Variables and Data Representation

You will recall that a computer program is a set of instructions that tell a computer how to transform a given set of input into a specific output. Any program, procedural, event driven or object oriented has the same three components: input, processing and output. Data must be received via a keyboard, a mouse, a disk drive or some other input device and then stored and manipulated in the computer's memory.

Variables

A **variable** is an area in memory where a particular piece of data is stored. Imagine that the computer's memory is a set of mailboxes designed to store a single piece of data. Each mailbox represents a variable. This analogy is frequently used to explain the concept of a variable to beginning programmers. I think it's a little simplistic but …

Each variable has a name, physical size and data type:

- name
    - The word programmers use to refer to the piece of data.
    - In our mailbox analogy, this is the name on the mailbox.
- size
    - The number of bits set aside or allocated for the storage of the piece of data. The amount of memory in a computer is fixed and addresses are numbered numerically. Consequently, each variable is allocated a fixed amount of space when the programmer declares his/her intention to use it.
    - Remember that each mailbox stores exactly 1 piece of data. That means that we could use mailbox storage space more efficiently if we had: tiny postcard-size mailboxes, small letter-size mailboxes, medium magazine-size mailboxes and large package-size mailboxes.
- data type
    - The kind of data that the variable can contain. The computer can store all kinds of different data: characters, sets of characters or strings, whole numbers or integers, real numbers or floating point numbers, true or false values or booleans to name a few.
    - Postcards and only postcards get stored in our tiny mailboxes. Postcards cannot be stored in small, medium or large mailboxes. A letter cannot be "shoved" into tiny mailboxes even if it is tiny and might fit.

Variables are stored in the computer's memory. Memory is composed of digital electronic circuits. Each circuit has 2 states – on and off. That means that storing data in a variable involves encoding or representing that data in a series of circuits that can be either on or off. No matter what the data type of the variable is, the only way to store the data is by combining the "on"s and "off"s in some understandable way. As you might imagine, the encoding schemes can be pretty complex. Several problems can arise for programmers that are a direct result of the way in each type of data is represented in memory.

Representing Unsigned Integers

It is relatively easy to understand how whole numbers or integers are represented in the computer's digital electronic memory. Restricting the discussion to positive numbers or **unsigned numbers** makes it even easier. Let's start there.

Remember that the circuits that make up memory have exactly 2 states – on and off. We could use a 0 to represent off and a 1 to represent on. You may even have noticed that the power switch on one of your electronic devices at home is labeled with 0 and 1 rather than the word "power" or "on/off". The set of digits 1001 means that there are 4 circuits and that the first one is one, the next 2 are off and the last one is on. Seems easy enough. Right?

1001 is an example of a **binary** or base 2 number. Each **binary digit** or **bit** can have one of 2 values, 0 or 1. The binary number 1001 is equivalent to 9. It gets its value from the sum of the positional values of each bit. The right most position has a value of 1 or 2^0 . The next position to the right has a value of 2 or 2^1. The position to the right of that has a value of 4 or 2^2 and so on. The decimal number system works exactly the same way, but you probably don't remember talking about it in 2nd or 3rd grade math!

$1001_{10} = (1 * 1) + (0 * 10) + (0 * 100) + (1 * 1000)$

$1001_2 = (1 * 1) + (0 * 2) + (0 * 4) + (1 * 8) = 9_{10}$

The chart below shows the value of each position in an 8 bit binary number as well as several numbers represented in decimal and binary notation.

Value of Each Position								
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	In Base 2 Exponential Notation
128	64	32	16	8	4	2	1	In Decimal Notation
Binary Examples								Decimal Equivalent
1	1	0	0	0	1	1	1	128 + 64 + 4 + 2 + 1 = 199
0	0	0	1	0	1	0	1	16 + 4 + 1 = 21
1	0	1	0	1	0	0	0	128 + 32 + 8 = 168

Now It's Your Turn

1. Convert each of the following decimal integers into its 8 bit binary equivalent.
   a. 5 = ?
   b. 13 = ?
   c. 27 = ?
   d. 31 =?
   e. 63 = ?

2. Convert each of the following unsigned binary integers into its decimal equivalent.
   a. 00010011 = ?
   b. 00111100 = ?
   c. 00001111 = ?
   d. 01111111 = ?
   e. 01010101 = ?r

Remember our discussion of variables? I said that each variable had a size or a specific number of circuits that are assigned to store that data. Assume that I have a variable that has been allocated 8 circuits or 8 bits in memory. The largest unsigned binary number that can be represented in 8 bits is 11111111 or 255. That means that a programmer can't try to put 256 in that variable. It just plain won't fit! Some of you will, no doubt, want to know what happens if you try to do it anyway. 256 will "overflow" the 8 bits allocated to the variable. Has your computer ever displayed an error message containing the word overflow?

