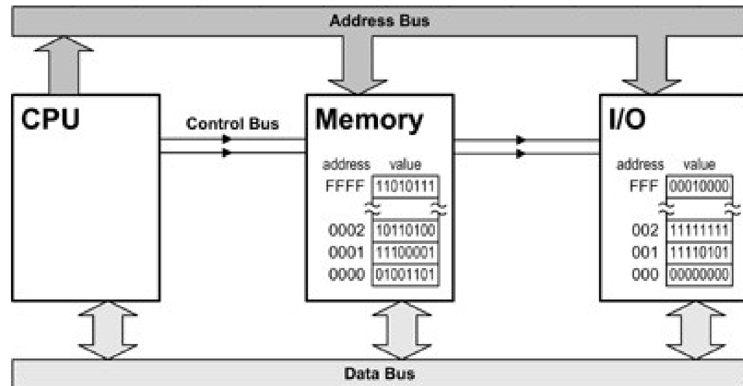
Chapter 2 – Native Data Types, Memory, Addressing, and the CPU

Identifiers are simply what we call pieces of code and data. Identifiers must start with a letter followed by any number of alphanumerics (0-9 & a-z) and/or underscores. Identifiers make code readable; so instead of saying a piece of data is located at memory address 19864, we can just refer to it as *lineNumber*. And if you add more code, the compiler can automatically adjust that number rather than you having to do it manually. It is also best practice to follow naming conventions; these tend to vary depending on the language. The conventions generally make reading code a little easier. Because compilers are pretty quick, we have a lot of memory, and there are auto-complete features in IDE's; there is no excuse to use cryptic identifiers such as *ln* instead of *lineNumber*. For multi-word identifiers, we can capitalize the start of a new word or use underscores, such as *line_number*. Finally, you can't use identifiers that are reserved words or keywords such as *if*, *for*, and *switch*. Note that identifiers are case-sensitive; when you have an error with an identifier, make sure the case matches.

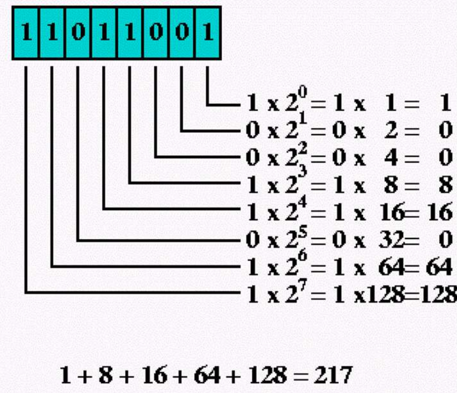
Exercises: Put all the non-coding answers into a single Word document for this chapter.

Exercise 2.1: Find the link to the naming conventions and reserved words (can also try keywords) for your chosen language. Also name 4 reserved words.

It doesn't matter which language you choose, the basic native data types are: integers, floating-point numbers, and characters. The computer treats EVERYTHING as numbers. The basic unit of memory is a byte which is typically 8 bits. A bit is either 0 or 1; its value depends on the bit position. This is similar to decimal places; a digit in the hundreds place has a different value than a digit in the ones place. 8 bits hold integer values between 0 to 255 for unsigned or -128 to 127 for signed. An *int* type is generally defined as the data bus size. So for a 32-bit processor that is 4 bytes (4,294,967,296) and a 64-bit processor is 8 bytes (1.85×10^{19}). The first Apple, Commodore, and Atari had 8-bit processors. The first IBM PC had a 16-bit processor. The address bus can have a different number of bits than the data bus. Currently, we can't have 1.85×10^{19} bytes of RAM (18 Giga-Giga bytes), so the address bus is typically smaller than the data bus. The 8-bit processors of yesteryear typically had a 16-bit address bus that means that it had 2^{16} or 64K of RAM. Choosing data and address bus sizes is a matter of what is practical. Each bit has to be physically wired from the Central Processing Unit (CPU) to memory and I/O (Input/Output).



Because the computer represents numbers with bits, it is useful to understand binary and hexadecimal numbers:



decimal	hexadecimal	binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Converting the same number (10011011_B), we group the digits by 4, starting from the right:

1011 = B (the last 4 digits)
1001 = 9 (the first 4 digits)
So $10011011_B = 9B_H$

When the context is obviously hexadecimal, you can leave out the (H). Another way of specifying hexadecimal is with a ($), so $9B.

Exercise 2.2: Find the decimal and hexadecimal value of the binary value: 10101101_B

Characters are nothing more than numbers. These numbers are associated with indexing into a font. So, here are the ASCII codes. If you are doing different languages (human, not computer), then you will have to switch to Unicode; there are no specific number of bytes for Unicodes.

Knowing the codes is useful when you are trying to debug your program.

