CAM: The C Abstract Machine

In which our hero learns about the abstraction on which C programs are built, and start developing a mental model of the system.

4.1 Your Mental Model

According to some folks at Princeton University, mental models are “...are psychological representations of real, hypothetical, or imaginary situations. They were first postulated by the American philosopher Charles Sanders Peirce, who postulated (1896) that reasoning is a process by which a human ‘examines the state of things asserted in the premisses, forms a diagram of that state of things, perceives in the parts of the diagram relations not explicitly mentioned in the premisses, satisfies itself by mental experiments upon the diagram that these relations would always subsist, or at least would do so in a certain proportion of cases, and concludes their necessary, or probable, truth.’” Pretty fancy, huh? But what do you expect from old King’s College, anyway? [P17]

One of our first goals in this book is to help you develop your mental model of how a computer works, through the lens of a running C program. Having a good mental model means you basically understand what is happening as a program executes. Having a good mental model is critical for many aspects of what you do as a computer scientist or programmer, as the model will help you write better programs, debug problems more readily, tune performance, and other important tasks common in programming and systems building.

To aid in this process, we will introduce an abstraction of a computer system in this chapter that we call the C Abstract Machine, or CAM for short. This model will be quite simple at first, and grow more sophisticated as we describe various language features.
4.2 The CAM Memory Model

We will assume throughout this part of the book that we are only concerned with the execution of a single program written in C. Specifically, we focus first on memory; how does such a program utilize memory?

In the CAM memory model, we will assume that this program has its own private memory which is very large, and that in that memory reside the program’s code as well as most data it is operating upon. Figure 4.1 shows how code and data might be laid out in memory; as you can see, all of the data (and, in fact, all of the code) need not be in one contiguous region of memory, but rather can be spread around (with unused portions in-between).

In reality, of course, many programs are running on your computer seemingly at the same time, and each of them thinks it has the entire memory of the machine to itself; in reality, however, each running program only has access to a subset of memory, and is prevented from interfering with the memory of other programs. The illusion of a per-program private memory is provided by the operating system, which uses virtualization techniques to seemlessly provide such an abstraction. Thus, for now, we will ignore other programs and just focus on a single program; if you want to learn more about operating systems, you’ll have to look elsewhere [AD17].
4.3 The C Instruction Pointer

At the machine level, all code and data is of course in binary format. Instructions, for example, are encoded into specific bit patterns that the CPU can interpret; data, as we will soon see, varies depending on what it represents (a character, an integer, etc.) but is also entirely in binary. However, for the purposes of our abstract machine, we will envision that the C code lives in memory, and that the C Abstract Machine executes one C statement at a time; we will live with this approximation until Part II of the book, where we drop down to the machine level to see what is really going on.

To envision how code is executed, we need something to track where we are in an execution. You can think of this as some kind of indicator, telling the machine which C statement to execute next. We call this indicator the C Instruction Pointer, or CIP for short. As we’ll see in the coming chapters, having a solid understand of how the CIP advances through a program is essential to analyzing and debugging real C code.

An example is shown in the previous figure. Therein, the CIP points to a simple C statement (c=a+b;), which, as you can probably guess (even though we haven’t learned any C yet!) adds the variables \(a\) and \(b\) together and puts the result in a variable called \(c\). After this statement is executed by the CPU, the CIP would advance to the next statement.

4.4 Summary

And now, so soon into this brief chapter, we summarize. Because you don’t know much C (yet), it is hard to say too much more about the C Abstract Machine at this point. As we proceed, look for asides and other comments about it; by the end of this large portion of the book, we should have a fully developed model of how a C program uses memory and performs computation, one C statement at a time.

1. This is a term we invented, and used throughout for clarity, hopefully!
2. If that doesn’t make sense, that too is OK – it will soon! Or, you should find a better book that is much clearer than this one...
References

Remzi Arpaci-Dusseau and Andrea Arpaci-Dusseau  
Arpaci-Dusseau Books, 2008-2017  
Available: http://www.ostep.org  
Shameless self promotion ... is what we live for.

[P17] “What Are Mental Models?”  
Phil Johnson-Laird and Collaborators  
mentalmodels.princeton.edu/about/what-are-mental-models  
A website that has way more information about mental models than you would ever need to know,  
unless you are a psychologist at Princeton, in which case you definitely need to know all of this  
stuff, but then why are you learning about C?