’re doing is looping through each instruction, checking if the value of that instruction is HLT, if it is then stop the loop, otherwise eat the instruction and repeat.

**Evaluating an instruction**[**#**](https://web.archive.org/web/20200121100942/https:/blog.felixangell.com/virtual-machine-in-c/#evaluating-an-instruction_2)

So this is the gist of our Virtual Machine, but we can do better. A virtual machine is so simple that you can write a huge switch statement. In fact, this is usually the best way to do it in terms of speed, as opposed to say a HashMap for all the instructions and some abstract class or interface with an execute method.

Each case in the switch statement would be an instruction that we defined in our enum. The eval function will take a single parameter, which is the instruction to evaluate. We won’t do any of the instruction pointer increments in this function unless we’re consuming operands.

void eval(int instr) {

switch (instr) {

case HLT:

running = false;

break;

}

}

Let’s add this back into the main loop of the virtual machine:

bool running = true;

int ip = 0;

// instruction enum

// eval function

// fetch function

int main() {

while (running) {

eval(fetch());

ip++; // increment the ip every iteration

}

}

**The stack!**[**#**](https://web.archive.org/web/20200121100942/https:/blog.felixangell.com/virtual-machine-in-c/#the-stack_2)

Great, that should work perfectly. Now before we add the other instructions, we need a stack. The stack is a very simple data structure. We’ll be using an array for this rather than a linked list. Because our stack is a fixed size, we don’t have to worry about resizing/copying. And it’s probably better in terms of cache efficiency to use an array rather than a linked list!

Similarly to how we have an ip that indexes the program array, we need a stack pointer (sp) to show where we are in the stack array.

Here’s a little visualisation of our stack data structure:

[] // empty