ASCII control characters		
00	NULL	(Null character)
01	SOH	(Start of Header)
02	STX	(Start of Text)
03	ETX	(End of Text)
04	EOT	(End of Trans.)
05	ENQ	(Enquiry)
06	ACK	(Acknowledgement)
07	BEL	(Bell)
08	BS	(Backspace)
09	HT	(Horizontal Tab)
10	LF	(Line feed)
11	VT	(Vertical Tab)
12	FF	(Form feed)
13	CR	(Carriage return)
14	SO	(Shift Out)
15	SI	(Shift In)
16	DLE	(Data link escape)
17	DC1	(Device control 1)
18	DC2	(Device control 2)
19	DC3	(Device control 3)
20	DC4	(Device control 4)
21	NAK	(Negative acknowl.)
22	SYN	(Synchronous idle)
23	ETB	(End of trans. block)
24	CAN	(Cancel)
25	EM	(End of medium)
26	SUB	(Substitute)
27	ESC	(Escape)
28	FS	(File separator)
29	GS	(Group separator)
30	RS	(Record separator)
31	US	(Unit separator)
127	DEL	(Delete)

ASCII printable characters					
32	space	64	@	96	`
33	!	65	A	97	a
34	"	66	B	98	b
35	#	67	C	99	c
36	$	68	D	100	d
37	%	69	E	101	e
38	&	70	F	102	f
39	'	71	G	103	g
40	(	72	H	104	h
41	)	73	I	105	i
42	*	74	J	106	j
43	+	75	K	107	k
44	,	76	L	108	l
45	-	77	M	109	m
46	.	78	N	110	n
47	/	79	O	111	o
48	0	80	P	112	p
49	1	81	Q	113	q
50	2	82	R	114	r
51	3	83	S	115	s
52	4	84	T	116	t
53	5	85	U	117	u
54	6	86	V	118	v
55	7	87	W	119	w
56	8	88	X	120	x
57	9	89	Y	121	y
58	:	90	Z	122	z
59	;	91	[	123	{
60	<	92	\	124	|
61	=	93	]	125	}
62	>	94	^	126	~
63	?	95	_		

The hardest type of native data type to understand is floats (*float* or *double*) but they are the easiest to use, however they are also easy to mis-use (more on this later). The easiest description is that it is similar to scientific notation, but done in binary. Most computers follow IEEE 754 convention for floating point. You are not required to know this format, just understand that looking at the binary value generally doesn't easily make sense. Depending on what type of precision and how many calculations are necessary, dictates whether you need a *float* or *double*. The more calculations you do, the more likely you need to use *double* to avoid rounding errors; this is because every time you do a calculation, you are likely to lose some precision.

Exercise 2.3: Find a link to a description of IEEE 754 and tell me how many significant decimal digits are in a 32-bit and a 64-bit floating point numbers.

The last native data type is *boolean*. The number of bits depends on the compiler and language. It holds a truth value: *true* or *false*. The actual values are non-zero for *true* and zero for *false*. Booleans will be covered in detail later.

Variables are defined using identifiers. Again, the compiler assigns a memory location for you. Depending on what you are defining, it uses a certain amount of bytes to hold the data. If the data is the bus size, then we call this a *word*, similar but not identical to an *int*. So on a 32-bit CPU, 32 bits is a *word*. And on a 64-bit CPU, 64 bits is a *word*. Why do we want variables? – The computer can really only do on calculation at a time, so when we do a calculation, we need to store it somewhere while we do other calculations. The more complex an algorithm, the more variables we need to perform the algorithm.

Exercise 2.4: Look back at the naming conventions and name 2 of them for variables.

Methods (or procedures or functions) are also defined using identifiers. This simplest way to view methods is a piece of code that is usually re-usable. So, instead of writing several lines of code to find the maximum number in one list and writing several more lines of code to find the maximum number in another list. It is best to write a method that finds a maximum in a list and just call it for each list; also, this is good coding practice because if you code it correctly, you can just keep using it without having to worry about getting the wrong result. The standard libraries have common methods that do a lot of work for you, so it is important to understand how to use and create them, eg: *abs*(x), *sin*(x), *cos*(x), *sqrt*(x), ...

Exercise 2.5: Look back at the naming conventions and name 2 of them for methods.

Printing just strings isn't very useful. To make programs useful, it needs to compute and/or evaluate things. For mathematic expressions, it follows the rules of BEDMAS. So 3+4*7-12/2 is evaluated as 3+(4*7)-(12/2) = 3+28-6 = 25. When you want a method to return a value, you must specify what type of variable or object that is being returned; and the method must return a value for every *return* statement otherwise you will get an error. Look at Example_2_1 and notice the difference between printing with and without quotes.