Some programming languages aren't user friendly enough to display an error message and cause the program to terminate. C++, for example, will do its best to try to shove 256 in an 8 bit variable if a programmer tells it to. Displaying the value of the variable on the screen will produce very unexpected results. In order to understand why the number on the screen is negative, you need to understand the representation of signed integers and binary addition.

Representing Signed Integers

1 + -1 should equal 0, right? The 8 bit binary representation of –1 should be the binary number that produces 000000000 when added to 00000001. This problem is a little difficult to solve without knowing how to add binary numbers.

$$\begin{array}{rr} & 00000001 \\ + & ???????? \\ \hline & 00000000 \end{array}$$

$$\begin{array}{rrr} & 000000^{1} & 01 \\ + & 000000 & 01 \\ \hline & 000000 & 10 \end{array}$$

$1_2 + 1_2$ doesn't equal 2 because 2 isn't a binary digit. But 10_2 is the same thing as 2_{10}. You write a 0 in the right most position, carry the 1 into the next place to the left and then add that column of number. 0 + 0 + the 1 that you carried in the previous step is 1. 1 is a valid binary digit so you write the 1 in the 2nd column from the right and repeat the process. As luck would have it, all of the other binary digits are 0 and 0 + 0 is still 0, even in binary addition!

The examples below illustrate 11 + 7 = 18, 23 + 37 = 60 and 1 + -1 = 0 respectively.

$$\begin{array}{rrrrrrr} & 000^{1} & 0^{1} & 1^{1} & 0^{1} & 1 & 1 \\ + & 000 & 0 & 0 & 1 & 1 & 1 \\ \hline & 000 & 1 & 0 & 0 & 1 & 0 \end{array} \qquad \begin{array}{rrrrrrrr} & 00 & 0 & 1^{1} & 0^{1} & 1^{1} & 1 & 1 \\ + & 00 & 1 & 0 & 0 & 1 & 0 & 1 \\ \hline & 00 & 1 & 1 & 1 & 1 & 0 & 0 \end{array} \qquad \begin{array}{rrrrrrrrr} & {}^{1}0 & {}^{1}0 & {}^{1}0 & {}^{1}0 & {}^{1}0 & {}^{1}0 & {}^{1}0 & 1 \\ + & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ \hline & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

Now It's Your Turn

3. Add each of the following pairs of 8 bit binary numbers.
   a. 00010101 + 00001101 = ?
   b. 00111110 + 00101001 = ?
   c. 00011111 + 00000001 = ?
   d. 01010101 + 00111111 = ?
   e. 00000001 + 11111111 = ?

-1_{10} is represented as 11111111_2. The left most bit is referred to as the **<u>sign bit</u>**. The binary representation of a negative number has a 1 in the sign bit. Positive numbers have a 0 in the sign bit. Because one bit of the 8 bits is used to record the sign of the number, only 7 bits can be used to store the value of the number. That means that the largest positive number that can be stored in an 8 bit variable is 01111111 and the smallest negative number that can be stored in the same variable is 10000000. 01111111_2 is 127 but what decimal number is represented by 10000000? You might guess that 10000000_2 is –0 but you would be wrong. Perhaps you weren't tempted to make that assumption because you already noticed that –1 is not 10000001_2.

The notation used for representing negative integers is called **<u>2s complement</u>**. To convert –1 to its binary representation using 2s complement:

1. Convert the absolute value of –1 to its binary representation — 00000001
2. Flip each of the bits in the number from step 1 to the "opposite" binary digit. Complement means opposite. — 11111110
3. Add 1 to the result from step 2.
4. The result from step 3 is the 2s complement representation of –1.

```
   11111110
+  00000001
   11111111
```

This process isn't intuitive but with practice you can become adept at representing negative numbers using 2s complement. It is important to remember that 2s complement representation allows the computer to add positive and negative integers correctly. It's not pretty but it works!

Now It's Your Turn

4. Convert each of the following decimal integers into its 8 bit signed binary equivalent.
   a. -3 = ?
   b. -18 = ?
   c. -25 = ?

