

Data Modeling

Before we can solve a problem with a computer, we first need to represent that problem in a manner that it can be worked on by the computer. That is we must build a "model" of the problem that can be represented by the computer. While sometimes we can represent things precisely (like the exact number of cents in a bank account), other times this model is merely a subsetting or abstracted representation (consider weather prediction; even if we could measure the location and velocity of every gas and water molecule in a storm, even the most powerful computer couldn't store all of that information). Identifying effective and efficient models for representing data on computers is an important part of computer science.

As discussed previously, the data storage hardware (the "memory") of modern computers tend to be organized as a set of consecutive storage elements, each of which can store 1 byte (8 bits) of data. In this section, we discuss common ways to represent data on computers.

Some common types of data items have standard representations on computer systems. For example, integers ("whole" numbers), non-integer numbers, and "characters" used for written language are common primitives by most programming languages and machines. Standard representations are used for these common data types for two reasons: 1) it enables the interoperation of two pieces of code written independently, and 2) operations on these representations can be accelerated on the machine (*i.e.*, most computers can add two arbitrary integers together in a single clock cycle).

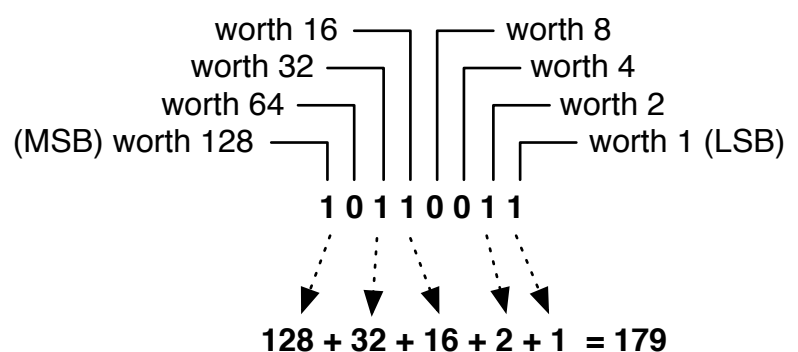
For all other data types, we must construct a representation from the primitive data types provided by the programming language/machine. For example, we might want to build an application for playing tic-tac-toe over a cell phone, and we need a way to represent the game board. Since a tic-tac-toe board is not something that most computer programs need, it won't be provided as a primitive; we'll need to build a representation using the primitives provided (*e.g.*, integers).

Integers: Integers are whole numbers. The representation for integers was overviewed above; the key feature of this representation is that the bits are ordered from a least-significant bit (LSB, typically written on the right) to a most-significant byte (MSB, typically written on the left), and each bit has a weight that is double the previous bit (as shown below). Again, this enables us to represent

2^N different numbers using N bits.
In modern computers, integers are typically represented using either 32 or 64 bits.

Two aspects of integers merit attention:

1) There are two types: unsigned



(for representing non-negative numbers, where 32 bits enables representing all whole numbers between 0 and ~4 billion, inclusive) and signed (where the range is split almost evenly between representing positive and negative numbers and 32 bits is used to represent the numbers from -2 billion to 2 billion). *Note: all of Java's integer types are signed.*

2) The finiteness of the representation: if you repeatedly add positive numbers together, you will eventually compute a number that is too large to represent in the number of bits that you have available. This is called "overflow." While this won't be a big concern in the relatively small programs that we'll write in CS125, it is a serious concern in making real programs.

Question: In a computer that represents unsigned integers using two bits, what is the lowest number that creates an overflow condition? If you add two numbers together, and the result is this number, what do you think the value of each of the two bits would be?

Characters: Computers are frequently used for storage, communication, and presentation of data written in human languages. These languages consist of sequences of "characters" (e.g., the letter 'A', the number '5', and the punctuation mark '!').


On a computer, we store these characters as numbers. For example the letter 'A' is stored as the number 65, the number '5' as 53, and the punctuation mark '!' as 33. These encodings are COMPLETELY ARBITRARY. There is no innate reason why the number 65 should mean the character 'A', but if we all agree to such a standard then we can write software independently that interoperates (i.e., I can write a program that displays a message written by your program). We call this association between a character set and numbers a "**mapping**" or an "**encoding**".

In the early days of computers, the **ASCII** encoding was the dominant standard, which uses a single byte for each character. ASCII has been increasingly replaced by a 2-byte **Unicode** encoding called UTF-16, which is a single encoding capable of representing most of the writing systems in the world and even some emoticons (😄😄😄😄😄😄😄😄😄 etc.).