Exercise 2.6: Modify Example_2_2 to include subtract, multiply, and divide methods; make the printout similar to add. <u>Do not</u> print the solution as strings. Sample printout:

```
adding 14 + 7 = 21
subtracting 14 – 7 = 7
multiplying 14 * 7 = 98
```

```
dividing 14 / 7 = 2
```

There aren't too many interesting things that we can do until we learn some more ways to control the computer. The previous example is known as hard-coded computing (things don't change). To make it more useful, we have the user enter values (interactive programming). Always prompt the user for input, so that they know what should be entered, especially for sentinels.

Exercise 2.7: Modify Example_2_3 to include the square and cube methods; make a printout that prints the number, its square, and its cube on a single line with blank spaces between them. You only need to enter a single value; two values aren't necessary. Sample printout:

```
Enter number: 5
Number: 5  Square: 25  Cube: 125
```

Programs that terminate after one set of input aren't too useful. We can use *do..while* and *if* statements to loop until a sentinel is entered. These statements will be fully explained later, but for now we want to be able to generate better behaving programs. A sentinel is simply checking for a certain value; a value that we won't normally use in calculations. In the follow example, we enter the number as a string so that we don't exclude any numbers and we can check the sentinel as a string. If the string isn't the sentinel, then we go ahead and convert it to a number.

Exercise 2.8: Modify Exercise_2_4 to include the additive inverse (the negative) of a number. Sample printout:

```
Enter number (Q to quit): 5
Number: 5  Additive Inverse: -5
Enter number (Q to quit): -8
Number: -8  Additive Inverse: 8
Enter number (Q to quit): Q
```

Scope

Here is some pseudocode to look at methods (pseudocode is just instructions that are written in human form rather than in a computer language). When variables are declared inside a method, they are local to that method and only exist from the time the method is entered to the time it exits. If variables are not declared inside any method, then they are global and last as long as the program is executing. The main learning point is understanding where the variables are allocated rather than how the rest of the code works:

```
Let A=24.
Let B=64.
Call common denominator of A and B.
Print CD of A and B.
```

CPU's have a program counter (PC) that points to the next instruction; once an instruction is completed, it goes to the following instruction. So, it would start off at "*Let A=24*" and execute it. Now the PC points to "*Let B=64*" and execute it. Now the PC points to the *call instruction*, but the common denominator method is located somewhere else in memory; so the PC needs to change to that memory address, but how does it get back to our next instruction which is "*Print CD*?" Well, the stack is here to save us; it is a section of memory that uses a data structure called *Last In/First Out* (LIFO). The CPU stores the address of "*Print CD*" on the stack before calling the "*Common Denominator*" method, then it

calls the "*Common Denominator*" method. Once the method is complete, it grabs the "*Print CD*" address off the stack and puts it into the PC. Then the PC happily executes the "*Print CD*" command.

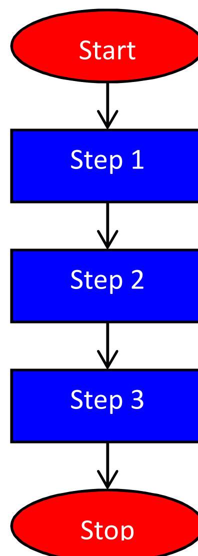
Exercise 2.9: modify Example_2_5 to get user input and loop until a sentinel is entered.
Sample printout:

```
Enter first number (Q to quit): 75
Enter second number: 35
GCD(75, 35) = 5
Enter first number (Q to quit): q
```

It's easy to make everything global and not do any parameter passing. But when you create larger programs, you will have to keep on creating more identifiers. Also, it is not possible to handle multi-tasking efficiently without local variables. So, take the time to learn about scope. Variables can also be created with objects, but we will get to this later.

One thing to note about programming: when you see '=', this is not an equation, it is an assignment. For example, 'x = x + 2', in algebra, results in '0 = 2' which is a false statement. In programming, let's say 'x = 7'; then 'x = x + 2' results in 'x = 7 + 2' meaning 'x = 9'. So, although you may set a variable to a certain value, it is likely to change values as the program is running. That's why it is important to print the value of variables or use the debugger to exam the variables as the program is running. Variable actually means to change, so we use a variable to track the changes. In math, we tend to use variables to represent an unknown quantity that we want to solve.

Visualizing code with flowcharts can make creating programs easier for some people, especially when the logic is more complicated. This flowchart shows the instructions are executed one after another:



Email me the exercise source files (not the whole folder) when you are done.

Image Sources:

http://www.talktoanit.com/A+/aplus-website/images/resources-system.jpg

http://images.tutorvista.com/cms/images/46/double-method.png

http://www.learn44.com/wp-content/uploads/2011/08/Binary-to-Decimal-and-Hexadecimal-Conversion-Memorization-Chart.jpg

http://www.theasciicode.com.ar/american-standard-code-information-interchange/ascii-codes-table.png