You can use a very similar process to determine the absolute value of a binary number that you know is negative. You know that 10000000_2 is the smallest negative number that can be represented in 8 bits but don't know what its decimal equivalent is. To determine the absolute value of the smallest negative number that can be represented in 8 bits:

1. Start with the binary representation of the negative number. — 10000000
2. Flip each of the bits. — 01111111
3. Add 1.
4. The result is the binary representation of the absolute value of the original number. In this case it's 128.

```
   01111111
+  00000001
   10000000
```

5. –128 is the decimal equivalent of the original signed binary number.

It may seem curious that the range of signed integers that can be represented in 8 bits is –128 through + 127. Doesn't –128 through +128 seems more reasonable? After all, shouldn't there be an equal number of positive and negative integers? Ah, but you're forgetting that 0, which is a positive number, doesn't have a complement. The 128 positive numbers that can be represented in 8 bits are 0 through 127. The 128 negative numbers that can be represented are –1 through –128.

Now It's Your Turn

5. Convert each of the following signed binary numbers into its decimal equivalent.
   a. 11001000 = ?
   b. 10111101 = ?
   c. 11111110 =

```
   01111111
+  00000001
   10000000
```

I started this discussion by wondering out loud about what would happen when you tried to put a number that is too large into a variable. The illustration starts with the binary representation of 127, the largest positive number that can be represented in an 8 bit signed integer, and adds 1. The resulting binary number looks OK, until you recall that the left most bit is the sign bit. Representing a bigger number causes the value to "overflow" into the sign bit. Some programming languages display an error message when an **overflow** occurs. Other programming languages, most notably C and C++, interpret the pattern of bits that result as an 8 bit signed integer and use –128 for the value in the variable with no error or warning. This behavior can result in some very strange output!

Many programming languages have a variety of data types that can be used to store integers of different sizes to deal with the overflow problem. Java has 4 integer data types: an 8 bit **byte**, a 16 bit **short**, a 32 bit **integer** and a 64 bit **long**. As a programmer, it is your responsibility to know the range of values that can be stored in each integer data type and how your programs will respond when you exceed this range. Knowing something about the underlying representation of integer data makes this easier to understand.

Representing Integers in Hexadecimal Notation

A 64 bit string of 1s and 0s is a very unwieldy number. For that reason, **hexadecimal** notation, or **hex**, is often used rather than binary notation when representing large integers. When Windows displays the general protection fault error message, for example, the problem address is displayed in hex.

Hexadecimal numbers use base 16. That means that 16 possible digits are used and each place value is a power of 16. 0 through 9 are used for the first 10 digits. A through F are used for digits with the value of 10_{10} through 15_{10} respectively.

Hex is a great shorthand notation for binary because 4 binary digits can be represented in 1 hex digit. A 16 bit address can be represented in 4 hex digits and a 64 bit Java long integer can be represented in 16 hex digits. The chart below illustrates these concepts and several examples.

Value of Each Position				
16^3	16^2	16^1	16^0	In Base 16 Exponential Notation
4096	256	16	1	In Decimal Notation
Hex/Binary Examples				Decimal Equivalent
0	7	A	3	(7*256) + (10*16) + 3 = 1955
0000	0111	1010	0011	
0	F	C	1	(15*256) + (12 * 16) + 1 = 4033
0000	1111	1100	0001	
1	B	4	D	(1*4096) + (11*256) + (4*16) + 13 = 6989
0001	1011	0100	1101	

Now It's Your Turn

6. Convert each of the following signed binary numbers into its hexadecimal equivalent.
   a. 01011101 = ?
   b. 01110111 = ?
7. Convert each of the following hexadecimal numbers into its binary equivalent.
   a. 19 = ?
   b. 89 = ?
   c. A6 = ?
   d. 23 = ?
   e. 51 = ?
   f. CB = ?

Representing Real Numbers

The standard for representing real numbers or as programming geeks would say, "floating point numbers" was established by the International Electronic and Electrical Engineering organization or IEEE. It is much too complicated for most "normal people" to understand, and therefore, I'll point out the just key features. Those of you who are engineers can get the rest of the story from the IEEE web site!

Like integers, **floating point numbers** have a sign and a "value" that have to be represented. Unlike integers, the location of the decimal point also has to be stored. That means that the representation has to have 3 distinct parts. In illustration of a 32 bit real number is given below.