Strings: When we communicate in human languages, we typically write messages consisting of more than one character. The length of these messages, however, varies; messages can be any number of characters long. For the message "CS125 rocks!", we'd need storage for 11 characters to hold each of the letters, numbers, spaces and punctuation marks. We refer to such a collection of characters as a "character string" or simply a "string."

One of the key challenges of using strings is their variable length. To deal with this problem, one common representation of strings – the one used in Java – is to store the length as an integer along with characters, as shown below. Each character in the string is encoded using the unicode mapping and stored as 16-bit (2 byte) integer values.

length	C	S	1	2	5		r	o	c	k	s	!
11	67	83	49	50	53	32	114	111	99	107	115	33

 = 1 byte

To support new data types, we have to create them. At this point in the course, we will consider two types: enumerated types and compound types.

Enumerated types: These types are for representing one of a fixed number of choices. For example, if you are writing a program to keep track of your road trip, you might need to store which state you slept in each night. We would choose an enumerated type because (until we annex Canada for its oil) there are a fixed number of states. We could assign each state a number (Alabama = 0, Alaska = 1, Arizona = 2... Wyoming = 49). Here we are creating a 'mapping' of each state to an integer. Enumerated types are much like the character type discussed above, but there is no standard encoding of states to numbers, so we'd have to define one.

Because there are only 50 states, a single byte would be sufficient to store a state; each byte can represent up to 256 different values (1 byte = 8 bits $\rightarrow 2^8 = 256$). If we were instead enumerating the seats in the U.S. House of Representatives, which has 435 members, we would need more than a byte, but two bytes would be sufficient (2 bytes = 16 bits $\rightarrow 2^{16} = 65,536$). Each additional byte sure seems to add a lot more representation possibilities, doesn't it? The power of twos!

Compound types: More complex types are generally constructed by composing primitive types. Consider for example, the aforementioned example of a tic-tac-toe game. Each square of the board can be in one of 3 states: unmarked (0), marked with X (1), marked with O (2). We could use an enumeration by mapping 1 byte values to hold the state of a given square, but the board consists of 9 squares. So, to record the complete state of the board we can compose 9 such enumerations together. If we add an additional enumeration to indicate whose turn it is --- 0 for X's turn, 1 for Y's turn --- we can record the complete state of the game, as shown below.

		x		
		1	2	3
y	1		O	
	2		X	
	3	O	X	

<x,y>	<1,1>	<1,2>	<1,3>	<2,1>	<2,2>	<2,3>	<3,1>	<3,2>	<3,3>	turn
	0	0	2	2	1	1	0	0	0	0

= 1 byte

To be able to interpret the data structure, we not only need to know that the first 9 bytes are the states of each square, but we need to know which byte corresponds to which square. Recall that memory consists of a 1-dimensional series of consecutive locations where we can store one byte each, so we necessarily need a mapping from the tic-tac-toe board's 2-dimensional grid to our 1-dimensional memory. It is totally up to the data structure designer to pick a mapping, as there is generally no single right answer. In the figure, each square of the tic-tac-toe board has been identified with an $\langle x, y \rangle$ coordinate pair and the enumerations are stored in memory by sequencing through all values of y for each x position.

Question: Design your own composite type for representing a Sudoku game (en.wikipedia.org/wiki/Sudoku). What are the decisions you will have to make? How many bytes will the representation take?

Storage Efficiency/Ease of Use Trade-offs:

The tic-tac-toe data structure provides a good example of a common theme in data modeling: the trade-off between how much space a data structure or type requires and how easy it is to read and update. An astute reader will notice that the above representation for the tic-tac-toe game is quite inefficient. With each square having only 3 possible values, there are only 3^9 or 19,683 possible boards. Furthermore, it is not necessary to store whose turn, if we assume X always goes first; if there are more X's than O's on the board it is O's turn, otherwise it is X's turn. As such, we could theoretically represent the state of the game in 2 bytes, 1/5 of the 10 bytes used in the above solution.

While this second encoding is definitely more space efficient, it is also more difficult to use. Rather than looking in the 10th bit to know who goes next, we have to scan the whole board. When drawing the board to the screen we would have to decode the second encoding to know the state of each square, whereas in the first encoding, we can read the associated byte. These are the types of design trade-offs we need to consider in our efforts to develop both *efficient* and *usable* computer programs.

Thought problem:

Identify a piece of data that you care about (chord fingerings on a guitar, the contact information on your cell phone, the instantaneous state of the Illinois-Missouri football game, etc.) and think about how it might be represented in a computer's memory.