Looking under the Hood

When you are driving a car in traffic, it is probably not a good idea to think too much about what the engine is doing—in fact, you really do not need to know how it works in order to drive. This example illustrates an important concept in programming: *abstraction*—the separation of interface (what something does or how you use it) from implementation (how something works).

Abstraction often comes in multiple levels. Driving a car, the level of abstraction you care about is that the steering wheel turns the car, the gas pedal makes it go faster, and the brake makes it slow down. Your mechanic's level of abstraction is how the pieces of the engine fit together, what level is appropriate for the brake fluid, and if your oil filter is screwed on tightly enough. The engineers who designed the car thought about the physics to make it all work efficiently. At each deeper level, you can think about details that were not important at higher levels, but are still crucial to making the system work. We could continue to lower and lower levels of abstraction until we start thinking about quantum interactions of atoms—fortunately you don't need to worry about that to merge onto the interstate!

There are times, however, when it is a good idea to take a look "under the hood"—to go deeper than the abstraction levels that you typically care about. At the very least, you might want to know whether the car has a diesel engine before filling up the tank, or to be aware that your car has oil, and you should get it changed sometimes.

Similarly, you need not constantly consider the inner workings of your CPU in order to write good code. Thinking about variables as boxes that store values is a good level of abstraction. But, having some knowledge of what goes on under the hood can be important. When you first declare your variables and assign them a type, it is a good idea to pause and consider what this actually means at the hardware level.

As mentioned earlier, a type indicates both a size and an interpretation. The figure above shows you the now-familiar figure with code and its conceptual representation. For this module, we will add a third column, showing you the underlying representation at the hardware level. When you declare a variable x of type int, you should think about this conceptually as a box called x with a value 42 inside. But at a hardware level, the type int means that you have allocated 32 bits dedicated to this variable, and you have chosen for these bits to be interpreted as an integer number in order to yield the value 42.

Hexadecimal

As you may well imagine, reading and writing out a series of 32 of 1's and 0's is tedious at best and error-prone at worst. Because of this, many computer scientists choose to write out the values of numbers they are thinking about in binary using an encoding called *hexadecimal*, or *hex* for short. Hex is base 16, meaning that it represents a number with a 1's column, a 16's column, a 256's column, and so on. As a hex digit can have 16 possible values (0–15), but our base 10 digits have only 10 possible symbols (0–9) we use the letters A-F to represent the values 10-15 in a single digit. The number eleven, for example, is represented by the single digit 'B' in hex.

Numbers represented in binary can easily be converted to hex by simply grouping them into 4-digit clusters, each of which can be represented by a single hex digit. For example, the 4 rightmost bits in the figure above (colored blue) are 1010, which has the decimal value 10 and the hex value A. The next 4 bits in the figure (colored green) are 0010, which has the decimal value 2 and the hex value 2. The remaining 24 bits in the number are all zeroes. Instead of writing out the entire 32 bit binary sequence, we can use 8 digits of hex (0x0000002A) or the shorthand 0x2A. (In both cases, the leading 0x (interchangeable with just x) indicates that the number is in hex.)

Completed