Sign (1 bit)	Exponent (8 bits)	Mantissa (23 bits)	Sign * Mantissa * 2 $^{\text{Exponent}}$
0	00001010	00000000000000000011010	$1*26*2^{10} = 26*1024 = 26624$
0	11110110	00000000000000000011010	$1*26*2^{-10} = 26*(1/1024) = 26 * 0.0009765625 = 0.025390625$

Notice that the exponent is used with a base of 2 rather than a base of 10 as in scientific notation. That means that the exponent portion has an impact on both the "position of the decimal point" and the "precision" of the number. Because a finite number of digits can be used for the exponent and the exponent impacts the precision of the number, there is a limit on the precision of any real number represented using this notation. And that means that some real numbers cannot be represented "exactly"!

Actually, the fact that real numbers cannot be represented exactly in memory is due to the nature of real numbers and digital electronic circuits rather than the IEEE notation itself. The real numbers are a **continuous number system** and graphing the set of real numbers produces a solid line. Between every pair of numbers there is always another real number. Computers, however, are **discrete** machines. Each circuit is on or off and cannot be something in between. It is impossible to make a discrete system behave exactly like a continuous system and one of the consequences for computers and real numbers is that real numbers must be approximations.

What does that mean for you as a programmer? A user may enter a real number such as 3.001 as input to a program. That same data may be displayed on the screen later as 3.0009985 or 3.0010012. Both of these numbers are very close approximations of 3.001 but they aren't exactly the number the user entered. The program may then use that data in several calculations and all of the results of those calculations will be "off" a little. Lots of computer users have a similar experience when summing a column of numbers that are the result of complex calculations in a spreadsheet. "Round off" errors are a fact of life in calculations involving real numbers in any computer program. A programmer can minimize those errors by using a data type that represents real numbers with more precision.

Representing Character Data

A similar kind of coding scheme must be used to represent **character** data so that the "a" that the user enters at the keyboard can be stored in the digital circuits in memory. The **ASCII** character set assigns a numeric value to each of the printable characters on a typical keyboard used in the US along with useful non-printing characters. It allows computers to store character data digitally, to compare characters, and therefore, to sort data "alphabetically" as well.

The chart below lists a subset of the ASCII character set and the decimal value given to each of those characters. B, for example, has an ASCII value of 66 and is represented as the 8 bit unsigned binary number 01000010 or 42 hex. ~ has an ASCII value of 126 and is represented as the 8 bit unsigned binary number 01111110 or 7E hex.

Please note that A and a are different characters and have different ASCII values. In fact, all of the upper case letters precede all of the lower case letters in the ASCII character set. Computers will sort a set of word beginning with an upper case letter before a set of word that begin with a lower case letter. Aardvark and Bat will most likely sort before apple in a computer program. This behavior is distressing for many computer users but makes perfect sense once you are familiar with the ASCII character set.

Some programming languages represent character data using other character sets. The **Unicode** character set was developed because the ASCII character set was insufficient for representing all of the symbols used in international languages. Each Unicode character is represented by 16 bits rather than 8 bits, but the concept is exactly the same. The characters that are part of the ASCII character set have the same values in the Unicode character set.

A Subset of the ASCII Character Set

Char	Decimal Value	Char	Decimal Value	Char	Decimal Value	Char	Decimal Value
Space	32	9	57	R	82	k	107
!	33	:	58	S	83	l	108
“	34	;	59	T	84	m	109
#	35	<	60	U	85	n	110
$	36	=	61	V	86	o	111
%	37	>	62	W	87	p	112
&	38	?	63	X	88	q	113
‘	39	@	64	Y	89	r	114
(	40	A	65	Z	90	s	115
)	41	B	66	[	91	t	116
*	42	C	67	\	92	u	117
+	43	D	68	]	93	v	118
‘	44	E	69	^	94	w	119
-	45	F	70	_	95	x	120
.	46	G	71	‘	96	y	121
/	47	H	72	a	97	z	122
0	48	I	73	b	98	{	123
1	49	J	74	c	99	|	124
2	50	K	75	d	100	}	125
3	51	L	76	e	101	~	126
4	52	M	77	f	102		
5	53	N	78	g	103		
6	54	O	79	h	104		
7	55	P	80	i	105		
8	56	Q	81	j	106		

Now It's Your Turn

8. Convert each of the following characters into the binary equivalent of its ASCII value.
    a. Q = ?
    b. q = ?
    c. + = ?
    d. 0 = ?
    e. 1 = ?
9. Convert each of the binary representations of any ASCII value to its equivalent character.
    a. 01100011 = ?
    b. 01000011 = ?
    c. 00110010 = ?
    d. 00110011 = ?

Now that you understand how data is stored in variables in memory and some of the programming implications of the representation of data, we can begin to talk about the process of creating procedural programming.

Author: Mari Good
Revised: Brian Bird 1/7/10