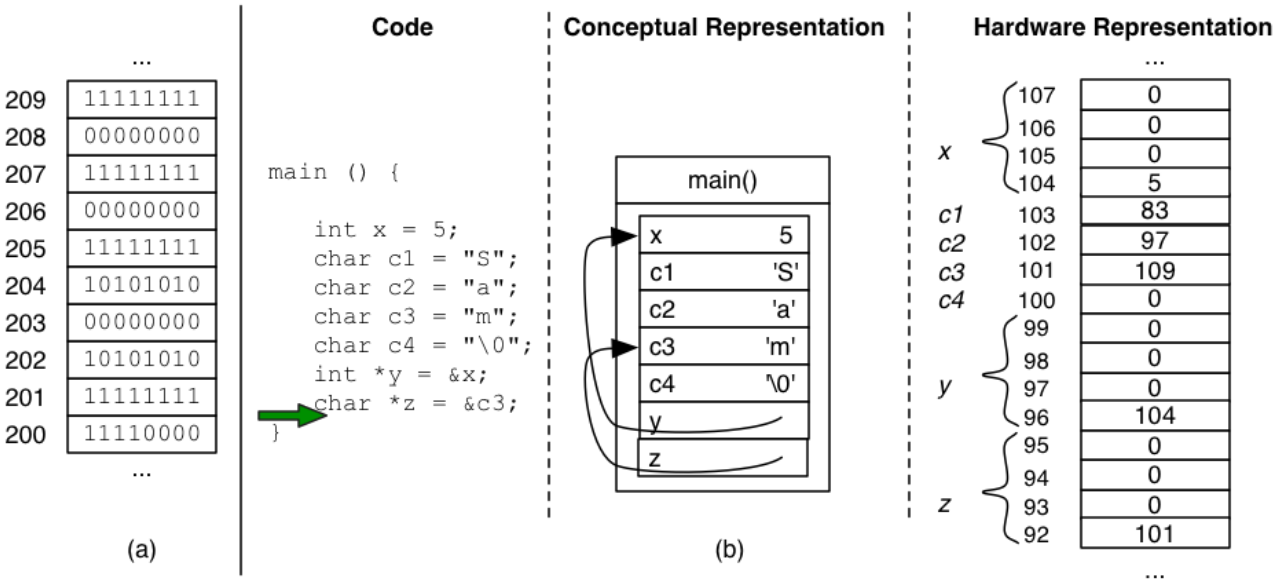
Pointers under the Hood

coursera.org/learn/pointers-arrays-recursion/supplement/Ng2jd/pointers-under-the-hood

In a conceptual drawing, representing a pointer with an arrow is an effective way of showing what the pointer points to. From a technical perspective, however, an arrow does not make much sense. (How exactly does one represent an arrow in hardware? After all, *everything is a number*—and arrows do not *seem* like numbers.)

The mechanics of pointers make a little more sense when we look under the hood at the hardware representation. By now you are familiar with the idea that data is stored in your computer in bytes. Some data types, like characters, require 1 byte (8 bits), and some data types, like integers, require 4 bytes18 (32 bits). When we draw boxes for our variables, we do not necessarily think about how big the box is, but that information is implicit in the type of the variable.



Addressing

A computer keeps track of all of the data by storing it in memory. The hardware names each location in memory by a numeric address. As each different address refers to one byte of data, this type of storage is called *byte-addressable memory*. A visualization of what this looks like is shown in section (a) of the figure above. Each box represents a byte of memory, and each byte has an address shown immediately to the left of it. For example, the address 208 contains one byte of data, with the value *00000000*.

Section (b) in the figure shows the code, conceptual representation, and hardware representation of the declaration and initialization of one 4-byte integer, four 1-byte characters, and finally two 4-byte pointers. Each variable has a base address. For example, the address of *x* is 104 and the address of *c3* is 101. The variable *x* is small enough that it can be expressed within the first byte of its allocated space in memory. If it

were larger (or simply negative), however, the bytes associated with addresses 105-107 would be non-zero. On a 32-bit machine, addresses are 32 bits in size. Therefore pointers are always 4 bytes in size, regardless of the size of the data they point to.

With this more concrete understanding of memory and addresses, the hardware representation of pointers becomes clear: pointers store the addresses of the variable they point to. The final variables, *y* and *z*, in (b) in the above figure show just that. The variable *y* is an integer pointer initialized to point to x. The conceptual drawing shows this as an arrow pointing to the box labeled *x*. The hardware drawing shows this as a variable that has the value *104,* the base address of *x*. The variable z is declared as a pointer to a *char*. Although a character is only 1 byte, an address is 32 bits, and so *z* is 4 bytes in size and contains the value *101*, the location of c3. (If these variables were located in memory locations with higher addresses, the numerical values of the addresses would be larger, and there would be non-zero values across all four bytes of the pointers *y* and *z*.)

Pointers are variables whose values are addresses. An important consequence of this fact is that a pointer can only point to *addressable* data. Expressions that do not correspond to a location in memory cannot be pointed to, and therefore a program cannot attempt to use the ampersand ("address of") operator for these expressions. For example, neither *3* nor (*x+y*) are expressions that represent an addressable "box" in memory. Consequently, lines like these are illegal: *int *ptr = &3*; and int **ptr = &(x + y)*;. Note that *3* and (*x+y*) are not lvalues—they do not name boxes, which is why they cannot have an address. The compiler would reject this code, and we would need to correct it—primarily by thinking carefully about what we were trying to do (drawing a picture). A corollary to this rule is that an assignment statement can only assign a variable that corresponds to a location in memory. So expressions such as *3 = 4;* or *x+y+z = 2* will also trigger a compiler error.